Querying Structured Text in an XML Database *

Shurug Al-Khalifa
University of Michigan
Ann Arbor, MI 48109
shurug@eecs.umich.edu

Cong Yu
University of Michigan
Ann Arbor, MI 48109
congy@eecs.umich.edu

H. V. Jagadish
University of Michigan
Ann Arbor, MI 48109
jag@eecs.umich.edu

ABSTRACT

XML databases often contain documents comprising structured text. Therefore, it is important to integrate "information retrieval style" query evaluation, which is well-suited for natural language text, with standard "database style" query evaluation, which handles structured queries efficiently. Relevance scoring is central to information retrieval. In the case of XML, this operation becomes more complex because the data required for scoring could reside not directly in an element itself but also in its descendant elements.

In this paper, we propose a bulk-algebra, TIX, and describe how it can be used as a basis for integrating information retrieval techniques into a standard pipelined database query evaluation engine. We develop new evaluation strategies essential to obtaining good performance, including a stack-based TermJoin algorithm for efficiently scoring composite elements. We report results from an extensive experimental evaluation, which show, among other things, that the new TermJoin access method outperforms a direct implementation of the same functionality using standard operators by a large factor.

1. INTRODUCTION

Boolean style queries against an XML document collection are useful when users are aware of the underlying schema and can specify queries precisely. However, collections of XML documents are frequently heterogeneous, with documents that do not share the same schema [23]. Moreover, users may frequently be satisfied if they find items that are relevant, even if not an exact match, particularly where elements have large textual content. Traditional databases, including XML databases, can efficiently compute answers to very complex queries precisely stated in terms of logic statements, but are not good at dealing with this fuzziness.

Relevance ranking is central to information retrieval, so one may be inclined to turn toward IR to address these problems. However, traditional IR has been "document-centric", where a document is assumed as the basic unit of information to be queried and retrieved. XML gives the user an ability to address into the specific elements of a document, at a much finer granularity, making it important to retrieve result elements at appropriate levels of granularity, and making it important to take document structure into account in ways that traditional IR cannot manage. Furthermore, it is usually not possible to pose IR queries that require more complex analysis beyond the determination of (document) relevance.

In short, information retrieval and database queries have hitherto followed two quite separate paths (with only a few intersections [22, 13, 23, 21] discussed in Sec. 7). With XML as a means of representing structured text, there is a need to integrate these two strands of work. In this paper, we make significant strides in this direction, in terms of both query specification and query evaluation.

Specifically, in Sec. 3, we develop an algebra, TIX, for querying Text In XML, based on the notion of a *scored tree*. Each operator in this algebra manipulates collections of scored trees. By this means, we are able to fold into a database framework the notion of relevance scoring and ranking. Deciding the granularity of elements to be returned to the user is a brand new challenge for IR-style XML query evaluation. By introducing a Pick operator, TIX provides a way of specifying how to select appropriate elements among all the relevant ones in an XML document. In Sec. 4, we suggest simple extensions to the XQuery[7] language that can effectively utilize these algebraic facilities.

Having a fancy framework is of little value unless efficient manipulation is possible. In Sec. 5 we address this issue. The core of XML query processing is generally believed to be the containment join, and a series of recent efforts have studied algorithms for this purpose [25, 2, 9, 6]. In particular, the *stack-based* family of structural join access methods has been shown to be superior to others. We introduce a new access method, TermJoin, that generalizes the stack-based family of algorithms to support the IR-style query processing model. We also propose an algorithm that can efficiently evaluate the Pick operator, as well as a PhraseFinder algorithm.

In Sec. 6, we present the results obtained from running experiments with the new access methods proposed. Among other things, we show that the TermJoin access method performs over 30 times better than a combination of basic access methods.

After discussion of related work in Sec. 7, we conclude in

*Supported in part by NSF under grant IIS-0208852 and IIS-0219513.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD 2003, June 9-12, 2003, San Diego, CA.
Copyright 2003 ACM 1-58113-634-X/03/06 ...$5.00.

Sec. 8. But, first, we begin with a motivating example and defined the problem to be addressed.

2. MOTIVATION

EXAMPLE 2.1. *Consider Query 1 in Figure 2 against the XML database in Figure 1. Such a query may be posed by a user looking for document components relevant to "search engine", and preferably also related to "internet" and "information retrieval".* □

Even such a simple query is not easily transformed into a boolean specification. Choosing "OR" to relate term-occurrence predicates, we retrieve document components relevant only to the two secondary terms but not to the primary term "search engine" (e.g., `#a15`). Choosing "AND" we lose the relevant paragraph (`#a18`) talking about "search engine". Choosing a mixture of "AND" and "OR" may produce better results, but it is hard to determine a suitable query expression that will be applicable over all possible database instances. In short, such queries require *weighting* and *ranking* support in the boolean query engine.

Using traditional IR-style retrieval, however, has a different kind of problem. While it can rank atomic elements (considered as "documents") according to the term preferences provided by the user, it is not clear which elements should be considered. If we choose to rank article, then the one article in the example is returned, with little additional information. The user will have to go through the first two, irrelevant, chapters before finally reaching the third, relevant, chapter. It is also possible this article is discarded, since it contains much irrelevant information, whereby the highly matched information contained in the third chapter is lost. Neither is a desirable outcome. If we choose to rank p elements, the three paragraphs (`#a18, #a19, #20`) under the third chapter, if returned, will be returned separately. The semantic linkage among the three is broken during the process and has to be reconstructed by the user. Furthermore, the section `#a12` within the same chapter may be ranked quite low and discarded. The user therefore will not know that the entire third chapter is perhaps relevant.

Traditional IR approaches typically use an inverted document index [20] to identify documents containing the specified query terms. One alternative would be using inverted indexes to index a term for every ancestor element it is contained in, all the way up to the root. This allows the query engine to evaluate each possible document component and decide which one to return. However, for a term that occurs frequently in a document, the redundancy can quickly cause the size of the index to explode. To address this problem, Fuhr and Großjohann have suggested [13] selecting a subset of the document components as the root of *index objects*, which are treated as "atomic" units for returning. However, choosing the right level of atomic unit is not straightforward, and in fact, may not be possible given a sufficiently heterogeneous query mix. It is important to be able to enhance IR systems to return results at an optimum level of granularity, possibly heterogeneous, the response to the same query could include both entire articles (where all of it is judged relevant) and specific chapters and paragraphs (that are considered relevant even if the rest of the containing article is not).

Furthermore, IR-style XML queries don't have to be standalone. A user having a better knowledge about the structure

articles.xml:
```
<article>#a1
   <article-title>#a2
      Internet Technologies
   </article-title>
   <author id=''first''>#a3
      <fname>Jane</fname>#a4
      <sname>Doe</sname>#a5
   </author>
   <chapter>#a6
      <ct>Caching and Replication</ct>#a7
      ...
   </chapter>
   <chapter>#a8
      <ct>Streaming Video</ct>#a9
      ...
   </chapter>
   <chapter>#a10
      <ct>Search and Retrieval</ct>#a11
      <section>#a12
         <section-title>#a13
            Search Engine Basics
         </section-title>
         ...
      </section>
      <section>#a14
         <section-title>#a15
            Information Retrieval Techniques
         </section-title>
         ...
      </section>
      <section>#a16
         <section-title>Examples</section-title>#a17
         <p> ... Here are some IR based
            search engines: ... </p>#a18
         <p> ... search engine NewsInEssence
            uses a new information retrieval
            technology ... </p>#a19
         <p> ... semantic information retrieval
            techniques are also being incorporated
            into some search engines ... </p>#a20
      </section>
   </chapter>
</article>
```

reviews.xml:
```
<review id=''1''>#r1
   <title>Internet Technologies</title>#r2
   <reviewer>#r3
      <fname>John</fname>#r4
      <sname>Doe</sname>#r5
   </reviewer>
   <comments> ... </comments>#r6
   <rating>5</rating>#r7
</review>
<review id=''2''>#r8
   <title>WWW Technologies</title>#r9
   <reviewer>Anonymous</reviewer>#r10
   <comments> ... </comments>#r11
   <rating>3</rating>#r12
</review>
```

Figure 1: Example XML Database. For the convenience of reference in the paper, a unique identifier is attached to each element at the opening bracket or, in the case of text immediately following the open bracket, at the closing bracket. Text not relevant in the context of example queries is shown in the form of "...".

Query 1: `simple IR-style query`
`Find document components in articles.xml that are about ``search engine''. Relevance to ``internet'' and ``information retrieval'' is desirable but not necessary.`

Query 2: `structured IR-style query`
`Find document components in articles.xml that are part of an article written by an author with last name ``Doe'' and are about ``search engine''. Relevance to ``internet'' and ``information retrieval'' is desirable but not necessary.`

Query 3: `IR-style join query`
`Find relevant document components in articles.xml as specified in Query 2 above. For articles containing such components, find reviews from reviews.xml for articles with similar titles.`

Figure 2: Example IR-style Queries.

of the XML document should be able to put some structural constraints into the query and therefore limit the number of uninteresting results.

EXAMPLE 2.2. *Consider Query 2 in Figure 2, which is the same as Query 1, except the additional restriction that the search be limited to articles with an author named "Doe". The author-name restriction is exactly the sort of predicate databases are so good at evaluating.* □

With the availability of structural information, IR performance can be boosted by taking this structure into account. For instance, a query that requires determining relevance at the article level may best be performed by integrating evidence over various components of the article, using a complex integration function, for instance, one that weights some elements (such as article-title) more heavily than others. Such weighting heuristics are already used by search engines, but are typically hard-wired into the search engine code. We would like to make declarative specification of such complex IR conditions possible in a structured text query framework.

3. ALGEBRA

XML data is commonly modeled as an ordered labeled tree, with each node in the tree corresponding to an XML element and each edge corresponding to an inclusion relationship. To achieve algebraic closure, a query algebra for XML must manipulate collections of such ordered labeled trees.

To model IR-style processing, we introduce the notion of a *scored* tree and define an algebra over collections of scored (ordered, labeled) trees. We call our algebra TIX, for querying Text In XML. We have chosen the name as a play on TAX [17], a bulk algebra for querying XML data, after which we have modeled our algebra for querying XML text. We illustrate how common algebra operators like Selection, Projection, and Join can be defined over scored data trees. Finally, we introduce two new operators called Threshold and Pick, which operate specifically on scored data trees and are central to the introduction of IR facilities into XML query processing.

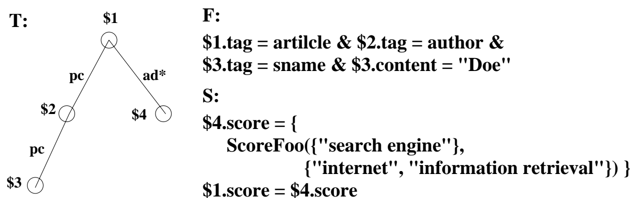


Figure 3: Scored Pattern Tree for Query 2: A Structured IR-style Query. ScoreFoo is a user defined scoring function takes two sets of phrases as parameters and assign a score to the input node. It can be found in Figure 9.

3.1 Scored Trees

Definition 1. **Scored Data Tree:** a *scored data tree* is a rooted ordered tree, such that each node carries data in the form of a set of attribute-value pairs, including at least a `tag` (indicating the type of the node) and a real number valued `score` (indicating the score of the node). The score of the tree is the score of the root node.

The traditional (unscored) data tree can be viewed as a scored data tree with the scores of all nodes being `null`. The score of a node becomes a non-`null` real number after it is matched to a node within a *scored pattern tree*, introduced below, where an IR-style predicate (and therefore a scoring function) is imposed on the node.

Definition 2. **Scored Pattern Tree:** a *scored pattern tree* is a triple $\mathcal{P} = (T, F, S)$, where $T = (V, E)$ is a node-labeled and edge-labeled tree such that:

- each node in V has a distinct integer as its label.
- each edge is labeled `pc` (for parent child relationship), `ad` (for ancestor descendant relationship), or `ad*` (for self-or-descendant relationship).
- F is a formula of boolean combination of predicates applicable to nodes.
- S is a set of scoring functions specifying how to calculate the scores of each node (and their ancestors) being applied with an IR-style predicate, and the scores of each IR-style join condition match.

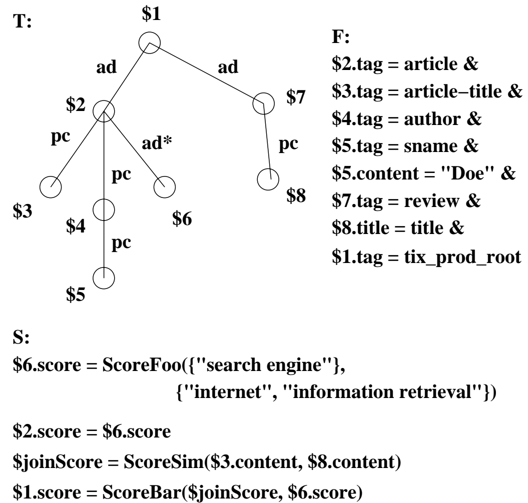
The scored pattern tree constrains nodes in the normal way: the pattern imposes structural requirements on the nodes while the formula imposes value-based constraints. In addition, the set of scoring functions, S, defines how the scores of some nodes (specified in S), including the root node, should be calculated. The traditional (TAX) pattern tree can be viewed as a scored pattern tree with an empty S. Figure 3 shows an example scored pattern tree corresponding to Query 2 in Figure 2.

Some features of the scored pattern tree are illustrated in the example. First, the notion of the `ad*` relationship. We seek articles or more specific units (sections, paragraphs) relevant to our IR query. While `$1` binds to an article element, `$4` binds to this unit, which could be the article bound to by `$1` or any descendant thereof. This sort of relationship

is especially common in IR-style queries against XML. Second, the notion of IR-nodes. A scoring function is defined for each node where an IR-style predicate (relevance finding) is applied to the node and we call those nodes *primary IR-nodes*. In addition, a scoring function is mandatory on a node that has primary IR-nodes in its subtree: the inclusion of a primary IR-node in the subtree automatically converts the otherwise non-IR root node into an IR-node. This ensures that the root of the scored pattern tree will have a scoring function unless there is no IR-node at all in the pattern tree. On the other hand, a node without any primary IR-node node in its subtree can become an IR-node once a scoring function is defined for it based on the scores of other IR-nodes. We call these two kinds of IR-nodes *secondary IR-nodes*. In Figure 3, `$4` is a primary IR node, `$1` is a secondary IR node, while `$2` and `$3` are not IR nodes. To distinguish IR-nodes in the pattern tree from those in the data tree matching the scored pattern tree, we call the former *query IR-nodes* and call the latter *data IR-nodes*.

To keep the arithmetic simple, we have chosen a rather simplistic weighted-sum scoring function in the example for node `$4`. In reality, we would expect the scoring function to be quite complex. For example, a *tf*idf* computation, taking into consideration the element size, may be more representative of what an IR system would do. We can also specify complex conditions. For instance, that the score of node `$4` is 0 unless the term "search engine" occurs at least once, in which case the score is calculated using the weighted-sum function. In many IR systems, the range of a scoring function is restricted to be $[0, 1]$. We could certainly do the same, and obtain all the attendant benefits. We have chosen a scoring function with range $[0, \infty)$ just to make the point that any such range restrictions on the scoring functions are not required for our algebra.

Sometime a scoring function can also be defined on an IR-style join condition. Figure 4 shows an example. The scoring function `ScoreSim` evaluates the degree of similarity between two `title` nodes based on the number of same words occurred in both. (Once again, a real function would be more complex, for example, using vector space [20] cosine similarity). The score generated by this function is attached to the temporary variable `$joinScore` and used subsequently in calculating the score of the root node just like scores attached to the IR-nodes.



S:
$6.score = ScoreFoo({"search engine"}, {"internet", "information retrieval"})
$2.score = $6.score
$joinScore = ScoreSim($3.content, $8.content)
$1.score = ScoreBar($joinScore, $6.score)

Figure 4: Scored Pattern Tree for Query 3: An IR-style Join Query. Like ScoreFoo, ScoreSim and ScoreBar are user defined scoring functions can be found in Figure 9.

3.2 Extension of Existing Operators

Score generating in TIX is done primarily via pattern matching, i.e., the matching of data trees to the scored pattern tree to generate scored data trees. Pattern matching is encapsulated in various operators in TIX, which also manipulate the generated score. In this section, we extend three existing operators for incorporating score manipulation capabilities.

3.2.1 *Scored Selection*

The scored selection operator ($\sigma'_{\mathcal{P}}(\mathcal{C})$) operates on scored trees. It takes a collection of data trees as input, and a scored pattern tree as parameters, and returns a collection of scored data trees as output. Each scored data tree in the output collection matches the scored pattern tree. The score of each data IR-node is calculated using the scoring function on their corresponding query IR-node. Figure 5 shows three representative result trees (The full collection of the result trees contains more than the three listed) after applying the scored pattern tree of query 2 in Figure 3 to the example XML database. Part (c) illustrates the case when the matching data IR-node is exactly the ancestor node in the `ad*` edge.

3.2.2 *Scored Projection*

The scored projection operator ($\pi'_{\mathcal{P},PL}(\mathcal{C})$) takes a collection of data trees as input, a scored pattern tree and a projection list PL (PL specifies the nodes to be retained in the output.) as parameters, and returns a collection of scored data trees as output. Each tree in the output collection can be regarded as an input tree with nodes not matching the scored pattern tree or not being preserved in the PL being eliminated. Figure 6 shows the result of applying the same scored pattern tree as in the previous example to the XML database with $PL = \{\$1, \$3, \$4\}$ (zero-score nodes are removed). It is important to note that the score of a data IR-node in a projection is calculated in two different ways based on which kind of query IR-nodes it matches. The scores of data IR-nodes matching those primary query IR-nodes are calculated in the same way as in the selection operator using the scoring function. The score of the data IR-nodes (e.g., the root <article> element) matching secondary query IR-nodes is obtained by selecting the highest score it can possibly achieve using the scoring function. For example, using the scoring function `$1.score = $4.score`, the root <article> node (`$1`) will have the same score as the highest scored data IR-node matching `$4`, which happens to be <article> itself in this case. This score changes dynamically when the set of `$4`-matching data IR-nodes is changed, for example, due to the pruning by Pick operator, discussed in Sec. 3.3.2.

3.2.3 *Scored Join*

The product operator ($\mathcal{C}_1 \times \mathcal{C}_2$) takes two collections of data trees as inputs and produces an output collection of scored trees. Each data tree in the output has a root with

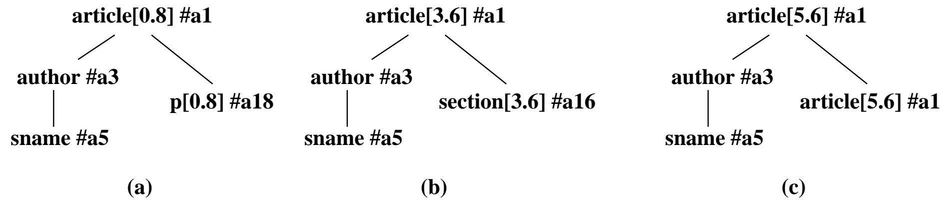


Figure 5: Three Representative Result Trees of Query 2 with Selection. This figure shows three of the results obtained by applying query 2 to the example database in Figure 1. The score of the IR-nodes are calculated using functions defined in Figure 9 and are indicated in the square bracket.

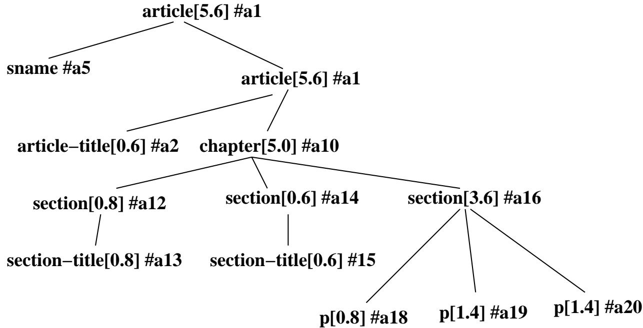


Figure 6: Result Tree of Query 2 with Projection. This figure shows the result tree obtained by applying query 2 to the example database with $PL = \{\$1, \$3, \$4\}$. The node <author> is not retained in the result since it is not in PL. Each data node matching the query IR-node \$4 are retained and assigned a score independent of others.

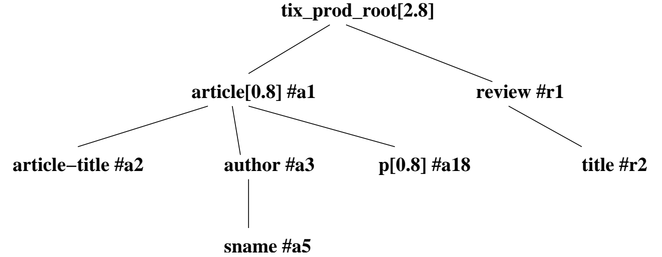


Figure 7: One Result Tree of Query 3. This figure shows one of the many result trees obtained by applying query 3 to the example database. The score of the root node <tix_prod_root> is the sum of the similarity score between `#a2` and `#r2` and the score of `#a18` (See ScoreBar function in Figure 9).

`tag = tix_prod_root` and the two children of the root correspond to the root nodes of every pair of trees from the two input collections. A scored join operator ($\mathcal{C}_1 \bowtie' \mathcal{C}_2$) can therefore be viewed as a selection on the product of two input collections. In the scored pattern tree applied to the product, the predicate conditions involving nodes in both input collections are called join conditions and can be assigned a scoring function just like an atomic query IR-node. Figure 7 shows one of the results obtained from applying the scored pattern tree of query 3 in Figure 4 to the example XML database.

3.3 New Operators

In this section, we introduce two new operators that manipulate scored data trees in ways that are frequently required in IR-style query processing.

3.3.1 Threshold

A Threshold operator $\tau'_{\mathcal{P},TC}(\mathcal{C})$ takes a collection of scored data trees as input, a scored pattern tree $\mathcal{P}$ and a threshold condition TC as parameters, and returns a collection of scored trees. TC is a set of conditions (either a real number value V or an integer K) on one or more query IR-nodes in $\mathcal{P}$. The output comprises exactly those scored data trees in the input that satisfy:

- for each query IR-node in $\mathcal{P}$ and a corresponding V in TC, and each scored data tree in the result, at least one data IR-node matching the query IR-node in the result data tree has a score higher than V.
- for each query IR-node in $\mathcal{P}$ and a corresponding K in TC, and each scored data tree in the result, at least one data IR-node matching the query IR-node in the result data tree has a rank higher than K, where the rank is obtained by sorting the data IR-nodes (among all the input data trees) based on the score.

The Threshold operator is very similar to the selection operator, in fact, thresholding based on V can be directly expressed in TIX as a selection on the score attribute of the data IR-node. Expressing K-based thresholding, however, is more complex. It requires a grouping on the data IR-nodes using an empty grouping basis with the ordering function based on the score. A projection is then applied to retain the leftmost K subtrees, which correspond to the top-K results. The introduction of the Threshold operator simplifies the expression of irrelevance filtering, which is necessary for an algebra targeting information retrieval, such as TIX.

3.3.2 Pick

The Pick operator $\rho'_{\mathcal{P},PC,AD}(\mathcal{C})$ is the key operator that removes the redundancy in the returned results for an IR-style query. It takes a collection of scored data trees as input, a scored pattern tree and a pick-criterion PC as parameters, and returns a collection of scored trees. PC is a set of conditions on one or more query IR-nodes such that

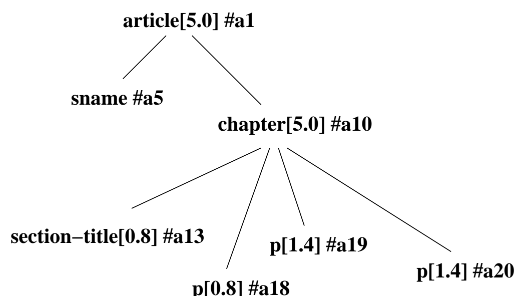


Figure 8: Result of Query 2 with Projection Followed by Pick. The user defined PC, PickFoo, is shown in Figure 9.

any data IR-node matching a query IR-node mentioned in PC must satisfy in order to be returned.

Pick operates on the set of data IR-nodes that correspond to the same query IR-node in the pattern tree and selects the nodes that satisfy the PC. Though easily mistaken as a special projection operator, Pick is quite different from projection in that projection only needs information local to the node being projected (e.g., the `tag name`), while Pick needs information that may reside elsewhere in the data tree (e.g., the ancestor nodes). Figure 8 shows the result of applying a Pick operator to the projection result in Figure 6, where the PC condition (see function `PickFoo` in Figure 9) specifies that: 1) any data IR-node with a score at least 0.8 is considered relevant; 2) for any data IR-node (starting with the one highest in the tree hierarchy), if more than 50% of its child nodes are relevant; 3) and its direct parent node is not picked or it has no parent node, then the data IR-node is picked. The last condition implies that between a parent node and a child node, only one of them will be returned, thus we call this parent/child redundancy elimination. A Pick operator is usually applied after a projection operator to further eliminate nodes that are redundant in the output (see Sec. 5.3 for details).

EXAMPLE 3.1. *Having defined all the necessary operators, it is now possible for us to see how an IR-style query such as example query 2 can return an appropriate set of results back to the user though a combination of these operators. Given the example database in Figure 1 and the scored pattern tree in Figure 3, the top result (*`#a10`*) can be obtained through the following 4 steps. First, a projection operator is applied to generate a single scored data tree as shown in Figure 6. Second, a* Pick *operator is applied on the result and generates the data tree as shown in Figure 8. Third, a selection operator (with appropriate modifications in the pattern tree) takes the data tree and produces a collection of five trees, corresponding to the five primary data IR-nodes. Finally, a* Threshold *operator selects the highest scored result, which contains the desired <chapter>(*`#a10`*) node. The subtree rooted at* `#a10` *can then be retrieved from the database and returned to the user.* □

4. EXTENSION OF XQUERY

We now illustrate how IR features can be incorporated into XQuery, the *de facto* XML query language. Because most of the ideas are discussed in the previous two sections, here we simply use the extended XQuery language to express the three previous example queries in Figure 10 and discuss some issues that are specific to the extension. The various user defined functions are shown in Figure 9.

```
define function ScoreFoo(element* $a, term* A, term* B)
   return void
{
   For $x in $a
      For each term α in A
         i += count(α, $a/alltext()) × 0.8
      For each term β in B
         i += count(β, $a/alltext()) × 0.6
      $x/@score = i
}

define function ScoreSim(element $a, element $b)
   return decimal
{
   return count-same($a/text(), $b/text())
}

define function ScoreBar(score1, score2)
   return decimal
{
   if score2 ≥ 0.0 return score1+score2
   else return 0.0
}

define function PickFoo†(element $a)
   return bool
{
   x = count($a/*[@score≥0.8])
   if (($a.parent == NULL OR $a.parent is not Picked)
         AND (x / count($a/*) ≥ 50%) )
      return TRUE
   else return FALSE
}
```

Figure 9: Example User Functions. alltext() function returns the entire textual content of the subtree root at the node. In the examples throughout the paper, a simple count of term occurrences is used in ScoreFoo, in reality, it should use more sophisticated methods involving term frequency and inverted document frequency. † See Figure 12 for an algorithm implementing a generalized version of this function.

Query 2 illustrates how a basic structured IR-style query can be expressed in XQuery. In particular, the Score clause assigns a score to each node bound to `$a` and the Pick clause eliminates redundant nodes based on the pick criterion. The Threshold clause restricts the output to contain nodes with sufficiently high score or rank. In addition, the expression `descendant-or-self::*` is used to represent the `ad*` relationship in the pattern tree. Query 3 shows how a more complex IR-style query involving a similarity join can be expressed in XQuery. It first generates a product of the two input documents with the join condition being scored. The result are then scored based on the query phrases and pruned using the pick criterion.

5. ACCESS METHODS

We have identified in the previous sections a number of requirements for effective information retrieval from XML databases. In this section, we study the effective evaluation of operators with IR-style functionality within the context of a set-oriented, pipelined, database-style query evaluation engine. The primary issue is how to manipulate scores – how

```
XQuery For Query 1 structured IR-style query
For   $a in document(``articles.xml'')//article/
            descendant-or-self::*
Score $a using ScoreFoo($a, {``search engine''},
               {``internet'', ``information retrieval''})
Pick  $a using PickFoo($a))
Return
   <result>
      <score>$a/@score</score>
      { $a }
   </result>
Sortby(score)
Threshold $a/@score ≥ 4 stop after 5
```

```
XQuery For Query 2 structured IR-style query
For   $a := document(``articles.xml'')//
            article[/author/sname/text()=``Doe'']/
            descendant-or-self::*
Score $a using ScoreFoo($a, {``search engine''},
               {``internet'', ``information retrieval''})
Pick  $a using PickFoo($a))
Return
   <result>
      <score>$a/@score</score>
      { $a }
   </result>
Sortby(score)
Threshold $a/@score ≥ 4 stop after 5
```

```
XQuery For Query 3 IR-style join query
Let   $c :=
      (<root>
         For   $a in document(``articles.xml'')//
                article[/author/sname/text()=``Doe'']
         For   $b in document(``reviews.xml'')//review
         For   $at in $a/article-title
         For   $bt in $b/title
         Return
            <tix_prod_root>
               <simScore>
                  ScoreSim($at/text(),$bt/text())
               </simScore>
                  { $a }
                  { $b }
            </tix_prod_root>
         Threshold simScore ≥ 1
      <root>)
For   $d := $c//tix_prod_root/article/descendant-or-self::*
Score $d using ScoreFoo($d, {``search engine''},
               {``internet'', ``information retrieval''})
Pick  $d using PickFoo($d)
For   $e := $c//tix_prod_root[//$d]
Score $e using ScoreBar(decimal($d/@score),
                        decimal($e/simScore))
Return
   <tix_prod_root>
      <score>$e/@score</score>
      { $d }
      { $e/review }
   </tix_prod_root>
Sortby(score)
```

Figure 10: XQuery Expression of IR-style Queries.

to generate them, how to propagate and consolidate them. In Sec. 5.1 we present new access methods that efficiently generate scores from scratch. In Sec. 5.2, we describe how standard access methods can be changed slightly to use and modify scores. Finally, in Sec. 5.3, we consider the implementation of the two new score-utilizing operators Threshold and Pick.

5.1 Score-Generating Methods

Scores are generated early in a query evaluation plan, upon initial data or index access. An index look-up for an individual indexed term would at the very least return identifiers of XML elements in which this term occurs. However, one can easily return more, such as the number of occurrences of the term in the XML element, possibly normalized in some way. Such auxiliary information can readily be used to obtain a meaningful score, such as the popular *tf*idf* measure. Indeed, scores in typical information retrieval systems are generated using just this sort of a technique.

Most queries may specify more than one term for IR relevance scoring. The simple score generation procedure described above can be composed with standard TIX operators, implemented as described in Sec. 5.2. However, there are significant opportunities to do better, by developing new access methods for operator combinations that are likely to occur frequently. We have identified two such access methods, TermJoin and PhraseFinder, which we describe next.

5.1.1 TermJoin

The most common IR-style predicate is term matching, where a node is scored based on how many terms in the query it has in the textual content of itself and its descendant nodes. The straightforward implementation is to use an inverted index to obtain a set of term-containing text nodes for each term in the query, find all their ancestor nodes, group nodes within each set according to node id, then union these sets. Initial scores are generated upon individual term index look-up; these are propagated and modified in the subsequent operators. We can express this operator sequence in TIX as follows:

$$\sigma'_{\mathcal{P}}(\mathcal{C}) = \bigcup_{1}^{n}(\gamma'_i(\sigma'_{\mathcal{P}_i}(\mathcal{C})))$$

where each pattern tree $\mathcal{P}_i$ contains only one term in the original pattern tree $\mathcal{P}$, n is the number of total terms, and γ'_i represents a grouping operation (based on node identifier) to generate a final score for each node.

We propose a novel access method TermJoin (shown in Figure 11) that efficiently implements score generation based on term matching. The algorithm is inspired by the stack-based family of algorithms for structural join described in [2, 6, 9]. The basic notion is to make a single "merge" pass through lists of leaf elements, one list for each term obtained by looking up an index, ordered by (start) position in the database tree. For each element in this "merged list", we first ensure that exactly all ancestors are placed on stack, and then note the corresponding term occurrence for each element in the stack. When an element is popped from stack, we would have accumulated information regarding term occurrences at itself and all descendant elements, permitting us to assign a score to it (using the ComputeScore() function provided) and output.

Algorithm TermJoin (P, I, s)

```
stack->Push(root)
For each term t in P
   get Next occurrence of t from I.
While (at least one node exists in I)
   let t-min = node from I with
            smallest startkey
   While (t-min.startkey >
         stack->Top().endkey)
      popped = stack->Pop()
      top = stack->Top()
      For each term t in P
         top->IncrementCounterBy(t,
            popped->GetCounter(t))
      if (!s)
         top->AppendToBufferAndList(
            popped->GetBufferAndList)
      popped-score = ComputeScore(
         popped->GetAllCounters(),
         popped->GetBufferAndList())
      Output popped and popped-score
   let A[] = ancestors of t-min, cnt = 0
   While (A[cnt] != root ||
         A[cnt] != stack->Top())
      stack->Push(A[cnt++])
   top = stack->Top()
   top->IncrementCounterBy(t-min,1)
   if (!s)
      top->AddToBufferAndList(t-min)
   read I for next t-min
While (stack != EMPTY)
   popped = stack->Pop()
   top = stack->Top()
   For each term t in P
      top->IncrementCounterBy(t,
         popped->GetCounter(t))
   if (!s)
      top->AppendToBufferAndList(
         popped->GetBufferAndList)
   popped-score = ComputeScore(
      popped->GetAllCounters(),
      popped->GetBufferAndList())
   Output popped and popped-score
```

Figure 11: Algorithm TermJoin. Algorithm finds all ancestors that are common among the terms in a query. Terms are read from an inverted index. Each of the ancestors is assigned a score that is computed in a simple or or complex way. Input *P* is a IR-style query phrase consisting of terms. Input *I* is an inverted index that is scanned for each of the terms in P. Parameter *s* is a boolean that decides which scoring mechanism to use, simple or complex.

Complex Scoring Function. Frequently, the score assigned to an internal (non-leaf) node may depend not just on which query terms were present in descendants, but also what proportion of the descendants are relevant. Thus, an article may be assigned a low score if there is only one paragraph buried in it that contains the query terms, even if all the query terms are present, and repeated many times, within this one paragraph. Scoring functions that require this sort of information regarding non-matches are called *complex*, as opposed to *simple* scoring functions we have considered hitherto. Evaluating such functions requires keeping around some additional information for nodes on stack. This additional work is reflected in Figure 11 through the sections of pseudo-code under the condition if(!s).

5.1.2 PhraseFinder

Sometimes, an IR specification may be interested in a specific phrase, such as "information retrieval" in our running example. XML elements that contain the individual terms "information" and "retrieval" are not relevant unless they contain the specific phrase "information retrieval". Typically, it is not feasible to have index information for all such phrases. As such, the standard implementation would involve looking up the index for each term in the phrase separately, performing an intersection of the element identifiers returned, and then verifying that the phrase of interest does indeed occur in the candidate XML elements returned by the intersection step.

IR systems often keep information regarding location in document for each occurrence of an indexed term, and can use this information to advantage. Following this idea, we developed a PhraseFinder access method that uses word offset information in the index to verify phrase occurrence during the intersection phase itself. Counts of phrase occurrences are then used to generate appropriate score values.

5.2 Score-Modifying Methods

Access methods for standard operators can be extended in a straightforward way to manipulate scores. We present two examples here, and move on.

EXAMPLE 5.1. *Consider the value join access method. It takes in two sets of scored witness trees and outputs a set of scored witness trees where each witness tree is the merging of two input witness trees that satisfied the join condition. Formally, we write:*

$$A \bowtie_{c,w_1,w_2} B = \{(x,s) \mid x \in \underline{A} \bowtie_c \underline{B} \ \wedge s = f(w_1, s_A, w_2, s_B)\}$$

c is the join condition. $\underline{A}$ and $\underline{B}$ are the non-scored versions of input sets A and B. s is a score that gets assigned to an output tree x. The value join may also take in two weights, w_1 and w_2, each corresponds to one of the two input sets of witness trees. The function f that calculates s can be as simple as a weighted addition of the two scores, s_A and s_B associated with the two input trees that composed x. A possible IR value join condition is a similarity condition. □

EXAMPLE 5.2. *Consider the set union access method. It takes in two sets of scored witness trees and outputs a set of scored witness trees where each witness tree belongs to at least one of the input sets. Formally, we write:*

$$A \cup_{w_1,w_2} B = \{(x,s) \mid x \in \underline{A} \cup \underline{B} \ \wedge s = f(w_1, s_A, w_2, s_B)\}$$

$\underline{A}$ and $\underline{B}$ are the non-scored versions of input sets A and B. s is a score that gets assigned to an output tree x. The set union may also take in two weights, w_1 and w_2, each corresponds to one of the two input sets of witness trees. The function f that calculates s can be a weighted addition of the two scores, s_A and s_B associated with the input trees that composed x. Note that s_A or s_B can be a zero since we may have the input witness tree be in only one input. Also, another modification to f may be to have it give more weight to x that belongs to both A and B than to an x that belongs to only one of them. □

5.3 Score-Utilizing Methods

Yet another important set of access methods are the ones that utilize the scores. Two such examples are the ones for evaluating Threshold and Pick. As discussed before, those two new TIX operators differ from existing operators in that their evaluation requires information not just local to the current node but also query-wide. For Threshold, the information possibly required (e.g., global ranking) can be efficiently generated from the input itself by employing techniques proposed in [8, 5]. The Pick operator poses much of a challenge. Let's look back to the example *PC* condition introduced in Sec. 3.3.2, where a predefined *relevance score threshold* (0.8) is used to determine relevance of nodes and a *qualification percentage* (50%) is used to decide whether to return the node. While it is reasonable to let the users define the qualification percentage, it is often unrealistic to ask the users for the exact relevance score threshold since they have no idea of the distribution of the scores for a given query. Auxiliary data like histogram of the number of data IR-nodes matching a query IR-node with respect to the score, enables the user to specify such scores more flexibly and allows the evaluation of Pick to be done more efficiently.

Although the *PC* conditions can be arbitrary, most of them will have the following properties: First, a notion of relevance score threshold for data IR-nodes in the input collection. Second, removing redundancy in the input tree either along the ancestor-descendant relationship (vertical, e.g., parent/child redundancy elimination) or along the sibling relationship (horizontal, e.g., returning only the first author of the relevant article). While the horizontal redundancy can be easily dealt with, the vertical redundancy is more difficult to remove as it potentially requires an examination of all nodes to decide about any one. We present a novel stack-based access method in Figure 12, which evaluates both cases efficiently with the help of auxiliary data. As shown in the figure, the algorithm takes an input tree collection, a `DetWorth` function that decides whether the node is worth returning based on the auxiliary data, and a `IsSameClass` function that decides whether two nodes belong to the same return class. The `PickFoo` function in Figure 9 is an instance of this algorithm, where the `DetWorth` function returns true if a node has more than 50% child nodes that are relevant (score ≥ 0.8) and the `IsSameClass` returns true if the two nodes are either both at the odd-numbered level or both at even-numbered level. Applying it to the scored tree in Figure 6, we obtain a redundancy eliminated scored tree shown in Figure 8. In particular, the <article>(`#a1`) data IR-node (not the root node) is dropped because less than 50% of it child nodes are relevant, and the <section>(`#a16`) is dropped because its parent node <chapter>(`#a10`) is determined to be returned. The algorithm presented here is blocking until some node is determined to be not worth returning, in which case, all the nodes in the subtree rooted at it can be outputted. This is necessary because the decision of whether to return a particular node takes into consideration not only the local properties of the node (e.g., number of child/descendant nodes, tag name, etc.) but also information of the nodes outside the subtree rooted at the node (e.g., whether the parent node is returned).

Algorithm Pick (T, DetWorth, IsSameClass)

```
Let L be the list of all leaf nodes in T
   in the order of their startkey
while (L != EMPTY)
   while (L->current is not a descendant of
            AnsStack->Top())
      n = AnsStack->Pop()
      stack->Push(n)
   P = stack->Top().parent
   if (P != NULL and P != L->current.parent)
      AnsStack->Push(P)
      P = L->current.parent
   stack->Push(L->current)
   L->advance
   while (L->current is a child of P)
      stack->Push(L->current)
      L->advance
   if (DetWorth(P) = TRUE)
      stack->Push(P)
   else /* P is not worth return */
      while (stack->Top() is a descendant of P)
         popped = stack->Pop()
         if (IsSameClass(popped,P) = FALSE)
            output popped
while ( (n = AnsStack->Pop()) != NULL)
   stack->Push(n)
P = stack->Pop()
output P
while (stack != EMPTY)
   popped = stack->Pop()
   if (IsSameClass(popped,P) = TRUE)
      output popped
```

Figure 12: Algorithm Pick. Algorithm goes through the input scored data tree *T* and selectively returns the nodes according to the provided *D*etWorth and *I*sSameClass functions. L can be easily generated from T in linear time. After L is empty, all nodes on the stack are potentially worth returning and we arbitrarily decide to output the top node.

6. EXPERIMENTAL EVALUATION

We ran an extensive set of experiments to evaluate the performance of our new access methods. Experiments were run on a 1.8 GHz PC-compatible machine with 256 MB of RAM running WindowsXP. The data set used is from the INEX [11] initiative. It comprises technical articles from IEEE Transactions marked up in XML: 18 million XML elements with a total size of 500 MB. After loading into our database, the total size grew to 5 GB. We ran the experiments using an XML database system [24] that was available to us. Each experiment was run five times. The lowest and highest readings were ignored and the remaining three were averaged.

There are three new access methods to evaluate: Ter-

mJoin, PhraseFinder and Pick. For the last of these, we found no alternative against which to compare. Therefore, we performed a series of experiments to evaluate it. For the lack of space, we merely report here that the algorithm took between 0.01 to 1.03 seconds for the parent/child redundancy elimination pick criterion and with an input size ranging from 200 nodes to 55,000 nodes.

6.1 TermJoin Evaluation

To evaluate the TermJoin algorithm, we compared it to two other techniques: Composite and Meet. We also implemented and evaluated a variant we called Enhanced TermJoin.

Composite: In Sec. 5.1.1, we showed how to express TermJoin in terms of standard operators. This operator expression can be evaluated directly to obtain a baseline implementation that is a composite of standard operators, we refer to this as *Comp1* in the tables. As advised by recent studies, pushing structural joins further down in the evaluation plan gives good performance. Therefore, we also implemented this technique and referred to it as *Comp2* in the tables.

Generalized Meet: Recently, [22] proposed an algorithm, Meet, to find the lowest common ancestor for a given set of elements (with term occurrences). We are interested in all common ancestors, which are easily obtained by traversing up the ancestor chain from the lowest common ancestor. We are also interested in ancestors that contain only some of the query terms but not all (with an appropriately lower score, of course). These are produced as intermediate results in the algorithm, and can be output. With these changes, the algorithm of [22] can be adapted to compute a Term Join. We call the resulting algorithm Generalized Meet. It recursively obtains the ancestors of the text node containing any of the terms and output them along with the term occurrences after grouping based on node id. The term occurrences are then used for calculating the score of the node.

Enhanced TermJoin: This is a slightly modified algorithm from the original TermJoin. It uses an index structure to get a parent of a given node. Along with the parent information, the number of children of this parent is returned. In the original algorithm, a data access to the database is performed and some navigation is needed to get the number of children. Therefore, we expect savings when using this index structure.

We use two different scoring functions: The *simple scoring function* is a weighted sum of the occurrences of each term under a given ancestor. The *Complex* scoring function examines the term distribution among child nodes. It assigns higher scores to nodes where the distances between terms are smaller. A distance between two terms is the offset difference if they are in the same text node or multiples of node-to-node distance if they are in different text nodes. The score is further multiplied by the ratio between the number of non-zero scored children and the number of total children.

6.1.1 Increasing Term Frequencies

We evaluated the TermJoin algorithm using two-term phrases with increasing term frequencies. Table 1 shows run times for the different techniques with varying frequencies of terms in the phrase. For example, the first line in the table shows run times of the different algorithms using a query with two terms each occurring around 20 times in the database. We kept selecting different pairs of terms, one for each two in the table, with increasing term frequency until it reached 10,000. With the simple scoring function, as shown in Table 1, TermJoin outperforms the Generalized Meet algorithm by up to four times. And it typically outperforms Comp1 and Comp2 by 2 to 4 orders of magnitude. We did not run Enhanced TermJoin here because it is not applicable in the context of simple scoring function. In Table 2, we compare TermJoin with Comp1, Comp2, Generalized Meet, and the Enhanced TermJoin using the complex scoring function. As expected, the run time of all algorithms increase compared with using the simple scoring function because we are keeping and maintaining more data with the ancestor in order to calculate its complex score. We can see from Table 2 that TermJoin still outperforms Comp1 and Comp2 by 2 to 4 orders of magnitude and Generalized Meet by up to 8 times. The Enhanced TermJoin performs better than the TermJoin by up to 8 times because the information can be obtained from index directly. In Table 3, we fixed the frequency of the first term at 1,000 and varied the frequency of the second term. We observe similar trends to those in previous tables. It is also interesting to see that Comp1 does not scale as well as the other techniques.

Approx. term freq.	*Comp1*	*Comp2*	*Generalized Meet*	*TermJoin*
20	0.01	283.70	0.01	0.01
100	0.09	414.40	0.03	0.02
200	0.36	468.76	0.05	0.03
300	1.66	523.78	0.17	0.11
500	2.92	536.42	2.01	1.45
1,000	18.37	613.15	7.92	5.77
2,000	42.64	644.60	27.29	12.16
3,000	93.37	655.87	28.52	16.34
5,500	492.98	732.49	30.28	18.01
7,000	955.94	766.07	36.22	19.42
10,000	1641.63	840.53	96.68	20.55

Table 1: Performance (in seconds) of the different techniques using a query with two index terms of different term frequencies. The simple scoring function has been used to calculate score.

6.1.2 Increasing Phrase Size

We chose two terms of roughly the same overall occurrence frequency at around 1,500 and kept adding terms of the same frequency to it. We started with a two-term query and kept adding one term at a time until we reached seven terms. In Table 4, we show the performance of different techniques. This time, the TermJoin performs two times better than Generalized Meet and up to 2 orders of magnitude better than Comp1 and Comp2. The Enhanced TermJoin performs better than the TermJoin by up to 4 times. We used the complex scoring function because it is more accurate than the simple one. The reason for this is that the complex scoring function makes a better use of XML's structure to enhance the quality of the score.

6.2 PhraseFinder Evaluation

In order to evaluate the performance of the PhraseFinder, we again compared it to a sequence of basic access methods supported by the database engine.

Composite of Access Methods: To achieve the same

Approx. term freq.	*Comp1*	*Comp2*	*Gen. Meet*	*Term Join*	*Enhanced TermJoin*
20	0.02	285.56	0.02	0.02	0.04
100	0.10	417.89	0.10	0.06	0.08
200	0.40	474.73	0.29	0.15	0.11
300	1.68	543.28	1.05	0.59	0.21
500	3.08	547.15	4.14	2.37	0.45
1,000	18.96	622.58	14.53	7.65	1.16
2,000	43.75	675.57	56.71	24.67	4.13
3,000	94.33	688.06	83.39	27.94	6.84
5,500	519.82	742.09	319.59	28.32	10.65
7,000	1070.95	781.00	331.79	48.61	15.46
10,000	1717.91	852.35	722.88	81.60	21.93

Table 2: Performance (in seconds) of the different techniques using a query with two index terms of different term frequencies. The complex scoring function has been used to calculate score.

Approx. term2 freq.	*Comp1*	*Comp2*	*Gen. Meet*	*Term Join*	*Enhanced TermJoin*
20	3.72	321.47	3.45	0.93	0.48
200	5.30	576.21	4.29	1.44	0.64
1,000	18.96	622.58	14.53	7.65	1.16
3,000	39.81	655.10	38.85	11.87	3.52
7,000	113.06	735.98	184.99	29.51	11.78

Table 3: Performance (in seconds) of the different techniques using a query with two index terms. Frequency of term1 is fixed at 1,000. The complex scoring function has been used to calculate score.

result as in the PhraseFinder, we perform an index access for each term in the phrase to get its occurrences in the database. Then, we intersect these occurrences to obtain a list of text nodes with at least one occurrence of each term. Each text node subsequently goes through a filter to make sure that the offsets of the terms in the phrase are exactly 1 apart and that they are in the same order in the text node as they are in the phrase. We refer to this as Comp3.

Table 5 shows the performances of Comp3 and PhraseFinder using thirteen different two-term phrases. The PhraseFinder performs up to 9 times better than Comp3. The reason for this is the extra work done at the filter level in the Access Methods to check offsets especially if the result of the intersection is big.

7. RELATED WORK

A number of studies focusing on supporting ranked or preference queries have been done in the relational context [1, 16, 8]. In the framework proposed in [1], a preference function is a mapping from a record (tuple) of a given record type (relation) to a score based on user preferences and a meta-combine function is a way of combining such functions to compute a new score based on those original scores. The scoring functions for the primary IR-nodes and secondary IR-nodes in our system resemble those two functions respectively. The PREFER [16] system utilizes materialized views of precomputed preference queries to compute the current query more efficiently. The preference function, however, is limited to the linear format, where each attribute of the record is assigned a weight and the total score is the sum of the scores of some weighted attributes. The *MPro*

# of terms in query	*Comp1*	*Comp2*	*Gen. Meet*	*Term Join*	*Enhanced TermJoin*
2	20.49	638.69	22.39	8.06	2.08
3	41.91	801.82	40.99	14.13	3.88
4	53.53	1072.16	44.35	16.09	6.56
5	71.56	1342.76	58.32	23.84	9.86
6	225.60	1625.05	79.48	34.59	13.69
7	329.70	1892.78	97.58	45.44	16.60

Table 4: Performance (in seconds) of the different techniques using queries with different number of terms. Frequencies of each term is around 1,500.

Query	*Term1 freq.*	*Term2 freq.*	*Result size*	*Comp3*	*Phrase-Finder*
1	121,076	44,930	27,991	10.15	1.33
2	121,076	79,677	462	3.04	1.06
3	107,269	146,477	1,219	5.98	2.04
4	107,269	79,677	1,212	6.36	1.49
5	98,405	146,477	877	4.30	1.98
6	121,076	146,477	1,189	5.84	2.15
7	90,482	68,801	116	5.10	1.30
8	121,076	45,988	34	3.22	1.34
9	121,076	107,269	320	4.56	1.82
10	98,405	28,044	455	3.82	1.02
11	146,477	68,801	1,372	8.75	1.74
12	121,076	68,801	249	4.12	1.52
13	98,405	107,269	17	5.84	1.65

Table 5: Performance (in seconds) of the PhraseFinder and Composite of Access Methods using 13 different phrases each is 2 terms long.

algorithm [8] by Chang and Hwang takes a different approach and minimizes the number of expensive user-defined predicates being evaluated. It requires the preference function to be "monotonic" and assumes that a ceiling value of the function can be easily derived. Sideway Value Algebra (SVA) [19], on the other hand, focuses on the algebraic representation of those preference scores without worrying about how they are generated. It keeps the preference scores as sideway values and propagate them during the evaluation of the algebraic operators. Our study distinguishes itself from those previous studies by focusing on ranked queries on the XML data model, where the queries are structured and inherently more complex. Like SVA, we provide a theoretical framework for ranked queries in the form of extension to an existing algebra. Unlike SVA, where the scores are assumed to be stored in the database, we provide means of generating them efficiently from a database with no prior preference scores using the user specified functions.

XXL [23] and XIRQL [13] are two query languages supporting ranked queries on XML data. XXL incorporates special operators, like *similar*, into the query language and uses ontological information to automatically calculate the scores. Our system, on the other hand, enables the user to specify scoring function by providing them with language extensions with which user-defined functions can be plugged. XIRQL is the first to address the result duplication problem in the IR-style structured query where element type is not specified. They choose to return only those nodes of predetermined types. We decide not to impose such a limitation and present both a default way and a user-defined way of

picking elements to be returned through a stack-based algorithm. The *meet* operator proposed in [22] finds the lowest common ancestor in the XML tree for all phrases given in the query. It recursively retrieves ancestor nodes containing individual phrases until it finds one node that contains all the phrases. Although useful in finding the nearest concept of multiple phrases, *meet* is not adequate in the context of IR queries since it lacks the support for relevance ranking and does not tolerate missing phrases.

Cohen [10] proposed a logic system called WHIRL where data from heterogeneous databases can be compared and integrated using a pure natural language based approach without the assumption of a common domain. At the core of WHIRL is the ability to determine the degree of similarity between two "name constants", which can potentially be adopted by the scoring function for the join condition in our system. Our work can also be applied to the field of probabilistic data storage and querying [18, 14], where the probability can be viewed as the equivalence of the score and be manipulated similarly.

8. CONCLUSIONS

In this paper we have devised a bulk algebra, TIX, that permits the integration of information-retrieval style query processing into a traditional pipelined query evaluator for an XML database. The major advances in TIX include (i) the ability to manage relevance scores, including score generation, manipulation, and use; and (ii) facilities for management of result granularity (necessitated because relevance may be associated with nested elements at multiple granularities). The algorithms TermJoin and PhraseFinder effectively implement the score generation by using a stack-based approach. Experiments show that they typically improve the performance by 2 times to 4 orders of magnitude. Similarly, the algorithm Pick uses a stack-based strategy to attack the result granularity problem through user-specified redundancy elimination. With the help of auxiliary data, it is able to efficiently determine the most appropriate irredundant set of results to return from among a larger set of relevant but redundant elements found in the XML database.

A tantalizing vision with XML is that of (partially) structured documents being managed and queried within a database. This paper describes significant steps toward making that vision a reality.

9. REFERENCES

[1] R. Agrawal and E. L. Wimmers. A framework for expressing and combining preferences. In *SIGMOD*, 2000.

[2] S. Al-Khalifa, H. V. Jagadish, N. Kouda, J. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In *ICDE*, 2001.

[3] D. Beech, A. Malhotra, and M. Rys. A formal data model and algebra for XML. W3C XML Query Working Group Note, September 1999.

[4] C. Beeri and Y. Tzaban. SAL: An algebra for semi-structured data and XML. In *ACM SIGMOD Workshop on the Web and Databases*, pages 37–42, Philadelphia, PA, June 1999.

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

[6] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal XML pattern matching. In *SIGMOD*, 2002.

[7] D. D. Chamberlin, J. Clark, D. Florescu, J. Robie, J. Simon, and M. Stefanescu. XQuery 1.0: An XML query language. W3C working draft, June 2001. http://www.w3.org/TR/xquery/.

[8] K. C.-C. Chang and S. won Hwang. Minimal probing: Supporting expensive predicates for top-k queries. In *SIGMOD*, 2002.

[9] S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient structural joins on indexed XML documents. In *VLDB*, 2002.

[10] W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *SIGMOD*, 1998.

[11] DELOS. Initiative for the evaluation of XML retrieval. http://qmir.dcs.qmw.ac.uk/inex/.

[12] P. Fankhauser, M. Fernandez, A. Malhotra, M. Rys, J. Simeon, and P. Wadler. The XML query algebra. W3C Working Draft, Feburary 2001.

[13] N. Fuhr and K. Großjohann. XIRQL: A query language for information retrieval in XML documents. In *International Conference on Information Retrieval (SIGIR)*, 2001.

[14] N. Fuhr and T. Rölleke. A probabilistic relational algebra for the integration of information retrieval and database system. *ACM Transactions on Information Systems (TOIS)*, 15(1), January 1997.

[15] C. M. Hoffmann and M. J. O'Donnell. Pattern-matching in trees. *JACM*, 29:68–95, 1982.

[16] V. Hristidis, N. Koudas, and Y. Papakonstantinou. PREFER: A system for the efficient execution of multiparametric ranked queries. In *SIGMOD*, 2001.

[17] H. V. Jagadish, L. V. S. Lakshmanan, D. Srivastava, and K. Thompson. TAX: A tree algebra for XML. In *International Workshop on Database Programming Languages (DBPL)*, Marino, Italy, September 2001.

[18] A. Nierman and H. V. Jagadish. ProTDB: Probabilistic data in XML. In *VLDB*, 2002.

[19] G. Ozsoyoglu, A. Al-Hamdani, I. S. Altingovde, S. A. Ozel, O. Ulusoy, and Z. M. Ozsoyoglu. Sideway value algebra for object-relational databases. In *VLDB*, 2002.

[20] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, New York, 1983.

[21] T. Schlieder and H. Meuss. Result ranking for structured queries against XML documents. In *DELOS Workshop on Information Seeking, Searching and Querying in Digital Libraries*, 2000.

[22] A. Schmidt, M. Kersten, and M. Windhouwer. Querying XML documents made easy: Nearest concept queries. In *ICDE*, 2001.

[23] A. Theobald and G. Weikum. The index-based XXL search engine for querying XML data with relevance ranking. In *EDBT*, 2002.

[24] U. of Michigan. The Timber system. http://www.eecs.umich.edu/db/timber/.

[25] C. Zhang, J. Naughton, D. Dewitt, Q. Luo, and G. Lohman. On supporting containment queries in relational database management systems. In *SIGMOD*, 2001.