An expressive and efficient language for XML information retrieval

Taurai Tapiwa Chinenyanga Nicholas Kushmerick
Smart Media Institute, Computer Science Department
University College Dublin, Dublin 4, Ireland
{taurai.chinenyanga, nick}@ucd.ie

To appear: Journal of the American Society for Information Science and Technology, Special Issue on XML and Information Retrieval, 2001. (Draft of June 5, 2001).

Abstract

XML has emerged as a standard for the exchange of structured data and documents. The database community has proposed several languages for querying and transforming XML, including XML-QL [DFF+99a], Quilt [CRF00] and XQL [Rob99]. However, these languages do not support ranked queries based on textual similarity, in the spirit of traditional information retrieval. Several attempts to extend these XML query languages to support keyword search have been made, but the resulting languages cannot express information-retrieval-style queries such as *"find books and CDs with similar titles"*. In some of these languages keywords are used merely as boolean filters without support for true ranked retrieval; others permit similarity calculations only between a data value and a constant, and thus cannot express the above query. WHIRL [Coh98, Coh00] avoids both problems, but assumes relational data.

We propose ELIXIR, an **e**xpressive and **e**fficient **l**anguage for **X**ML **i**nformation **r**etrieval that extends XML-QL with a textual similarity operator. This operator can be used for similarity joins, so ELIXIR is sufficiently expressive to handle the sample query above. ELIXIR thus qualifies as a general-purpose XML information retrieval query language. Our central contribution is an efficient algorithm for answering ELIXIR queries. The algorithm rewrites the original ELIXIR query into a series of XML-QL queries that generate intermediate relational data, and uses WHIRL to efficiently evaluate the similarity operators on this intermediate data, yielding an XML document with nodes ranked by similarity. Our experiments demonstrate that our prototype scales well with the size of the query and the XML data.

1 Introduction

XML has emerged as an important standard for the interchange of structured documents and data. Two distinct application communities have arisen (see [ABD00] for a detailed discussion). On the one hand, the database community treats XML as a convenient standard for data exchange. On the other hand, the document management community sees XML, like SGML and HTML, as a language for annotating documents containing natural text.

The World Wide Web Consortium[1] (W3C) has announced a draft standard XML query language, XQuery [CFR$^+$01], distilled from several contenders, including XML-QL [DFF$^+$99a, DFF$^+$99b], Quilt [CRF00], XQL [Rob99], and Lorel [LPVV99]. These languages provide a declarative way to access and filter elements from XML databases based on various predicates, and rewrite the original XML data into an arbitrary new structure. These languages have been developed primarily by members of the database community. As such, they generally support (the semistructured equivalent of) the full relational algebra, making them useful for a wide variety of data interchange and transformation applications. There have been several commercial implementations of these languages, including Software AG's Tamino [Sof], Infonyte [Glo] and XYZFind [XYZ].

However, these languages do not fully meet the needs of the document management and IR communities. We focus on one specific deficit: the lack of support for ranking or weighting of query results based on textual similarity or other relevance metrics. For example, consider the XML database containing book and CD titles in Figure 1(a). A typical query, in the spirit of traditional IR, would be to find books or CDs with titles similar to the phrase "Ukrainian cookery". Figure 1(b) shows such a query Q_A in ELIXIR, our extension to XML-QL that supports ranked retrieval based on textual similarity. The result of such a query should be a sequence of the items ordered by similarity to the target phrase according to some textual similarity metric, rather than (for example) an unordered subset of the items that contain both terms. Furthermore, presumably the application that posed this query is interested only in the most-similar titles. The ELIXIR query processor formalizes this notion by returning the best r answers, where r is an evaluation-time parameter. In this example, if textual similarity is computed using a standard IR term vector model, and if the query processor is asked for the best $r = 2$ answers, then the first and fourth items in Figure 1(a) would be returned, in that order; see Figure 1(c).

A more complicated query would be to identify books and CDs with similar titles; see query Q_B in Figure 1(d). Note that in this case "$\sim$" expresses a similarity join (it operates on two variables, rather than a variable and a constant). The result of this query should not be merely books and CDs with identical titles, with unequal pairs discarded. Rather, the result should be an ordered sequence of items, where earlier items are more similar than later items. For example, Figure 1(e) shows the $r = 2$ best answers to this query. The meaning of this XML output is that, across all book/CD pairs, "`Traditional Ukrainian cookery`" is the book that is most similar to some CD, and "`Being and nothingness`" is the book that is second-most similar to some CD. Note that, for the sake of simplicity, this query does not actually output the CDs to which these books are similar ("`Traditional Ukrainian folk music`" and "`Being there`", respectively).

[1] http://www.w3.org.

```
(a)  <items><book>Traditional Ukrainian cookery</>
            <book>Being and nothingness</>
            <book>Shooting Elvis</>                          } "db.xml"
            <cd>Traditional Ukrainian folk music</>
            <cd>Being there</>
            <cd>Milk cow blues</></>

(b)  CONSTRUCT <item>$t</>
     WHERE     <items.(book|cd)>$t</> in "db.xml",           } Q_A
               $t ~ "Ukrainian cookery".

(c)  <item>Traditional Ukrainian cookery</>
     <item>Traditional Ukrainian folk music</>

(d)  CONSTRUCT <item>$b</>
     WHERE     <items.book>$b</> in "db.xml",                } Q_B
               <items.cd>$c</> in "db.xml",
               $b ~ $c.

(e)  <item>Traditional Ukrainian cookery</>
     <item>Being and nothingness</>
```

Figure 1: (a) An XML document describing five books and CDs; (b) an ELIXIR query Q_A for finding items similar to the phrase "Ukrainian cookery" ("$\sim$" is ELIXIR's textual similarity operator); (c) the two best answers to query Q_A; (d) an ELIXIR query Q_B for finding items with similar titles (note that, unlike Q_A, Q_B uses a similaity join); and (e) the two best answers to Q_B.

Several attempts to extend XML query languages to support ranked XML IR have been made, but none of these extensions can express queries such as Q_B, and some cannot express Q_A. Some of these extended languages (*e.g.* [FMK00]) use keywords merely as boolean filters and thus do not support true ranked retrieval. Other extensions permit similarity calculations only between a data value and a constant (*e.g.* [TW00, FG00]). This restriction means that while such languages can express Q_A, they cannot express Q_B, since Q_B requires evaluating a similarity join between two data values. (We discuss related work at length in Section 2.)

To fill this gap, we propose ELIXIR, an **e**xpressive and **e**fficient **l**anguage for **X**ML **i**nformation **r**etrieval that extends XML-QL with a textual similarity operator which can be used just like ordinary XML-QL operators. In particular, both operands can be variables (see line 5 of query Q_B), and therefore ELIXIR queries can express similarity joins, a capability missing from previous languages. ELIXIR is therefore sufficiently expressive to qualify as a general purpose XML information retrieval language.

Of course, the additional expressiveness permitted by ELIXIR's similarity joins has important computational ramifications. The fundamental problem is that if one wants to compute a similarity join between two variables, then it is infeasible to simply generate the full cross product of the bindings of the variables, and then compute the similarity of every pair, as shown in Figure 2.

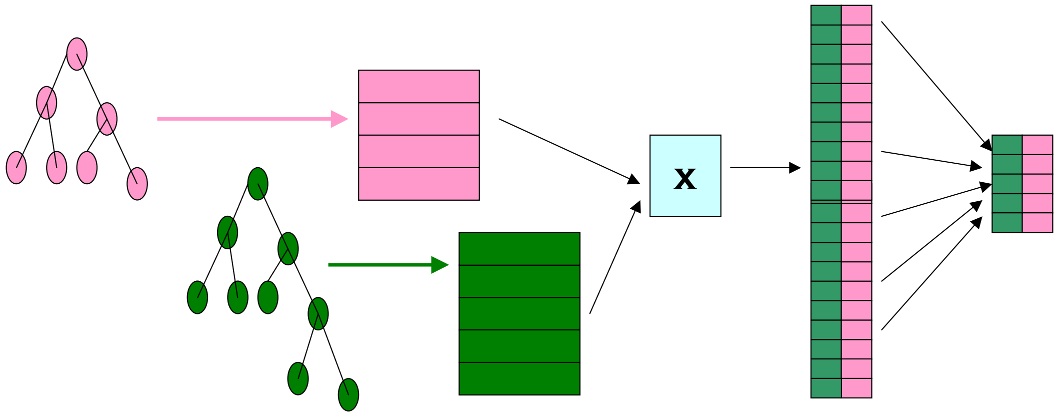

Figure 2: A naive evaluation of a similarity join generates the full cross product of variable bindings.

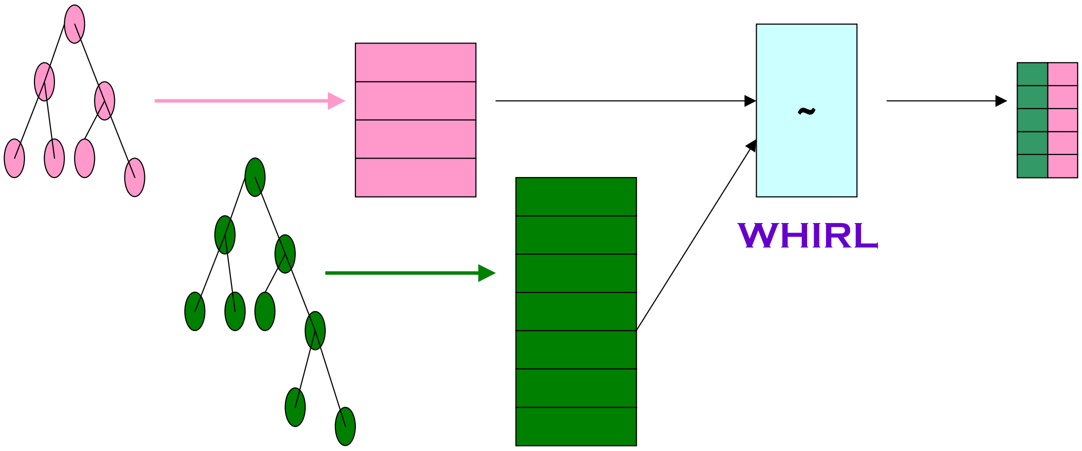

Figure 3: ELIXIR invokes WHIRL to avoid computing the full cross product of variable bindings when evaluating similarity joins.

We have designed an efficient algorithm for answering ELIXIR queries that avoids unnecessarily computing such variable binding cross products; see Figure 3. Specifically, the ELIXIR query processing algorithm invokes WHIRL [Coh98, Coh00] as a "subroutine". As described in Section 2.2, WHIRL extends Datalog (*e.g.* [UW97]) with a textual similarity metric. Our query processor rewrites an ELIXIR query Q into a series of XML-QL queries that generate intermediate relational data, and then invokes WHIRL to efficiently evaluate Q's similarity predicates on this intermediate data. Finally, WHIRL's relational output is translated into the XML structure specified by Q using a final XML-QL query.

Note that the algorithm does *not* simply map the original XML data into relations (*e.g.* [DFS99, STH$^+$99, FTS00]) (which can be expensive in general) and the ELIXIR query into a WHIRL query. Rather, the algorithm uses XML-QL to query the XML data in its native form as much as possible. The resulting intermediate data is then efficiently and losslessly converted into relations for processing by WHIRL. Once the similarity predicates have been evaluated, the resulting tuples are converted back into an arbitrary XML format as specified by the original ELIXIR query.

		data model	
		relational	XML
full support for	no	SQL, Datalog	Quilt, XML-QL, *etc.*
information retrieval?	yes	WHIRL	**ELIXIR**

Figure 4: ELIXIR fills an open gap in the space of query languages that support ranked IR.

The additional expressiveness permitted by ELIXIR's similarity joins is not merely of theoretical interest. For example, data integration (*e.g.* [Wie92, KLD+98, LKMC98]) is an important application that requires similarity joins. Data integration systems present a uniform view over data retrieved from multiple heterogeneous sources. Many data integration applications manipulate "noisy" textual identifiers, such as person or company names, book or movie titles, addresses, and so forth. Given the decentralized nature of such applications, it is unlikely that data integration systems can assume shared domains, or accurate normalization routines, for such identifiers. For example, one might want to compute a join on the "title" attribute of two book databases. But do "Data structures and algorithms in Java" and "Data Structures & Algorithms in Java" refer to the same book? Are "Michael T. Goodrich" and "Goodrich, M. (Johns Hopkins University)" the same person? Cohen [Coh98, Coh00] describes several experiments demonstrating that WHIRL's textual similarity predicates can be highly effective at integrating heterogeneous textual data sources, without relying either on common domains or hand-crafted normalization routines.

To summarize, our central contribution is a query language for ranked IR from XML data that is both:

- more expressive than current languages (arbitrary similarity joins are permitted), and
- efficient (our query processor does not compute unnecessary cross-products of the bindings of variables involved in similarity predicates).

In the remainder of this paper, we describe related work (Section 2), define our ELIXIR language (Section 3), describe our algorithm for answering ELIXIR queries (Section 4), describe an experimental evaluation of our prototype (Section 5), and finally we discuss open issues and summarize our conclusions (Section 6).

2 Related work

As shown in Figure 4, ELIXIR fills an open gap in the space of expressive query languages for XML. In this section, we discuss related work, in order to both situate ELIXIR relative to other research, and to introduce two particular systems—XML-QL and WHIRL—on top of which our ELIXIR query processor is implemented.

2.1 XML query languages

Several languages for querying and transforming XML data have been proposed, including Quilt [CRF00], XQL [Rob99], XML-QL [DFF+99a, DFF+99b] and XQuery [CFR+01], the W3C draft standard XML query language. These languages allow a query to extract particular fragments from one or more XML sources, discard fragments that fail to satisfy certain predicates, and rewrite the original data into any desired XML format.

Our ELIXIR query language is based on the XML-QL query language. This decision was made largely on pragmatic grounds, and we anticipate few theoretical difficulties in applying our ideas to other query languages with similar expressiveness, including XQuery.

In this section we briefly describe XML-QL, focusing on the features that will be needed to understand our ELIXIR query language and processor. This description is somewhat simplified and tailored to the needs of this paper; see [DFF+99a, DFF+99b] for further examples and a formal characterization of the language.

An XML-QL query is an expression of the form

```
CONSTRUCT C
WHERE W
```

The `CONSTRUCT` clause specifies the desired XML format of the output.[2] We might want to output a list of book details, with years and authors encoded as attributes, and titles encoded as contents, of `<book>` tags:

```
CONSTRUCT <book year=$y author=$a>$title</>
WHERE     W
```

Tokens preceded by "`$`" are variables; the pattern above contains three variables: `$y`, `$a` and `$title`. C = "`<book year=$y author=$a>$title</>`" is a template for converting bindings of these variables into an XML fragment. The query processor generates one instantiation of C for each set of variable bindings.

These variable bindings are generated by the `WHERE` clause W. W contains *pattern* statements of the form "*pattern* `in` *source*". "*pattern*" is a pattern indicating how to traverse an XML object. "*source*" is a reference to an XML object (*e.g.*, its URL). Variables in a pattern are bound according to all possible ways to match the pattern against the XML data. For example, we might simply transform some existing data into our desired format:

```
CONSTRUCT <book year=$y author=$a>$title</>
WHERE     <books><item><year>$y</>
                      <author>$a</>
                      <title>$t</></></> in "bib.xml".
```

[2]Well-formed XML documents must contain a single root tag, and thus to be precise an XML-QL query is of the form "`<root>{CONSTRUCT` C `WHERE` W`}</>`". However, in this article we ignore such root tags for the sake of simplicity.

This query would transform the XML document

```
<books><item><year>1957</><author>Stechishin</>
            <title>Traditional Ukrainian cookery</></></>
       <item><year>1956</><author>Sartre</>
            <title>Being and nothingness</></></></>
```

into

```
<book year=1957 author=Stechishin>Traditional Ukrainian cookery</>
<book year=1956 author=Sartre>Being and nothingness</>
```

In addition to pattern statements of the form "*pattern* `in` *source*", the `WHERE` clause can also include boolean operators. For example, the following XML-QL query retrieves books published after 1950:

```
CONSTRUCT <book year=$y author=$a>$title</>
WHERE     <catalog><book><year>$y</>
                         <author>$a</>
                         <title>$t</></></> in "bib.xml",
          $y > 1950.
```

XML-QL supports a variety of typical operators, such as "`>`" "`<`", "`=`", "`!=`" and so forth.

To simplify pattern statements, XML-QL permits *regular path expressions*. For example, a pattern such as "`<catalog.book.year>$y</>`" traverses several tags at once, and "`<catalog.(book|cd).year>$y</>`" illustrates a disjunctive pattern.

Finally, XML-QL queries can be nested, with an entire XML-QL query embedded inside the `CONSTRUCT` C clause. In this paper, we ignore nested queries: our ELIXIR query language is an extension to the non-nested subset of XML-QL.

2.2 WHIRL

While our ELIXIR language borrows heavily from XML-QL in terms of XML data querying and transformation, the language is also closely related to the WHIRL relational database system. The essential aspect of WHIRL is that it efficiently supports ranked IR based on textual similarity. As with our discussion of XML-QL, this description of WHIRL is highly simplified; see [Coh98, Coh00] for details.

WHIRL extends Datalog (*e.g.* [UW97]), with a primitive textual similarity operator "$\sim$". Datalog is a Prolog-like database query language. A Datalog query is a Horn clause, where the head of the clause defines an "output" relation that stores variable bindings that constitute the answer to the query, and the body of the clause is a conjunction of relational predicates. For example, if we have a `book` relation with year, author and title columns, then we could use the Datalog query below to retrieve books published after 1950:

```
output($y,$a,$t) :- book($y,$a,$t), $y>1950.
```

To illustrate WHIRL's ranked retrieval capabilities, the following WHIRL query yields similar results to the previous query, but with the tuples ordered by textual similarity of the title to the phrase "traditional Ukrainian":

```
output($y,$a,$t) :- book($y,$a,$t), $y>1950, $t ~ "traditional Ukrainian".
```

Crucially, WHIRL allows for similarity joins, in which the "$\sim$" operator is applied to two variables, rather than a variable and a constant alone. For example, the following WHIRL query finds books whose titles happen to be similar to their authors:

```
output($y,$a,$t) :- book($y,$a,$t), $t~$a.
```

Of course, WHIRL handles only relational data. Our ELIXIR query processing algorithm in effect extends WHIRL so that it can handle arbitrary XML data.

WHIRL's definition of similarity is based on standard statistical IR term vector techniques (*e.g.* [Sal89]). In this approach, we model textual documents as points in a high-dimensional space. Let V be the *vocabulary*, the set of distinct terms across the documents whose similarities are being computed. Each dimension of this space corresponds to a term $t \in V$. A document d is represented as a point $d \in \Re^{|V|}$. The value of document d in dimension t (written d_t) corresponds to the "strength" or "weight" with which term t is associated with document d. For example, if d does not contain t then presumably $d_t = 0$, while if d contains many occurrences of t then d_t will be relatively large.

Specifically, WHIRL uses the standard IR *term-frequency inverse-document-frequency* (TF-IDF) scheme for weighting terms. In this approach, $d_t = -\log(\mathrm{TF}(d,t)+1) \cdot \log \mathrm{DF}(t)$, where "term frequency" $\mathrm{TF}(d,t)$ is the number of times term t occurs in document d, and "document frequency" $\mathrm{DF}(t)$ is fraction of the documents that contain term t. Given two documents d and d', the *similarity* between d and d' is defined as follows:

$$\mathrm{sim}(d,d') = \sum_{t \in V} \frac{d_t \cdot d'_t}{||d|| \cdot ||d'||} \tag{1}$$

If the document vectors are unit-normalized, similarity values are between 0 and 1, with 0 indicating that the documents share no terms, and 1 indicating that the documents are identical.

With these preliminaries in place, we now briefly describe the semantics of a WHIRL query Q. Let Q' be defined to be Q with all similarity predicates deleted, and let R be the output relation of Q and Q'. Now consider Q's similarity predicates. Each such predicate is of the form "*variable* $\sim$ *variable*" or "*variable* $\sim$ *constant*". Every tuple in R generates bindings for each similarity predicate s_k. The *score* of a tuple from R is defined to be the product of the similarities of the variable bindings in each similarity predicate. The motivation of this definition is that similarity scores are taken to be independent probabilities, so that the probability of their conjunction is the product of their probabilities. The *complete answer* to Q is the relation R with the tuples ordered according to their score. WHIRL does not generate *complete answer*s, but rather *r-answers*: for any integer r the r-answer to Q is the first r answers in Q's complete answer.

The WHIRL query processor efficiently generates r-answers. To do so, it takes advantage of certain properties of the definition of similarity in Equation 1. The basic idea is that, if one wants to find the r documents most similar to some document d, then one need not consider all documents, but only those that share the most terms with d. Furthermore, the most similar documents will likely share terms t with high weight d_t. For example, to find the documents most similar to the phrase "tall octogenarian", it makes sense to process first the few documents containing the rare (and therefore high-weight) term "octogenarian" rather than the many documents containing the common term "tall". The details of the WHIRL algorithm are beyond the scope of this paper; see [Coh98, Coh00] for details.

2.3 Keyword search in XML query languages

There have been several recent attempts to extend various XML query languages with support for keyword search. This work falls into two main categories: techniques for using keywords as boolean filters (rather than for genuine ranked retrieval), and techniques that support relevance ranking but not similarity joins (*i.e.* the similarity operator operates only on a variable and a constant, not two variables).

Florescu *et al* [FMK00] extend XML-QL to support keyword filtering. Their extension adds a `contains` operator to XML-QL that supports keyword search. For example, the predicate `contains($t, "Ukrainian")` discards bindings for `$t` that do not contain this term, in exactly the same way that the predicate `$y>1990` discards bindings for `$y` that do not satisfy the given constraint. Since bindings for `$t` are discarded, not ordered, by `contains`, this extension to XML-QL does not support true relevance ranking/weighting of query results. Other systems that are related to ELIXIR in the same way include Macherius *et al*'s extension to XQL [MHF99], and Naughton *et al*'s XML search engine [NDM00].

While these systems do not rank their results, several research groups have extended other XML query languages to support true ranked retrieval. Like ELIXIR, these languages augment some existing XML query language with a similarity operator. However, these languages appear to be less expressive than ELIXIR: their similarity operators must be applied to a variable and a constant, but not two variables, and thus these languages do not appear to support similarity joins. XXL [TW00] extend an SQL-like language with a textual similarity operator. However, none of the examples in [TW00] suggest how to express similarity joins in XXL and there is no discussion of how similarity joins would be processed efficiently if allowed. We conclude that XXL's similarity operator requires that one argument must be a constant.

XIRQL [FG00] extends XQL with a similarity operator, but again the use of this operator appears to be restricted to a variable and a constant. Finally, Hayashi *et al* define a novel XML query language with support for ranked retrieval [HJG00], but their similarity operator appears to be restricted in the same way. On the other hand, XIRQL and Hayashi *et al*'s languages are more expressive than ELIXIR in the following limited sense. In these two languages, one can attach numeric weights to the query terms. Thus one might query for books that are similar to "Ukrainian cookery", but with the additional constraint that documents that contain “Ukrainian” but not “cookery” should (to some specified degree) be ranked higher than documents that contain “cookery” but not “Ukrainian”. While neither ELIXIR nor WHIRL currently support such keyword weighting, this is an implementation issue only and would present no theoretical difficulties.

2.4 Flattening XML to relations

There has been significant work on techniques for automatically mapping XML data and queries to a relational model (*e.g.*, [DFS99, STH$^+$99, FTS00]). In the case that the XML data happens to have a tabular structure, such a mapping can be performed quickly and accurately. However, in general there may be no simple mapping to a relational model. In complex XML structures, the mapped data may contain (for example) many null values, duplicated data items, or “overflow” relations for handling rare XML structures. Furthermore, there may be many possible such mappings, and so an automatic approach to mapping XML to relations (*e.g.* [DFS99]) must search for a mapping that appropriately trades off complexity for accuracy, but such an optimization process scales poorly with the size and complexity of the original XML data.

We stress that the ELIXIR query processor does *not* simply map the original XML data and query into a relational model. This is the approach taken by XXL [TW00]. As described in Section 3, when processing an ELIXIR query, the XML data itself is never mapped into relations. Rather, tuples of bindings for the query variables are passed to and from WHIRL.

2.5 Alternative text similarity metrics

WHIRL, and therefore ELIXIR, adopts a standard, but very simple, similarity metric based on term overlap. Essentially, similarity grows in proportion to the number of terms shared by two documents. Of course, there are many situations where a variety of real-world knowledge is essential for determining similarity. Furthermore, similarity is a relative concept. In some cases it might be appropriate to treat “Microsoft Research” and “Microsoft Corporation” as the same entity, but for other applications this could be incorrect.

There has been research on more sophisticated similarity metrics, within both the IR and database communities, including substantial work on using lexical resources such as thesauri and conceptual hierarchies to improve retrieval (see [BYRN99] for an overview). Database researchers have investigated a variety of techniques for automatically normalizing and reconciling heterogeneous textual identifiers (*e.g.*, [ME96, HS95]).

Ideally, all of these techniques could complement ELIXIR's simple term-based notion of similarity. However, it remains an open question whether WHIRL, whose efficiency derives from exploiting certain properties of the term-based metric, could be extended to handle these more complicated similarity metrics.

2.6 Structured text databases

One central role for XML is encoding structured documents, and thus ELIXIR is related to research on databases for collections of structured text. Any IR process for structured text must take into account the inherent structure of the documents, in both representing and retrieving the structured documents. In this section, we briefly discuss the structured documents database work that is most relevant to ELIXIR.

The Proximal Nodes model [NBY97] permits any operations in which the fact that a node belongs to the final result can be determined by the identity and position of the node itself and of those nodes in the operands that are “proximal” to it. This query mechanism could be added to ELIXIR. However, the Proximal Nodes model does not support relevance ranking of query results. Another use of proximity of objects within a structured document is [GS98]. Any objects that are relevant to a user’s information need are first scored and ranked before they are presented to the user. This approach differs from that of [NBY97] in that proximity metrics are applied to nodes in more that one document object.

Finally, [Lal97, Lal99] describe a model in which a generalized belief function based on the Dempster-Shafer theory of evidence is used to determine the relevance of a document component, as opposed to WHIRL and ELIXIR’s “hardwired” use of the standard TF-IDF metric. Like ELIXIR, this approach produces ranked query results, but there is no discussion of how such answers can be efficiently generated for an arbitrary belief function.

3 The ELIXIR language

ELIXIR augments XML-QL with a primitive textual similarity operator, written “$\sim$”. This operator can be used just like the ordinary XML-QL operators such as “$<$”, “$=$”, and so forth. (Recall that, as discussed above, we ignore nested XML-QL queries.)

The semantics of ELIXIR combines aspects of the semantics of XML-QL and of WHIRL. Let Q be an ELIXIR query of the form `CONSTRUCT` C `WHERE` W. Since ELIXIR extends the non-nested subset of XML-QL, W consists only of pattern statements of the form “*pattern* `in` *source*” and predicates of the form “*variable operator variable*” or “*variable operator constant*”, where each “*operator*” is either an ordinary XML-QL operator (“$<$”, “$=$”, *etc.*) or ELIXIR’s “$\sim$” operator. Let W' be W with all similarity predicates removed, so that $S = W \setminus W'$ is the set of similarity predicates in W. For query Q_A in Figure 1(b), we have

$$\begin{aligned}
W &= \texttt{<items.*>\$t</> in "db.xml", \$t} \sim \texttt{"traditional Ukrainian"},\\
W' &= \texttt{<items.*>\$t</> in "db.xml"},\\
S &= \texttt{\$t} \sim \texttt{"traditional Ukrainian"},
\end{aligned}$$

while for Q_B in Figure 1(d) we have

$$\begin{aligned}
W &= \texttt{<items.book>\$b</> in "db.xml", <items.cd>\$c</> in "db.xml", \$b} \sim \texttt{\$c},\\
W' &= \texttt{<items.book>\$b</> in "db.xml", <items.cd>\$c</> in "db.xml",}\\
S &= \texttt{\$b} \sim \texttt{\$c}.
\end{aligned}$$

Suppose W contains M variables, $\texttt{\$x}_1, \ldots, \texttt{\$x}_M$. Note that W' must also contain M variables, since (like ordinary XML-QL operators) ELIXIR's "∼" cannot introduce new variables. Following [DFF+99a, DFF+99b], we observe that evaluating W' results in a relation $R(\texttt{\$x}_1, \ldots, \texttt{\$x}_M)$, with one column for each variable $\texttt{\$x}_i \in$W, and one tuple for each set of variables that satisfy W'. For query Q_B in Figure 1(d), we have

$$R(\texttt{\$b}, \texttt{\$c}) = \left\{ \begin{array}{c}
\langle \texttt{Traditional Ukrainian cookery,\ \ Traditional Ukrainian folk music} \rangle, \\
\langle \texttt{Traditional Ukrainian cookery,\ \ Being there} \rangle, \\
\langle \texttt{Traditional Ukrainian cookery,\ \ Milk cow blues} \rangle, \\
\langle \texttt{Being and nothingness,\ \ Traditional Ukrainian folk music} \rangle, \\
\langle \texttt{Being and nothingness,\ \ Being there} \rangle, \\
\langle \texttt{Being and nothingness,\ \ Milk cow blues} \rangle, \\
\langle \texttt{Shooting Elvis,\ \ Traditional Ukrainian folk music} \rangle, \\
\langle \texttt{Shooting Elvis,\ \ Being there} \rangle, \\
\langle \texttt{Shooting Elvis,\ \ Milk cow blues} \rangle
\end{array} \right\} \qquad (2)$$

(Note that this complete cross-product serves only to define ELIXIR's semantics. The ELIXIR query processing algorithm does not generate such full cross-products.)

The similarity predicates in S are either one-variable predicates of the form "*variable* ∼ *constant*" or similarity join predicates of the form "*variable* ∼ *variable*" . Let $S_1 = \{\ldots, \texttt{\$x}_i \sim c, \ldots\}$ be the set of one-variable similarity predicates and Let $S_2 = \{\ldots, \texttt{\$x}_i \sim \texttt{\$x}_j, \ldots\}$ be the set of similarity join predicates. Note that $S = S_1 \cup S_2$ and $S_1 \cap S_2 = \emptyset$. For query Q_A in Figure 1(b), $S_1 = \{\texttt{\$t} \sim \texttt{"traditional Ukrainian"}\}$ and $S_2 = \emptyset$, while for Q_B in Figure 1(d), $S_1 = \emptyset$ and $S_2 = \{\texttt{\$b} \sim \texttt{\$c}\}$.

The *score* of a similarity predicate when applied to a given pair of values was defined in Equation 1. The score of a tuple $t_i = \{\langle x_1, v_1^i \rangle, \langle x_2, v_2^i \rangle, \ldots, \langle x_M, v_M^i \rangle\} \in R$ for each $1 \le i \le |R|$, is defined to be the product of the scores of all the similarity predicates, with the variables bound as specified in t_i. More precisely, for tuple $t_i = \{\langle x_1, v_1^i \rangle, \langle x_2, v_2^i \rangle, \ldots, \langle x_M, v_M^i \rangle\} \in R$ we define:

$$\mathrm{score}(t_i) = \prod_{\texttt{\$x}_j \sim c \in S_1} \mathrm{sim}(v_j^i, c) \times \prod_{\texttt{\$x}_j \sim \texttt{\$x}_k \in S_2} \mathrm{sim}(v_j^i, v_k^i). \qquad (3)$$

The motivation of this definition is that, like WHIRL, we treat similarity scores as if they were probabilities, and treat a query's similarity predicates as independent events, so that the probability of their conjunction is the product of their probabilities.

Note that if an ELIXIR query does not contain any similarity predicates ($S = S_1 = S_2 = \emptyset$), then the query is an ordinary XML-QL query and score$(t_i) = 1$ for every $t_i \in R$.

We are now in a position to define the answer to an ELIXIR query. Following [DFF+99a, DFF+99b], we observe that the `CONSTRUCT` C clause generates one XML fragment for each tuple in R. The *complete answer* to an ELIXIR query is defined to be the sequence of $|R|$ such XML fragments, ordered by the score of the tuple $t_i \in R$ that generated the fragment. The ELIXIR query processor does not compute the *complete answer*s, but instead computes *r-answers.* The r-answer of an ELIXIR query Q is defined to be the r highest-scoring XML fragments of Q's complete answer. The motivation for this specification is that presumably an application is interested only in the highest-scoring answers, whereas all answers may be significant in traditional non-ranked query processing.

4 Efficiently answering ELIXIR queries

The ELIXIR query processing algorithm efficiently computes an r-answer to an ELIXIR query. Let Q_1 be an ELIXIR query. We seek the best r XML objects that are generated by Q_1, where the score of an XML object is defined by Equation 3. The algorithm proceeds in three stages:

- First, the algorithm generates and evaluates a set $\{\ldots, Q_2^i, \ldots\}$ of XML-QL queries. The `WHERE` clauses of these queries consist of one or more pattern or ordinary predicate statements copied directly from Q_1. The `CONSTRUCT` clause of each Q_2^i query builds (XML corresponding to) a relation, where each tuple encodes a set of bindings for all variables in Q_2^i's `WHERE` clause. To avoid generating the full cross-product of data values compared with similarity predicates, the algorithm partitions Q_1's pattern statements such that two such statements are placed in distinct Q_2^i queries if they contain variables that are compared with a similarity predicate.

- Second, the algorithm generates and evaluates a WHIRL query Q_3 to process the similarity predicates. Q_3 takes as input the relations built by the Q_2^i queries, and generates an ordered table of the r highest-scoring sets of variable bindings.

- Finally, the algorithm generates and evaluates one more XML-QL query Q_4. This query rewrites all tuples generated by Q_3 into the XML form specified by query Q_1's `CONSTRUCT` clause.

Figure 5 specifies the ELIXIR query processing algorithm in detail. We explain the algorithm by illustrating its operation on two example queries. Consider query Q_B in Figure 1(d). The algorithm first parses Q_B as follows:

`CONSTRUCT` clause C = `<item>$b</item>`

procedure <u>ANSWER</u>(ELIXIR query Q_1, integer r)

- Parse Q_1 to extract its root tag R, CONSTRUCT clause C, similarity predicates S, ordinary predicates O, and pattern statements P.
- $K_1, \ldots, K_N, D \leftarrow \textsc{partition}(P, O, S)$.
- For each $1 \leq i \leq N$:
  - Generate XML-QL query $Q_2^i =$

    $$\texttt{<q2}i\texttt{>}\{\texttt{CONSTRUCT <tuple><x}_1^i\texttt{>\$x}_1^i\texttt{</>}\ldots\texttt{<x}_{T_i}^i\texttt{>\$x}_{T_i}^i\texttt{</></> WHERE } K_i\}\texttt{</>}$$

    where K_i contains T_i variables $\texttt{\$x}_1^i, \ldots, \texttt{\$x}_{T_i}^i$. Invoke the XML-QL query processor on Q_2^i, and temporarily store the resulting tabular XML document as a relation $\texttt{q2}i$.

- Generate WHIRL query $Q_3 =$

  $$\texttt{q3(\$x}_1\texttt{,...,\$x}_T\texttt{) :- q21(\$x}^1_1\texttt{,...,\$x}^1_{T_1}\texttt{), ..., q2}N\texttt{(\$x}^N_1\texttt{,...,\$x}^N_{T_N}\texttt{), } S\texttt{, } D\texttt{.}$$

  where $\texttt{\$x}_1, \ldots, \texttt{\$x}_T$ are the variables in C, and $\texttt{\$x}_1^i, \ldots, \texttt{\$x}_{T_i}^i$ are the variables in K_i. Invoke the WHIRL query processor on Q_3 with the parameter r, and temporarily store the resulting relation q3 as a tabular XML document "q3.xml".

- Generate XML-QL query $Q_4 =$

  $$\texttt{<}R\texttt{>}\{\texttt{CONSTRUCT } C \texttt{ WHERE <q3.tuple><x}_1\texttt{>\$x}_1\texttt{</>...<x}_T\texttt{>\$x}_T\texttt{</></> in "q3.xml".}\}\texttt{</>}$$

  where $\texttt{\$x}_1, \ldots, \texttt{\$x}_T$ are the variables in C. Invoke the XML-QL query processor on Q_4 to generate the final r-answer to Q_1.

procedure <u>PARTITION</u>(pattern statements P, ordinary predicates O, similarity predicates S)

- Construct a graph from P, O and S, with nodes $P \cup O \cup S$, with an edge between $o \in O$ and $p \in P$ (written $o \leftrightarrow p$) iff o and p share a variable, and with an edge between $p \in P$ and $s \in S$ iff p and s share a variable.
- Partition the nodes $P \cup O$ as follows. First, $p \in P$ and $p' \in P$ are in the same subset iff there is no $s \in S$ such that $p \leftrightarrow s$ or $p' \leftrightarrow s$. Second, $p \in P$ and $o \in O$ are in the same subset iff $o \leftrightarrow p$, and there is no $s \in S$ and $p' \in P \neq p$ such that $p \leftrightarrow s$ and $o \leftrightarrow p' \leftrightarrow s$. Some of the resulting subsets will contain nodes from P; call these subsets $K_1, \ldots, K_N$. Other subsets will consist only of nodes from O; let D be the union of all such subsets. (Note that there may be multiple partitions consistent with these rules, in which case PARTITION returns one arbitrarily.)

Figure 5: The ELIXIR query processing algorithm.

$$
\begin{aligned}
\text{similarity predicates } S &= \{\texttt{\$b} \sim \texttt{\$c}\} \\
\text{ordinary predicates } O &= \emptyset \\
\text{patterns } P &= \{\texttt{<items.book>\$b</> in "db.xml", <items.cd>\$c</> in "db.xml"}\}
\end{aligned}
$$

Next, the algorithm invokes the PARTITION subroutine to separate the pattern statements of Q_1's `WHERE` clause so as to avoid computing unnecessary cross-products of the variables involved in similarity predicates. PARTITION builds the graph shown in Figure 6(a), and then partitions $P \cup O$ as:

$$
\begin{aligned}
K_1 &= \{\texttt{<items.book>\$b</> in "db.xml"}\} \\
K_2 &= \{\texttt{<items.cd>\$c</> in "db.xml"}\} \\
D &= \emptyset
\end{aligned}
$$

Intuitively, each K_i represents operations that can be efficiently processed together, in the sense that doing so does not generate unnecessary variable binding cross-products. In this example, the entire query can be decomposed in this fashion, but in general there might be ordinary predicates that operate on the same pairs of variables as the similarity predicates, so for the sake of efficiency these ordinary predicates must be deferred until after the similarity predicates have been processed. The D subset contains precisely the ordinary predicates that must be deferred in this manner.

As shown in Figure 7, the algorithm then generates two XML-QL Q_2^i queries, one for each K_i. Q_2^1 retrieves books from the XML database, and Q_2^2 retrieves CDs. The algorithm then generates the WHIRL query Q_3, which applies Q_B's similarity predicate. (Note that in Figure 7 we finesse the distinction between WHIRL's relational input and output, and a straightforward XML encoding of these tables.) WHIRL then generates the best r answers to this query. At this point, the correct data has been identified, but it is stored in a relation rather than in the XML structure specified by Q_B. The algorithm transforms this data by copying the original `CONSTRUCT` clause in Q_B to a final XML-QL query Q_4, which transforms WHIRL's output into the desired structure.

To understand why it is important to partition $P \cup O$, consider the alternative of creating one Q_2 query instead of Q_2^1 and Q_2^2:

alternative Q_2:
```
CONSTRUCT <tuple><b>$b</><c>$c</></>
WHERE     <items.book>$b</> in "db.xml",
          <items.cd>$c</> in "db.xml"
```

alternative Q_3: `q3($b, $c) :- q2($b, $c), $b ~ $c.`

This alternative approach does not scale well, because the output of Q_2 is the full cross-product of bindings for `$b` and `$c` shown in Equation 2.

This first example query is relatively simple, because it contains just a single similarity predicate and no ordinary predicates. We complete the presentation of the ELIXIR query processing algorithm by describing how it would answer the following more complicated query:

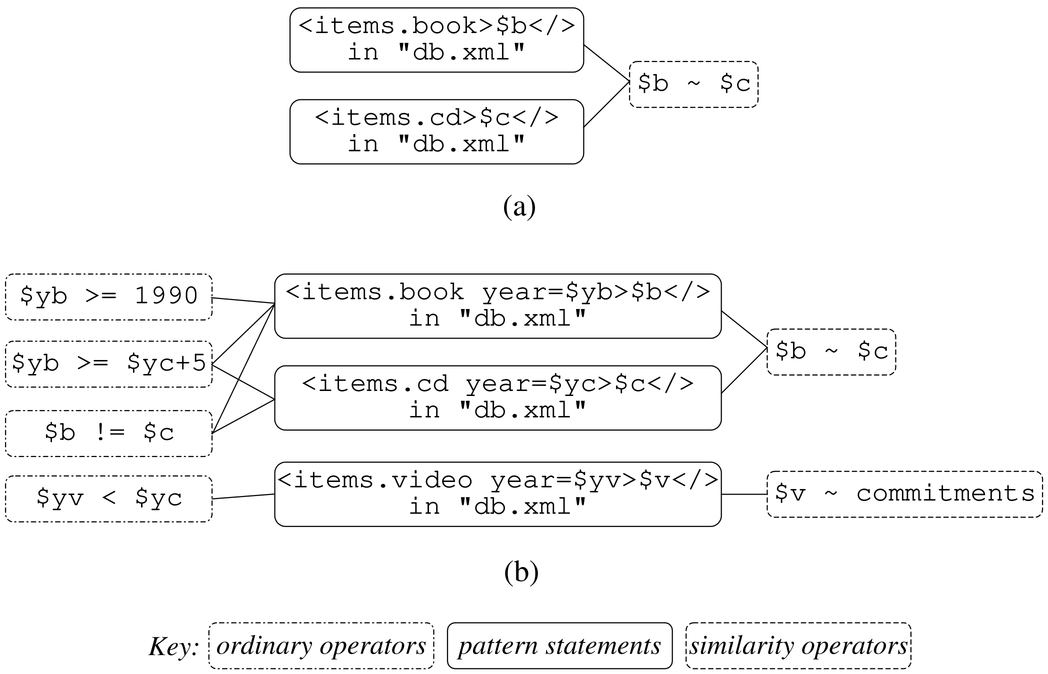

Figure 6: Graphs generated by the PARTITION subroutine for two example queries.

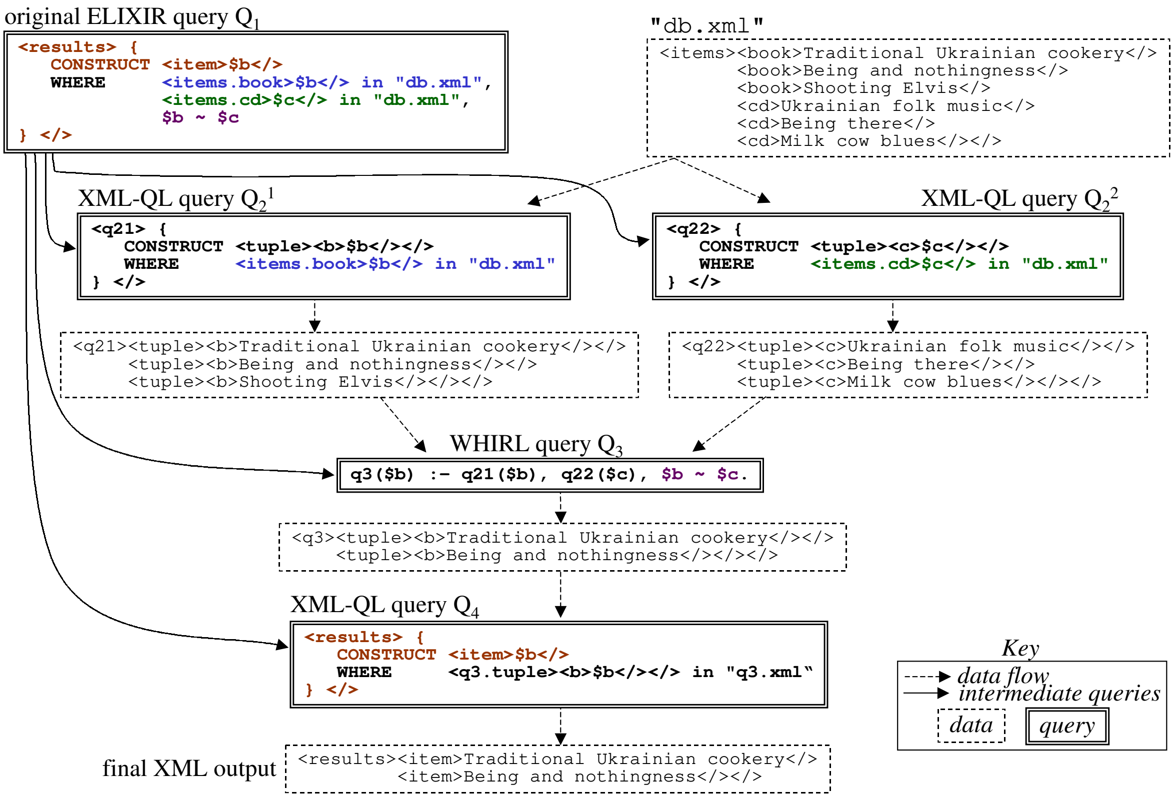

Figure 7: The ELIXIR query processor efficiently generates an *r-answer* to a query Q_1 by rewriting Q_1 into one or more XML-QL Q_2^i queries, a WHIRL query Q_3, and a final XML-QL query Q_4. Colored text indicates which parts of Q_1 are used to generate the intermediate queries.

```
CONSTRUCT <stuff book=$b cd=$c video=$v>
WHERE     <items.book year=$yb>$b</> in "db.xml",
          <items.cd year=$yc>$c</> in "db.xml",
          <items.video year=$yv>$v</> in "db.xml",
          $yb > $yc+5,
          $yb > 1990,
          $b != $c,
          $yv < $yc,
          $b ~ $c,
          $v ~ "dog".
```

(This query is not intended to be sensible or useful, but merely to illustrate our query processing algorithm.)

Our query processor would first parse this query as follows:

CONSTRUCT clause C = `<stuff book=$b cd=$c video=$v>`,

similarity predicates S = {`$b` ~ `$c`, `$v` ~ `"dog"`},

```
ordinary predicates O = {$yb>$yc+5, $yb>1990, $b!=$c, $yv<$yc},

patterns P = { <items.book year=$yb>$b</> in "db.xml",
               <items.cd year=$yc>$c</> in "db.xml",
               <items.video year=$yv>$v</> in "db.xml" }
```

The PARTITION subroutine would construct the graph shown in Figure 6(b), and then return the following partition:

```
K1 = {<items.book year=$yb>$b</> in "db.xml", $yb>1990}
K2 = {<items.cd year=$yc>$c</> in "db.xml"}
K3 = {<items.video year=$yv>$v</> in "db.xml"}
D  = {$yb>$yc+5, $b!=$c, $yv<$yc}
```

This partition of $P \cup O$ into D, K_1, K_2 and K_3 expresses the fact that each type of item should be retrieved separately: many of the items will be discarded when WHIRL computes an r-answer to Q_3 below, and thus there is no point in computing a cross-product of the variable bindings only to have many of these bindings discarded by WHIRL. For the same reason, the deferred ordinary predicates D—those predicates whose variables span K_i subsets —should be processed not in any Q_2^i query (which would generate unnecessary variable binding cross-products) but only in the WHIRL query Q_3.

Finally, the algorithm generates the following five intermediate queries:

```
       CONSTRUCT <tuple><b>$b</><yb>$yb</></>
Q2^1 = WHERE     <items.book year=$yb>$y</> in "db.xml",
                 $yb > 1990.

Q2^2 = CONSTRUCT <tuple><c>$c</><yc>$yc</><v>$v</><yv>$yv</></>
       WHERE     <items.cd year=$yc>$c</> in "db.xml".

Q2^3 = CONSTRUCT <tuple><v>$v</><yv>$yv</></>
       WHERE     <items.video year=$yv>$v</> in "db.xml".

       q3($b, $yb, $c, $yc, $v, $yv) :-
Q3   =     q21($b, $yb), q22($c, $yc), q23($v, $yv),
           $b ∼ $c, $v ∼ "dog", $yv < $yc, $yb > $yc+5, $b != $c.

       CONSTRUCT <stuff book=$b cd=$c video=$v/>
Q4   = WHERE     <q3.tuple><b>$b</><yb>$yb</><c>$c</><yc>$yc</>
                           <v>$v</><yv>$yv</></> in "q3.xml".
```

Note that our algorithm rewrites an ELIXIR query as a three-stage pipeline, with PARTITION assigning various parts of the query to each stage according to a fixed policy. However, as discussed in Section 6, this particular policy may not always be optimal with respect to standard metrics such as the size of intermediate data. In future work we will explore optimal query rewriting techniques.

To summarize, our query processing algorithm rewrites the original ELIXIR query Q_1 into a series, Q_2^1, $\ldots$, Q_2^N, of XML-QL queries that retrieve the XML data needed for later similarity-based ranking. Q_1's ordinary predicates are evaluated in a Q_2^i query if doing so would not require unnecessarily computing a cross-product of variable bindings. Once the required data is retrieved and filtered as much as possible, the WHIRL query Q_3 generates the data needed to create an r-answer to Q_1. Finally, an XML-QL query Q_4 transforms Q_3's output into the XML structure specified by query Q_1. We note that a straightforward implementation of PARTITION runs in time $O(|S| \cdot |O| \cdot |P|)$), and thus ELIXIR's expressiveness adds a negligible overhead to the underlying XML-QL and WHIRL query processors.

5 Experiments

Our ELIXIR query processing algorithm has been implemented in Java. In this section we report detailed empirical results for thirteen queries against five XML databases; see Figure 8. We used version 0.9 of the XML-QL query processor from AT&T Research and the University of Pennsylvania[3], and version 1.6 of the WHIRL query processor from AT&T Research[4]. The experiments were carried out on a 500MHz Pentium III processor with 256MB of RAM running Red Hat Linux 6.2 and Sun's Java 2 SDK version 1.3.

We used five XML databases widely discussed in the XML research community: the SIGMOD Record table of contents[5], the New Testament[6], and Shakespeare's[7] Macbeth, Julius Caesar, and Romeo and Juliet. To give an indication of their size and complexity, Figure 8 lists, for each database, the size in bytes and number of tags.

The queries in Figure 8 are ordered by the total CPU time required for processing. The first nine columns describe the query itself. As an indication of the query's complexity, we report the number of similarity joins, simple (*i.e.* non-join) similarity predicates, and pattern statements in each query. The "$\surd$" marks in columns 4–8 indicate which queries use which databases. The ninth column indicates the sum of the sizes of the input databases for each query.

The final five columns indicate the performance of the query processing algorithm. Column ten indicates the size of the intermediate data passed from the XML-QL Q_2^i queries to the WHIRL Q_3 query. Column eleven gives the size of the final output generated by Q_4. Note that the size of Q_3's output is approximately the same as that final output from Q_4, so we do not report these values. Note also that the size of the final output is largely determined by r, the number of answers requested; in all cases we use $r = 20$. The final three columns list the CPU time required to answer each query: column twelve is the total CPU time

[3]http://www.research.att.com/sw/tools/xmlql
[4]http://whirl.research.att.com
[5]http://www.acm.org/sigmod/record/xml/index.html
[6]http://metalab.unc.edu/bosak/xml/eg/rel200.zip
[7]http://metalab.unc.edu/bosak/xml/eg/shak200.zip

query	number of similarity join predicates	number of simple similarity predicates	number of pattern statements	SIGMOD Record (482KB, 7.8K tags)	New Testament (1022KB, 8.6K tags)	Macbeth (165KB, 4.0K tags)	Julius Caesar(185KB, 4.5K tags)	Romeo and Juliet (219KB, 5.1K tags)	Input data size (KB)	Q_3 input data size (KB)	Output data size (KB)	Total Q_2^i time (CPU seconds)	Q_3 time (CPU seconds)	Total time (CPU seconds)
1	1	0	2			√	√		350	5.2	3.7	9	1	14
2	1	0	2			√	√		350	210	2.4	15	2	21
3	1	0	2		√				2044	117	3.3	19	2	25
4	1	0	2			√	√		350	456	3.7	23	5	32
5	0	1	1	√					482	19	2.9	30	1	35
6	1	1	2	√					964	408	3.0	39	3	47
7	1	0	2		√				2044	1403	4.3	46	7	58
8	1	0	2	√	√				1504	1451	4.0	51	20	75
9	1	0	2	√					964	1052	3.6	70	11	89
10	1	0	2	√					964	367	2.6	110	8	129
11	2	0	3	√	√	√			1669	1658	4.6	69	60	136
12	3	0	4	√	√	√	√		1854	2021	5.4	78	97	183
13	4	0	5	√	√	√	√	√	2073	2309	6.5	90	114	213

Figure 8: Performance of thirteen ELIXIR queries over five XML databases.

consumed by the Q_2^i queries; column thirteen is the CPU time consumed by Q_4, and the last column gives the total CPU time for the entire ELIXIR query. Note that the total CPU time reported includes the time to rewrite the ELIXIR query into the Q_2^i, Q_3 and Q_4, as well as the time to execute Q_4, but these times are negligible compared to the total and thus we do not report these values explicitly.

As an example of the queries used in our experiments, query 11 requests recent SIGMOD Record papers titles, New Testament verses, and lines from Macbeth that are all similar:

```
CONSTRUCT <similar><title>$t</><verse>$v</><line>$l</></>
WHERE     <SigmodRecord.issue>
              <volume>$vol</>
              <articles.article.title>$t</>
          </> in "SIGMODRecord.xml",
          <tsts.bookcoll.book.chapter.v>$v</> in "NewTest.xml",
          <PLAY.ACT.SCENE.SPEEC.LINE.PCDATA>$l</> in "Macbeth.xml",
          $vol > 24, $t ~ $v, $t ~ $l.
```

Answers discovered by ELIXIR include:

```
<similar>
    <title>Opportunities in Information Management and Assurance.</>
    <verse>And from that time he sought opportunity to betray him.</>
    <line>But yet I'll make assurance double sure,</></>
<similar>
    <title>Size Separation Spatial Join.</>
    <verse>But he that is joined unto the Lord is one spirit.</>
    <line>To Ireland, I; our separated fortune</></>
<similar>
    <title>Where Will Object Technology Drive Data Administration?</>
    <verse>And there are differences of administrations, but the same Lord.</>
    <line>Where?</></>
...
```

It is difficult to draw clear trends from the data in Figure 8, because query processing time depends, in complex ways, on the details of each query and its input data. Note that the XML-QL Q_2^i queries usually dominates the total time to answer the query. This reflects the fact that the underlying process of parsing and traversing the input XML data is expensive, even without any similarity predicates. In an attempt to see how well our ELIXIR query processing algorithm scales, we now report the results for several families of queries that result from varying one or more query parameters.

Our first experiment investigated how ELIXIR scales with the size of the intermediate data generated by the Q_2^i queries. To vary the size of the intermediate data, we modified query 6 from Figure 8:

```
CONSTRUCT <similar.title>$t1</>
WHERE     <SigmodRecord.issue>
              <volume>$v1</>
              <articles.article.title>$t1</>
          </> in "SIGMODRecord.xml",
          <SigmodRecord.issue>
              <volume>$v2</>
              <articles.article.title>$t2</>
          </> in "SIGMODRecord.xml",
          $v1 > 24, $v2 > 24, $t1 ~ $t2.
```

By varying the volume threshold (24 in this query) we can vary the amount of intermediate data passed from the XML-QL Q_2^i queries to the WHIRL query Q_3. As shown in Figure 9, the total time to process these queries increases linearly with the size of the intermediate data.

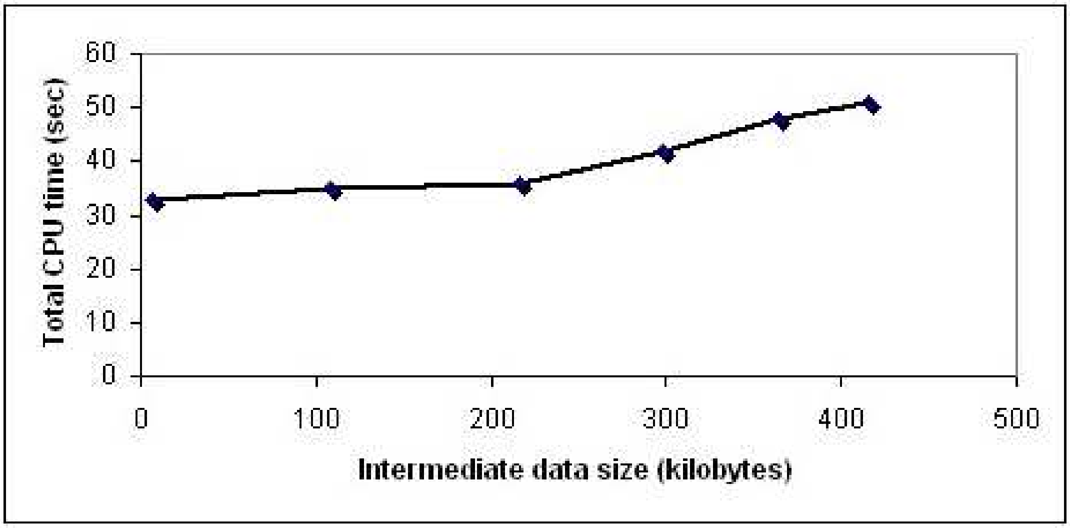

Figure 9: Total ELIXIR query processing time increases with the size of the intermediate data.

Our second experiment investigated how the number of answers, r, affects the performance of our ELIXIR query processor. Figure 10 shows the results of systematically varying r from 1 to 50, for queries 10 and 11 from Figure 8. These data demonstrate that, for these queries, there is a marginal increase in total query processing time as r increases.

A third experiment measured the effect of the numbers of similarity join predicates on the total ELIXIR query processing time. We used four ELIXIR queries, containing from one to four similarity join predicates. The most complicated query was query 13 from Figure 8:

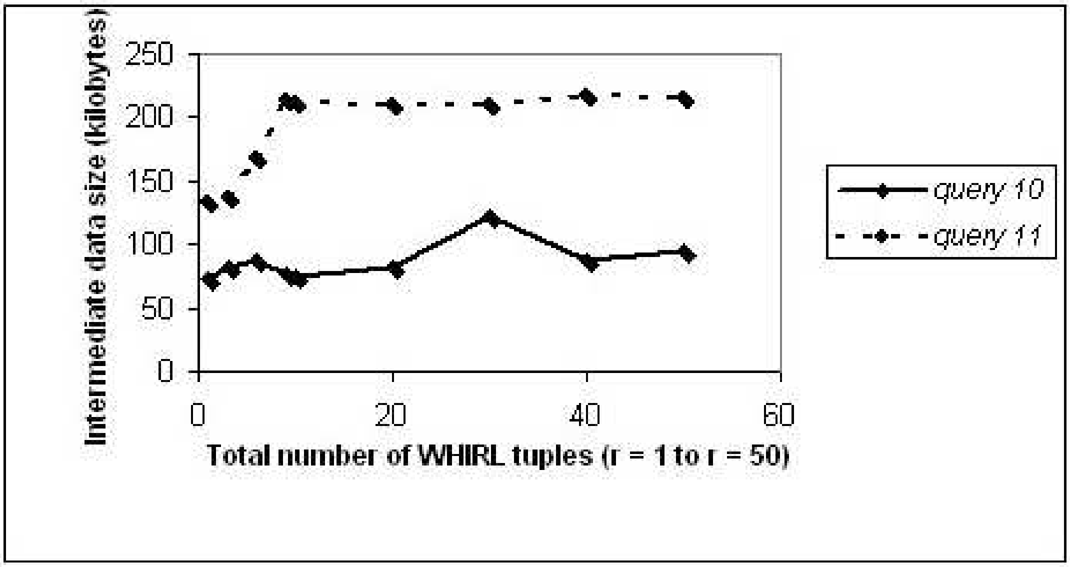

Figure 10: Total ELIXIR query processing time increases marginally as the number of desired answers (r) increases.

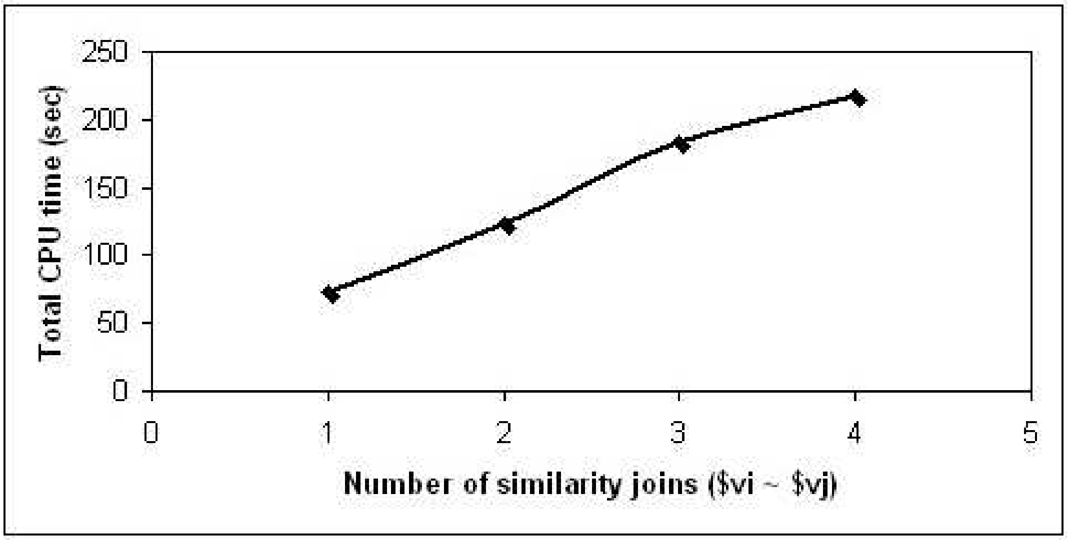

Figure 11: Total ELIXIR query processing time increases linearly with the number similarity joins predicates.

```
CONSTRUCT <similar><title>$t</><verse>$v</><macb>$m</><jc>$j</><rj>$r</></>
WHERE     <SigmodRecord.issue.articles.article.title>$t</>
              in "SIGMODRecord.xml",
          <tstmt.bookcoll.book.chapter.v>$v</>
              in "NewTest.xml",
          <PLAY.ACT.SCENE.SPEECH.LINE.PCDATA>$m</>
              in "Macbeth.xml",
          <PLAY.ACT.SCENE.SPEECH.LINE.PCDATA>$j</>
              in "J_Caesar.xml",
          <PLAY.ACT.SCENE.SPEECH.LINE.PCDATA>$r</>
              in "Romeo_and_Juliet.xml",
          $t ~ $v, $m ~ $j, $j ~ $r, $t ~ $m.
```

The other queries were formed by removing one similarity join at a time from this query. Figure 11 shows that total ELIXIR query processing time increases linearly with the number of similarity join predicates in these queries.

To summarize, we have reported detailed performance results for a set of ELIXIR queries, and we have systematically varied aspects of these queries in order to measure how well the algorithm scales. While we have not yet conducted thorough experiments on a real-world distribution of queries, we conclude that the

ELIXIR query processing algorithm scales reasonably well with a query's size and complexity.

6 Discussion

Summary. We have described ELIXIR, an expressive and efficient language for XML IR. Inspired by the WHIRL system, ELIXIR extends the existing XML query language XML-QL with a primitive similarity operator "$\sim$" that orders the output according to textual similarity. Unlike previous efforts to integrate keyword search into XML, ELIXIR both supports genuine ranked retrieval (the output is ordered according to the similarity constraints stated in the query), and permits similarity joins ("$\sim$" can be applied to two variables, rather than just a variable and a constant). ELIXIR is thus sufficiently expressive to be a candidate for a general purpose XML IR query language. In this paper we have focused on XML data retrieval, integration and transformation applications that involve text fragments that have not been or cannot be normalized.

As our experiments demonstrate, our ELIXIR query processing algorithm efficiently computes the best r answers to a query. To do so, the algorithm rewrites the original ELIXIR query into a series of intermediate XML-QL and WHIRL queries, and then executes them in sequence to generate an XML object that answers the original ELIXIR query. Unlike systems that automatically map XML data to a relational model, ELIXIR does not map the XML data *per se*, but only tuples of variable bindings. WHIRL then efficiently orders these tuples of bindings according to the similarity predicates in the original ELIXIR query.

Future work We are currently exploring several open issues regarding ELIXIR and ranked XML query languages.

Our implementation of ELIXIR extends the non-nested subset of XML-QL. We are currently extending the implementation to handle nested queries. We envision a straightforward extension of our approach, with nested queries simply invoking the ELIXIR query processor recursively.

Our implementation of ELIXIR was rather straightforward, since XML-QL is a relatively simple language. We therefore are exploring the problems that will be encountered when we apply our ideas to more complicated languages such as Quilt [CRF00] or XQuery [CFR$^+$01], the W3C draft standard XML query language. Our intuition is that the rewriting process will be somewhat more complicated, but we think that the fundamental ideas underlying our approach will remain relevant.

We have not yet explored optimization issues. The ELIXIR query processor follows a strict three-stage approach to rewriting the original query, but this may be sub-optimal in some cases. For example, consider a query that searches for books about Elvis that are similar to newer CDs:

```
CONSTRUCT <book>$b</>
WHERE     <items.book year=$yb>$b</> in "db.xml",
          <items.cd year=$yc>$c</> in "db.xml",
          $b ~ $c, $b ~ "elvis", $yb > $yc.
```

Should `$yb > $yc` be evaluated "early" (in one of the Q_2^i queries) or "late" (in query Q_3)? Our PARTITION algorithm greedily defers such predicates until Q_3, to avoid generating the full cross-product of book/CD pairs. Suppose there are N books and M CDs in the database, and $n < N$ books are published after some CD, and $m < M$ CDs are published before some book. Our current policy is optimal only if $N + M < nm$. Note also that the current approach assigns both similarity predicates to a single WHIRL query Q_3. However, if there are very few books about Elvis, then a better rewriting would be to generate one WHIRL query to find such books, and then compare them to the CDs with a second WHIRL query. As shown in Figure 12, we are exploring techniques for automatically searching the space of query rewritings in order to find one that is optimal with respect to standard metrics such as intermediate data sizes.

Finally, we are interested in using ELIXIR to support a richer variety of information access tasks beyond simple retrieval. For example, the IR community has explored techniques such as query reformulation and relevance feedback to help users generate effective queries, and clustering techniques to help users visualize large information spaces. It would be interesting to see whether ELIXIR could be used to extend such ideas from sets of flat documents to structured XML text databases.

Acknowledgments. This research was supported by Enterprise Ireland grant ST/1999/071. We thank Michela Bertolotto, Arthur Cater, Joe Carthy, William Cohen, Kai Großjohann, Alan McClean, Mel Ó Cinnéide and Mounia Lalmas for helpful discussions, and the XML-QL and WHIRL developers for making their systems available.

References

[ABD00] S. Abiteboul, P. Buneman, and Suciu. D. *Data on the Web: From relations to semistructured data and XML.* Morgan Kauffman, 2000.

[BYRN99] R. Baeza-Yates and B. Ribeiro-Neto. *Modern information retrieval.* Addison Wesley, 1999.

[CFR+01] D. Chamberlin, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu. XQuery: A query language for XML, 2001. http://www.w3.org/TR/2001/WD-xquery-20010215.

[Coh98] W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *Proc. SIGMOD*, pages 201–211, 1998.

[Coh00] W. Cohen. WHIRL: A word-based information representation language. *J. Artificial Intelligence*, 118(163–196), 2000.

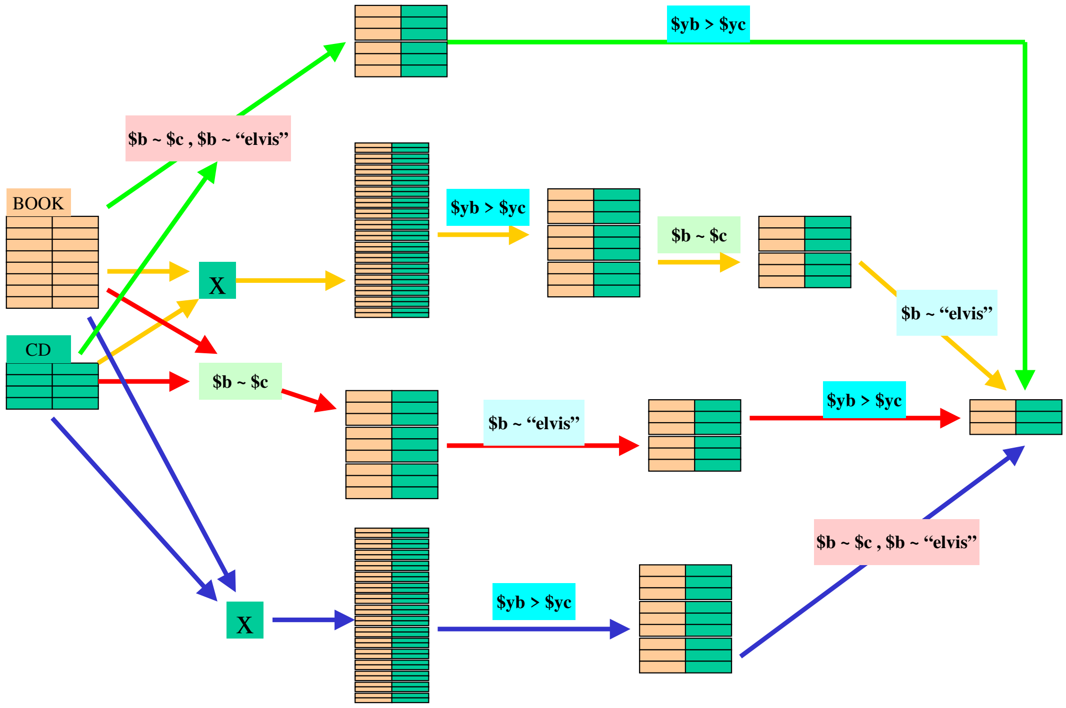

Figure 12: ELIXIR query optimization involves finding a query rewriting that is optimal with respect to the size of the intermediate data. Each path in this graph represents a distinct rewrite of an ELIXIR query; our implementation currently uses the top-most rewrite, but this policy may be suboptimal depending on the properties of the data sources.

[CRF00] D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML query language for heterogeneous data sources. In *Proc. SIGMOD/PODS Workshop on the Web and Databases*, 2000.

[DFF+99a] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, , and D. Suciu. XML-QL: A query language for XML. In *Proc. 8th Int. World Wide Web Conf.*, 1999.

[DFF+99b] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, D. Maier, and D. Suciu. Querying XML data. *Data Engineering Bulletin*, 22(3):10–18, 1999.

[DFS99] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *Proc. SIGMOD*, 1999.

[FG00] N. Fuhr and K. Großjohann. XIRQL: An extension of XQL for information retrieval. In *SIGIR Workshop on XML and Information Retrieval*, 2000.

[FMK00] D. Florescu, I. Manolescu, and D. Kossmann. Integrating keyword search into XML query processing. In *Proc. 9th Int. World Wide Web Conf.*, 2000.

[FTS00] M. Fernandez, W.-C. Tan, and D. Suciu. SilkRoute: Trading between relations and XML. In *Proc. 9th Int. World Wide Web Conf.*, 2000.

[Glo] Globit.com. Infonyte. http://www.infonyte.com.

[GS98] R. Goldman and N. Shivakumar. Proximity search in databases. In *24th Int'l VLDB conference*, 1998.

[HJG00] Y. Hayashi, Tomita J., and Kikui G. Searching text-rich XML documents with relevance ranking. In *SIGIR Workshop on XML and Information Retrieval*, 2000.

[HS95] M. Hernandez and S. Stolfo. The merge/purge problem for large databases. In *Proc. SIGMOD*, 1995.

[KLD+98] A. Knoblock, A. Levy, O. Duschka, D. Florescu, and N. Kushmerick, editors. *Proc. 1998 Workshop on AI and Information Integration.* AAAI Press, 1998.

[Lal97] M. Lalmas. Dempster-Shafer's theory of evidence applied to structured documents: Modelling uncertainty. In *Proc. Int. SIGIR Conf. Research and Development in Information Retrieval*, pages 110–118, 1997.

[Lal99] M. Lalmas. A model for representing and retrieving heterogeneous structured documents based on evidential reasoning. *The Computer Journal*, 42:547–568, 1999.

[LKMC98] A. Levy, C. Knoblock, S. Minton, and W. Cohen. Trends and controversies: Information integration. *IEEE Intelligent Systems*, 13(5):12–24, 1998.

[LPVV99] B. Ludaescher, Y. Papakonstantinou, P. Velikhov, and V. Vianu. View definitions and DTD inference for XML. In *Prob. Workshop on Semistructured data and nonstandard data formats*, 1999.

[ME96] A. Monge and C. Elkan. The field matching problem: Algorithms and applications. In *Proc. Int. Conf. Knowledge Discovery and Data Mining*, 1996.

[MHF99] I. Macherius, G. Huck, and P. Fankhasuer. XQL extensions in the GMD-IPSI XQL Engine, 1999. http://xml.darmstadt.gmd.de/xql/extensions.

[NBY97] G. Navarro and R. Baeza-Yates. Proximal nodes : A model to query document databases by content and structure. In *Proc. ACM Transcations on Information Systems*, pages 400 – 435, 1997.

[NDM00] J. Naughton, D. DeWitt, and D. Maier. The Niagara Internet query system, 2000. http://www.cs.wisc.edu/niagara.

[Rob99] J. Robie. XQL (XML Query Language), August 1999. http://metalab.unc.edu/xql/xql-proposal.xml.

[Sal89] G. Salton, editor. *Automatic text processing.* Addison Wesley, 1989.

[Sof] SoftwareAG.com. Tamino: The XML Power Database. http://www.softwareag.com/tamino.

[STH+99] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *Proc. Int. Conf. Very Large Databases*, 1999.

[TW00] A. Theobald and G. Weikum. Adding relevant to XML. In *Proc. SIGMOD/PODS Workshop on the Web and Databases*, 2000.

[UW97] J. Ullman and J. Widom. *A first course in database systems.* Prentice Hall, 1997.

[Wie92] G Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, pages 38–49, March 1992.

[XYZ] XYZFind.com. XYZFind: The schema-independent native XML database. http://www.xyzfind.com.