XQuery Processing with Relevance Ranking

Leonidas Fegaras

University of Texas at Arlington
416 Yates Street, P.O. Box 19015
Arlington, TX 76019-19015, USA,
fegaras@cse.uta.edu

Abstract. We are presenting a coherent framework for XQuery processing that incorporates IR-style approximate matching and allows the ordering of results by their relevance score. Our relevance ranking algorithm is based on both stem matching and term proximity. Our XQuery processor is stream-based, consisting of iterators connected into pipelines. In our framework, all values produced by XQuery expressions are assigned scores, and these scores propagate and are combined when piped through the iterators. The most important feature of our evaluation engine is the use of structural and content-based inverse indexes that deliver data in document order and facilitate the use of efficient merge joins to evaluate path expressions and search predicates. We present the rules for the translation from a large part of XQuery to iterator pipeline. Our modular approach of building pipelines to evaluate XQuery scales up to any query complexity because the pipes can be connected in the same way complex queries are formed from simpler ones.

1 Introduction

It has been noted since its conception that the structured nature of XML allows a better representation and more precise querying of text documents. Even though current XML query languages, such as XPath and XQuery [5], are very powerful in expressing exact queries over XML data, they do not meet the needs of the IR community since they do not support approximate matching based on textual similarity with relevance ranking. There are already a number of recent proposals on extending the XQuery language with IR-style search primitives, such as TexQuery [3,6], XQuery/IR [7], ELIXIR [9], XIRQL [10], XXL [17,18,19], XRANK [13], and XIRCUS [15]. Furthermore, there is an ongoing effort by the W3C community to provide usage scenarios for full-text queries [22], but there is still no emerging standard for a full-text extension to XQuery. It is not the goal of this paper to propose yet another document-centric language for XML documents. Instead, we present a framework for XQuery processing in which the most important features of these emerging proposals can be seamlessly incorporated to the XQuery semantics and can be processed efficiently.

More specifically, the objectives of this work are:

1. to use simple but powerful language extensions to XQuery based on current proposals to accommodate IR-style approximate matching with relevance

ranking; these syntactic extensions should require minimal changes to the existing XQuery semantics and should be amenable to efficient evaluation;
2. to design an indexing scheme for XML data for efficient evaluation of both search predicates and path navigation;
3. to build a highly pipelined XQuery engine, called *XQueryRank*, that evaluates XQueries against the indexes using merge joins exclusively and without materializing intermediate results;
4. this engine should consist of iterators that naturally reflect the syntactic structures of XQuery and can be composed into pipelines in the same way the corresponding XQuery structures are composed to form complex queries;
5. the XQuery translation should be concise, clean, and completely compositional, so that optimizations can be easily incorporated later;
6. even though the prototype system is not intended to be complete, since XQuery is a very complex language, it should be designed in such a way that all supported language features can be be incorporated later without strain.

The main contribution of this paper is the development of a coherent framework for XQuery processing that incorporates IR-style approximate matching and allows the ordering of results by their relevance score. Our relevance ranking algorithm is based on both stem matching and term proximity. Our XQuery processor is stream-based, consisting of *iterators* connected into pipelines [12]. This pipeline model avoids materialization of intermediate results when possible. In our framework, all values produced by XQuery expressions are assigned scores, and these scores propagate and are combined when piped through the iterators. The scoring is done implicitly by the stream iterators that evaluate the query. The most important feature of our evaluation engine is the use of structural and content-based inverse indexes that deliver data in document order and facilitate the use of efficient merge joins to evaluate path expressions and search predicates. We present the rules for the translation from a large part of XQuery (which includes the search extensions) to query plans that represent the iterator pipeline.

The closest work to ours is the TIX algebra [2]. Like the TAX algebra, TIX works on pattern trees, which captures path expressions over single documents, even though there is a proposed syntax for incorporating relevance ranking to XQuery that corresponds to the TIX operators. With XQueryRank, on the other hand, one may correlate multiple documents in the same query, may query all indexed documents at once, and may use any kind of query nesting and complexity. More importantly, we give the full details of the XQuery translation into efficient evaluation plans. Furthermore, while their term matching algorithm is limited to conjunctions of search terms, our method is as effective for disjunction and negation. Finally, in contrast to our approach, which propagates and combines relevance scores when fragments are piped through operators, TIX trees are annotated with relevance scores, called scored trees. This is a non-functional approach that does not work well with the functional semantics of XQuery, because the same tree may be assigned a different score under a different context.

2 Approximate Matching in XQuery

There are already a number of recent proposals on extending the XQuery language with IR-style search primitives. It is not the main goal of this paper to propose yet another document-centric query language; rather, we aim at developing a framework in which these query language extensions can be easily incorporated into an XQuery processor by providing a language that reflects the most important features of these proposals. We are using the following simple extensions to XQuery that have been influenced by TexQuery [3,6]:

1. the expression document() that matches any indexed document and returns the top level elements of all these documents. The order of the documents is unspecified and may depend on the time the documents were indexed.
2. the boolean IR predicate $e \sim S$, where e is any XQuery expression, that returns true if at least one element from the sequence returned by e matches the IR *search specification*, S. The specification S has the following syntax:

$$\begin{array}{rll} S,\, S_1,\, S_2 ::= & \text{“}text\text{”} & \text{phrase} \\ | & S_1 \ \underline{\text{and}}\ S_2 & \text{conjunction} \\ | & S_1 \ \underline{\text{or}}\ S_2 & \text{disjunction} \\ | & \underline{\text{not}}\ S & \text{negation} \end{array}$$

   Note that a phrase must be present in the text of some descendant of the element e and must be in consecutive words that do not cross element boundaries.
3. the function score(e), for each element from the XQuery expression e, returns its score (the relevance assessment). A score is a real number between 0 and 1: a zero score means false, while a non-zero score means true. The closer the value score is to 1, the more the value is relevant to the query result.

For example, the following query

```
<answer>{
( for $db in document()/biblio,
      $b in $db/bib[title ~ ("XQuery processing" and "relevance")]
  where $b/abstract ~ ("SAX" and not "DOM")
  order by score($b) descending
  return <paper>{ $b/author/name, $b/title, score($b) }</paper>
)[position()<=10]
}</answer>
```

searches all indexed documents for papers that contain the phrase “XQuery processing” and the word “relevance” in the title, and whose abstract contains the word “SAX” but not the word “DOM”. The resulting papers are ordered by descending order of relevance and the top ten relevant papers are returned. The result of the query consists of links to document fragments (ie, triples with the document location and the begin/end positions of an element), rather than the actual text of the fragments, because it is impossible to reconstruct the

original text from the inverse indexes alone. A user may in turn dereference some of the returned links, assuming that the original XML documents are still available. Note also that, if not ordered, the resulting papers will be listed in the document order, regardless of scores, as is enforced by the XQuery semantics. Since there may be many indexed documents involved, if not ordered, the paper listings of each document would appear together and the order of documents may depend on the time they were indexed. As expected, scores are propagated and combined across path expressions and other XQuery syntactic constructs, so that results from any kind of XQuery expression can be assigned scores. Finally, words in phrases must be continuous in the XML document and term weighting is based on both stem matching and term proximity.

2.1 Semantics of the XQuery Extensions

In our framework, an XML element in an indexed XML document is assigned a triple (begin,end,level), based on its position in the document (ie, the positions of its begin/end tags) and on its depth level [21]. If an element x is a child of an element y in an XML document, then the (begin,end) region of x is contained by the corresponding region of y and the level of x is the level of y plus one. Furthermore, a preceding sibling of an element has a preceding, non-overlapping region. This numbering scheme, called *region encoding* [21], has already been used by many systems for effective XML indexing and efficient path expression evaluation.

An XML fragment (called XF) is an XML element from an indexed document that may contain text. Text is tokenized into terms (words) that reflect the linguistic tokens of a given human language. A term in an XF is assigned a pair (position,level) so that the position of the term is contained by the XF's region and the position order reflects the term order in the XF. The level of a term is equal to the level of the enclosing XF. In addition, to facilitate phrase matching and proximity calculations, our numbering scheme guarantees that consecutive terms be assigned succeeding positions.

Since the unit of XQuery processing is a sequence of XML elements, the unit of full-text searching should be a sequence of XF elements, rather than the entire document. A term in an XF can be assigned a weight $\mathcal{W}[\![\text{term}]\!]$ based on the standard IR term-frequency/inverse-document-frequency (tf-idf) relevance ranking. We are using the following weight:

$$\frac{\text{frequency-of-term-in-XF}}{\text{number-of-terms-in-XF}} \times \log\left(\frac{\text{term-level} - \text{e-level} + 1}{\text{e-level}}\right) \times \log\left(\frac{\text{XFs-containing-term}}{\text{total-number-of-XFs}}\right)$$

where e-level is the level of the element e being tested for matching. The first term is the frequency of the term in the XF, the second term depends on how close is the level of the term with that of the element under consideration, and the last term depends on how rare is the term in the database (rare terms are given higher weight). The boolean connectives in a search specification can be

assigned a score based on the probabilistic interpretation of weights:

$$\begin{aligned}\mathcal{W}[\![S_1 \underline{\text{ and }} S_2]\!] &= \mathcal{W}[\![S_1]\!] \times \mathcal{W}[\![S_2]\!] \\ \mathcal{W}[\![S_1 \underline{\text{ or }} S_2]\!] &= \mathcal{W}[\![S_1]\!] + \mathcal{W}[\![S_2]\!] - \mathcal{W}[\![S_1]\!] \times \mathcal{W}[\![S_2]\!] \\ \mathcal{W}[\![\underline{\text{not }} S]\!] &= 1 - \mathcal{W}[\![S]\!]\end{aligned}$$

As it can be easily proven, this interpretation preserves almost all the boolean laws, such as $\mathcal{W}[\![\underline{\text{not}}\ (S_1 \underline{\text{ and }} S_2)]\!] = \mathcal{W}[\![(\underline{\text{not}}\ S_1) \underline{\text{ or }} (\underline{\text{not}}\ S_2)]\!]$ and $\mathcal{W}[\![\underline{\text{not}}\ (\underline{\text{not}}\ S)]\!] = \mathcal{W}[\![S]\!]$, but does not preserve idempotent laws, such as $(S \underline{\text{ and }} S) = S$. Unfortunately, this probabilistic interpretation is not very useful, since it does not take term proximity into account; the following interpretation though does.

Consider a conjunction of n terms tested against an XF

$$\text{“}T_1\text{”} \underline{\text{ and }} \ldots \underline{\text{ and }} \text{“}T_n\text{”}$$

where some instance of the term “T_i” is located at position p_i in the XF and has weight w_i. We abstract this conjunction with the quadruple

$$(\,\min(p_1, \ldots, p_n),\, \max(p_1, \ldots, p_n),\, n,\, w_1 \times \cdots \times w_n\,)$$

That is, instead of remembering all term positions, we abstract them with the smallest interval that contains these positions and assume that these n positions are distributed uniformly along this interval. In general, a conjunction of terms is represented by the quadruple (b, e, n, w) (which stand for begin-position, end-position, number-of-points, and weight). It is assigned the score $\left(1 - \frac{e-b}{n \times \text{size}}\right) \times w$, where ‘size’ is the XF size (the total number of terms in all descendants of the XF). The first score factor is proportional to the term proximity since $(e-b)/n$ is the average distance between terms.

To make this idea work for any type of boolean connective in a search predicate, S, we associate two sets to S: one, $\mathcal{W}[\![S]\!].T$, that contains the positive quadruples and another, $\mathcal{W}[\![S]\!].F$, that contains the negative quadruples. Since a quadruple abstracts a conjunction of terms, $\mathcal{W}[\![S]\!].T$ is the disjunction of all these conjunctions. If there are no negative terms, then the score of a search predicate S is:

$$\text{score}(S) = \oplus/\{\, \tfrac{n \times \text{size}}{e-b} \times w \,[\!]\, (b, e, n, w) \in \mathcal{W}[\![S]\!].T \,\}$$

where $\{\, e \,[\!]\, \ldots \,\}$ is a set former notation, much like the tuple relational calculus, the binary associative function $\oplus$ gives the weight of a disjunction:

$$w_1 \oplus w_2 \;=\; w_1 + w_2 - w_1 \times w_2$$

and $\oplus/\{w_1, \ldots, w_n\}$ reduces the set $\{w_1, \ldots, w_n\}$ by $\oplus$ as follows:

$$\oplus/\{w_1, w_2, \ldots, w_n\} \;=\; w_1 \oplus w_2 \oplus \cdots \oplus w_n$$

That is, each cost is proportional to both the term weight and the proximity between the constituent terms and the total score is accumulated by repeatedly applying the weighting function for disjunctions.

Before we give the general score formula for any search predicate, we describe the rules for the boolean connectives. Conjunctions are formed by merging quadruples using the following operator:

$$A \bowtie \emptyset \;=\; \emptyset \bowtie A \;=\; \{\!\!\{\, (b, e, n, \tfrac{w}{2}) \;\|\; (b, e, n, w) \in A \,\}\!\!\}$$

$$A_1 \bowtie A_2 = \{\!\!\{\, (\min(b_1, b_2),\, \max(e_1, e_2),\, n_1 + n_2,\, w_1 \times w_2) \;\|\; (b_1, e_1, n_1, w_1) \in A_1 \wedge (b_2, e_2, n_2, w_2) \in A_2 \,\}\!\!\}$$

Then, the search specifications are translated as follows:

$$\mathcal{W}[\![S_1 \underline{\text{and}}\; S_2]\!].T = \mathcal{W}[\![S_1]\!].T \bowtie \mathcal{W}[\![S_2]\!].T \qquad \mathcal{W}[\![S_1 \underline{\text{and}}\; S_2]\!].F = \mathcal{W}[\![S_1]\!].F \cup \mathcal{W}[\![S_2]\!].F$$

$$\mathcal{W}[\![S_1 \underline{\text{or}}\; S_2]\!].T = \mathcal{W}[\![S_1]\!].T \cup \mathcal{W}[\![S_2]\!].T \qquad \mathcal{W}[\![S_1 \underline{\text{or}}\; S_2]\!].F = \mathcal{W}[\![S_1]\!].F \bowtie \mathcal{W}[\![S_2]\!].F$$

$$\mathcal{W}[\![\underline{\text{not}}\; S]\!].T = \mathcal{W}[\![S]\!].F \qquad \mathcal{W}[\![\underline{\text{not}}\; S]\!].F = \mathcal{W}[\![S]\!].T$$

This interpretation too preserves almost all boolean laws, as it can be seen from the following case:

$$\begin{aligned}\mathcal{W}[\![\underline{\text{not}}\; (S_1 \underline{\text{and}}\; S_2)]\!].T \;&=\; \mathcal{W}[\![S_1 \underline{\text{and}}\; S_2]\!].F \;=\; \mathcal{W}[\![S_1]\!].F \cup \mathcal{W}[\![S_2]\!].F \\ &=\; \mathcal{W}[\![\underline{\text{not}}\; S_1]\!].T \cup \mathcal{W}[\![\underline{\text{not}}\; S_2]\!].T \;=\; \mathcal{W}[\![(\underline{\text{not}}\; S_1) \underline{\text{or}}\; (\underline{\text{not}}\; S_2)]\!].T\end{aligned}$$

Furthermore, a term "T" with weight w that appears at the positions $p_1, \ldots, p_n$, for $n \geq 0$, in an XF has $\mathcal{W}[\![\text{“}T\text{”}]\!].F = \emptyset$ and:

$$\mathcal{W}[\![\text{“}T\text{”}]\!].T \;=\; \{\, (p_1, p_1, 1, w), \ldots, (p_n, p_n, 1, w) \,\}$$

In addition, a phrase "$T_1\, T_2 \ldots T_n$" has the following $\mathcal{W}[\![\text{“}T_1\, T_2 \ldots T_n\text{”}]\!].T$:

$$\begin{aligned}\{\!\!\{\, (p_1, p_n, n, w_1 \times \cdots \times w_n) \;\|\; &(p_1, p_1, 1, w_1) \in \mathcal{W}[\![\text{“}T_1\text{”}]\!].T \\ &\wedge \ldots \wedge (p_n, p_n, 1, w_n) \in \mathcal{W}[\![\text{“}T_n\text{”}]\!].T \\ &\wedge p_2 = p_1 + 1 \wedge \ldots \wedge p_n = p_{n-1} + 1 \,\}\!\!\}\end{aligned}$$

That is, terms in a phrase must have succeeding positions. Finally, the total score of any search predicate S, for a non-empty $\mathcal{W}[\![S]\!].F$, is given by:

$$\begin{aligned}\oplus/\{\!\!\{\, &\tfrac{|\min(e_1,e_2) - \max(b_1,b_2)|}{\text{size}} \times (1 - \tfrac{e_1 - b_1}{n_1 \times \text{size}}) \times w_1 \times (1 - (1 - \tfrac{e_2 - b_2}{n_2 \times \text{size}}) \times w_2) \\ &\|\; (b_1, e_1, n_1, w_1) \in \mathcal{W}[\![S]\!].T \wedge (b_2, e_2, n_2, w_2) \in \mathcal{W}[\![S]\!].F \,\}\!\!\}\end{aligned}$$

The motivation behind this formula is that negative terms should be as far as possible from positive terms. This requirement is materialized by the factor $|\min(e_1, e_2) - \max(b_1, b_2)|$, which is proportional to the distance between the positive and the negative quadruples.

In our framework, all values produced by XQuery expressions are assigned scores. This is done implicitly by the stream iterators that evaluate the query, which are described in Section 4. The unit of communication between iterators is a tuple that contains a number of elements, which typically correspond to document fragments. Each fragment is associated with a score, and these scores propagate and combined when piped through the iterators. For example, a predicate in a path expression uses the disjunction weighting function, $\oplus$, to combine the predicate scores. If the resulting score is zero, the path fragment is discarded; otherwise, its score is multiplied by the resulting predicate score. Therefore, the score of consecutive predicates in a path is the product of their constituent scores.

3 The Inverse XML Indexes

Our system uses four inverse indexes: one for XML tags, one for text terms, one for attribute names, and one for attribute values. The design of these indexes was done in a way that facilitates efficient query processing (using merge joins exclusively). Each inverse index consists of one table, called documents, which is common to all indexes, and three other tables that constitute the inverse index (the code is given in Java):

```
class InverseIndex {
    Document[] documents;
    Key[]      keys;
    Posting[]  postings;
    Hit[]      hits;  }
```

The vector documents is a binary search vector that contains the URLs of the indexed XML documents:

```
class Document { String url; }
```

Each XML document URL appears once in the documents vector. The keys vector is a binary search vector that implements the index dictionary:

```
class Key {
    String key;
    int fragments;       // how many fragments contain this term
    int total_frequency; // times term appears in all documents
    int first_posting; } // location of the first posting
```

Each term/tag must appear once in the dictionary. That is, $\forall i, j : \; i < j \Rightarrow$ keys[i].key < keys[j].key. For every XML document that contains a key in keys, there is a unique entry in postings, and all these entries appear in succeeding positions ordered by document number, starting at the first_posting position of the key:

```
class Posting {
    int document;     // document location
    int frequency;    // how many times the tag appears in document
    int first_hit; } // location of the first hit
```

That is, all postings between keys[i].first_posting and keys[$i + 1$].first_posting are associated with the key keys[i].key and are ordered by document number. Each posting, postings[i], is associated with postings[i].frequency number of hits in the hits vector, starting from postings[i].first_hit, which are ordered by the begin/position attribute. The Hit element structure depends on the type of the index. For the tag index, it is:

```
class TagHit extends Hit {
    short begin;      // the start position of term in document
    short end;        // the end position of term in document
    short level;      // depth of term in document
    short ordinal; } // ordinal of element within parent
```

while for the term index is:

```
class TermHit extends Hit {
    short position; // the start position of term in document
    short level; }  // depth of term in document
```

The iterator used for accessing the inverse indexes delivers the Posting/Hit pairs sorted by (document_number,begin/position) order, called the *index order*. That is, the major order is the document number and the minor order is the begin position of a TagHit or the position of a TermHit. This is a very crucial property because it makes possible the implementation of both path expressions and IR search specifications using merge joins without the need of sorting the input first. Furthermore, this order is actually the document order for each document. The IndexIterator for an inverse index provides the method open(key) to open the stream for accessing key hits, and the method next() to get the next Posting/Hit from the stream:

```
class IndexIterator {
    InverseIndex index;
    int ck;         // current key
    int cp;         // current posting
    int ch;         // current hit
    int max_posting;
    boolean eos;    // is this the end of stream?
    void open ( String key ) {
        ck = binary_search(index.keys,key);
        eos = !key.equals(index.keys[ck].key);
        cp = index.keys[ck].first_posting;
        ch = index.postings[cp].first_hit;
        max_posting = cp + index.keys[ck].document_number;
    }
    void next () {
        if (ch >= index.postings[cp].first_hit
                    + index.postings[cp].frequency - 1)
            if (++cp < max_posting)
                ch = index.postings[cp].first_hit;
            else eos = true;
        else ch++;
    }
    boolean eos () { return eos; } }
```

Our system populates the indexes by parsing XML documents using SAX. The text in XF elements is tokenized, stopwords are eliminated, and each token is stemmed using Porter's algorithm [16]. The SAX events startElement and endElement, and each token in text (from the SAX event characters) are assigned succeeding positions that correspond to document order. Currently, the indexes reside entirely in memory but they can be dumped to and read from binary files. We leave the implementation of indexes using B^+-trees for a future work.

4 The XQuery Processor

Our XQuery processor is stream-based, consisting of iterators that read data from input streams and deliver data to the output stream. This is done in a pull-based fashion, in which iterators are connected through pipelines and an iterator (the producer) delivers a stream element to the output only when requested by the next operator in pipeline (the consumer). To deliver one stream element to the output, the producer becomes a consumer by requesting from the previous iterator as many elements as necessary to produce a single element, etc, until the end of stream. This pipeline model is very popular in database query processing [12] because it avoids materialization of intermediate results, when possible (it is not possible for blocking operations, such as sorting and grouping).

The stream unit in our framework is a fragment retrieved from the inverse indexes. Because of the complexity of XQuery, though, XML elements may be constructed on the fly and operated upon by path expressions or other XQuery operations. It is essential, therefore, to support fragments from indexed documents as well as XML elements constructed on the fly. They are all subclasses of Element:

```
abstract class Element {
    float score; }      // relevance assessment of element
```

Then, a fragment describes an element from an indexed document:

```
class Fragment extends Element {
    int document;       // document ID
    short begin;        // the start position in document
    short end;          // the end position in document
    short level; }      // depth of term in document
```

An XML element constructed on the fly is similar to a DOM element:

```
class ConstructedElement extends Element {
    String tagname;
    Element[] sequence;       // children
    Attributes attributes; } // SAX-like attributes
class PCData extends Element { String data; }
```

The support of constructed XML elements allows us to process XML documents in their native form (not indexed) using the `document(url)` XQuery construct.

Finally, to support queries on indexed documents effectively in the case we do not have an index key to search, such as in the query `count(document()/*/*)`, which counts all indexed elements of level 2, we support patterns that match all indexed fragments within a given depth region:

```
class Pattern extends Element {
    int min_level;      // minimum depth in document
    int max_level; }    // maximum depth in document
```

In fact, the `document()` expression is translated into an iterator that produces only one element: namely, Pattern(0,0), that matches all top-level indexed ele-

ments. Later, other iterators may convert this element into a stream of Fragments, when more information is provided (such as, after a tagged projection).

Since XQuery supports FLWOR expressions with variables, all variables bindings of for-clauses are concatenated into a tuple. Recall that the values of for-clauses are always single elements, while the values of let-clauses may actually be sequences. We will see how let-variables are treated later. Thus, the unit of communication between iterators is a Tuple defined as:

```
class Tuple { Element[] components; }
```

All iterators are objects that are instances of classes that are subclasses of the class Iterator:

```
class Iterator {
    Tuple current;       // current tuple from stream
    void open ();        // open the stream iterator
    Tuple next ();       // get the next tuple from stream
    boolean eos (); }    // is this the end of stream?
```

One iterator that uses the tag index is, Child(tag,input)

```
class Child extends Iterator {
    String tag;
    Iterator input;
    IndexIterator ti;  }
```

where ti=tag_index.open(tag) is the iterator over the tag index. The body of the next() method below merges the input stream with the stream produced by the tag index. This is possible because almost all operators preserve the index order (the only exception is the order-by in FLWOR, which destroys the order and, thus, requires reordering if processed further).

```
Tuple next () {
  while (!ti.eos() && !input.eos()) {
     if (input.current[0] instanceof Fragment) {
        Fragment lf = (Fragment) input.current[0];
        Key k = ti.key();
        Posting p = ti.posting();
        TagHit h = (TagHit) ti.hit();
        if (lf.document < p.document)
            input.next();
        else if (lf.document > p.document)
            ti.next();
        else if (lf.begin < h.begin && lf.end > h.end
                 && h.level == lf.level+1) {
            current = new Tuple(new Fragment(p.document,
                                  h.begin,h.end,h.level));
            ti.next();
            return current;
        } else if (lf.begin < h.begin)
            input.next();
        else ti.next();   ...
```

This method works for non-recursive schemas only. A more general algorithm that can handle recursive schemas would require a stack of ancestors (see, for example, the stack-tree algorithm [1] and the holistic twig join algorithm [8]). For example, the XQuery `document("cs.xml")/department/gradstudent/name` is translated into the following pipeline of iterators:

```
new Child("name",new Child("gradstudent",new Child("department",
                                                    new Document("cs.xml"))))
```

where the Document iterator delivers a single ConstructedElement object, namely the top-level XML element of the document, after the entire document has been read and stored in memory.

For-loops in a FLWOR expression are evaluated using the Loop and Step iterators. For example, the FLWOR expression

```
for $d in document()/department, $s in $d/gradstudent
```

is translated into the pipeline:

```
Iterator s = new Step();
new Loop(new Child("department",new Step()),
         s,
         new Child("gradstudent",new Select(0,s)));
```

The Select iterator returns a singleton tuple containing the nth element of the tuple. It basically accesses a for-variable. Our FLWOR evaluation scheme completely avoids materialization of intermediate results. Basically, the Step iterator delivers one element only and then signals the end of stream:

```
class Step extends Iterator {
    boolean first;
    Tuple tuple = new Tuple(new Pattern(0,0));
    void open () { first = true; current = tuple; }
    Tuple next () { first = false; return current; }
    void set ( Tuple t ) { tuple = t; }
    boolean eos () { return !first; } }
```

Now, the Loop iterator, which is defined as:

```
class Loop extends Iterator {
    Iterator left;
    Step right_step;
    Iterator right; }
```

processes the left stream one tuple at a time, and sets the current tuple of the Step iterator. Then it opens the right stream and processes it one tuple at a time, up to the end of stream. Then it repeats this process for the next left tuple, etc, until the end of the left stream. More specifically, the next method of Loop is:

```
Tuple next () {
    if (!left.eos()) {
        while (right.eos()) {
            left.next();
```

```
            right_step.set(left.current);
            right.open(); };
        current = left.current.append(right.current);
        right.next();
        return current; } }
```

The same technique is used in evaluating predicates in path expressions and in FLWOR expressions. For example, the path expression:

```
document()/department/gradstudent[gpa/text()="3.5"]/name
```

is translated into:

```
Iterator s = new Step();
new Child("name",
      new Predicate(new Child("gradstudent",
                              new Child("department",new Step())),
                    s,
                    new Call("eq",new Text(new Child("gpa",s)),
                             new Constant("3.5"))))
```

where the Predicate iterator operates like Loop, by reopening and processing the condition stream for each tuple in the left stream. Two other iterators, Return and Sort, use the same stepper to evaluate an expression (the return expression of Return and the sorting values of Sort) for each input tuple.

The let-bindings in FLWOR expressions are the hardest to implement because each variable may be bound to a sequence (ie, a stream of tuples), rather than one tuple. Of course, one may materialize the bound stream into a vector, but this would be infeasible for large sequences. In our implementation, a let-variable is bound to an Iterator that delivers the sequence of tuples, but the Iterator stream is *cloned* as many times as the times the let-variable is accessed. The cloning is done using a queue with multiple pointers (one for each instance of the let-variable). The size of the queue depends on the backlog between the fastest and the slowest consumer. In some extreme cases, such as `let $v:=e return ($v,$v)`, it is the size of the entire stream (since concatenation waits for the end of the left stream before opening the right stream).

5 XQuery Translation

Figure 1 gives the rules for the translation. An XQuery e is translated into the plan $\mathcal{T}([\![e]\!], \mathbf{Step}())$, where the semantic brackets ($[\![\,]\!]$) enclose XQuery syntax (ie, they represent the abstract syntax tree associated with the syntax). Basically, $\mathcal{T}([\![e]\!], c)$ translates the XQuery syntax e into an iterator (the consumer) that receives stream data from the the iterator c (the producer). Function calls, $f(a_1, \ldots, a_n)$, include binary operations, such as $+$, $*$, and, $=$, $<$, if-then-else, etc. Element construction $< tag > \ldots < /tag >$ is equivalent to a call to element(tag, e), where e is the concatenation of the element components (using the comma operator). Function $\mathcal{F}([\![FL]\!], c, s)$ translates a sequence of for/let

Translation of XQuery expressions: e, $e_1, \ldots, e_n$

$$
\begin{aligned}
\mathcal{T}([\![\text{“text”}]\!], c) &= \mathbf{Constant}(\text{“text”}, c) \\
\mathcal{T}([\![\$v]\!], c) &= \mathbf{Select}(v, c) && \text{(a For-variable)} \\
\mathcal{T}([\![\$v]\!], c) &= \mathbf{LetVar}(v, k) && \text{(a Let-variable that uses the } k\text{th copy of the cloned Let-binding)} \\
\mathcal{T}([\![path]\!], c) &= \mathcal{P}([\![path]\!], c) \\
\mathcal{T}([\![\text{document}()]\!], c) &= \mathbf{Step}() \\
\mathcal{T}([\![\text{document}(url)]\!], c) &= \mathbf{Document}(url) \\
\mathcal{T}([\![\text{element}(tag, attrs, e)]\!], c) &= \mathbf{Element}(tag, attrs, \mathcal{T}([\![e]\!], c)) \\
\mathcal{T}([\![\text{score}(e)]\!], c) &= \mathbf{Score}(\mathcal{T}([\![e]\!], c)) \\
\mathcal{T}([\![e \sim S]\!], c) &= \mathcal{R}([\![S]\!], \mathcal{T}([\![e]\!], c)) \\
\mathcal{T}([\![f(e_1, \ldots, e_n)]\!], c) &= \mathbf{Call}(f, \mathcal{T}([\![e_1]\!], c), \ldots, \mathcal{T}([\![e_n]\!], c)) \\
\mathcal{T}([\![e_1,\ e_2]\!], c) &= \mathbf{Concatenate}(\mathcal{T}([\![e_1]\!], c), \mathcal{T}([\![e_2]\!], c)) \\
\mathcal{T}([\![\underline{\text{some}}\ \$v\ \underline{\text{in}}\ e_1\ \underline{\text{satisfies}}\ e_2]\!], c) &= \mathcal{T}([\![e_1]\!], \mathcal{T}([\![e_2]\!], c)) \\
\mathcal{T}([\![\underline{\text{every}}\ \$v\ \underline{\text{in}}\ e_1\ \underline{\text{satisfies}}\ e_2]\!], c) &= \mathcal{T}([\![\text{not}(\underline{\text{some}}\ \$v\ \underline{\text{in}}\ e_1\ \underline{\text{satisfies}}\ \text{not}(e_2))]\!], c) \\
\mathcal{T}([\![FL\ \underline{\text{where}}\ e_1\ \underline{\text{order by}}\ e_2\ \underline{\text{return}}\ e_3]\!], c) & \\
&= \left\{ \begin{array}{l}
\mathbf{Assign}(w, \mathbf{Step}(), \\
\quad \mathbf{Sort}(\mathbf{Return}(\mathbf{Predicate}(\mathcal{F}([\![FL]\!], c, \mathbf{Var}(w)), \mathbf{Var}(w), \\
\qquad \mathcal{T}([\![e_1]\!], \mathbf{Var}(w))), \\
\qquad \mathbf{Var}(w), \mathcal{T}([\![e_3]\!], \mathbf{Var}(w))) \\
\quad \mathbf{Var}(w), \mathcal{T}([\![e_2]\!], \mathbf{Var}(w))))
\end{array} \right.
\end{aligned}
$$

Translation of path expressions: *path*

$$
\begin{aligned}
\mathcal{P}([\![/A\, path]\!], c) &= \mathbf{Child}(A, \mathcal{P}([\![path]\!], c)) \\
\mathcal{P}([\![//A\, path]\!], c) &= \mathbf{Descendant}(A, \mathcal{P}([\![path]\!], c)) \\
\mathcal{P}([\![/*\ path]\!], c) &= \mathbf{Any}(\mathcal{P}([\![path]\!], c)) \\
\mathcal{P}([\![/@A\, path]\!], c) &= \mathbf{Attribute}(A, \mathcal{P}([\![path]\!], c)) \\
\mathcal{P}([\![[e]\, path]\!], c) &= \mathbf{Assign}(w, \mathbf{Step}(), \mathbf{Predicate}(\mathcal{P}([\![path]\!], c), \mathbf{Var}(w), \mathcal{T}([\![e]\!], \mathbf{Var}(w)))) \\
\mathcal{P}([\![e\, path]\!], c) &= \mathcal{T}([\![e]\!], \mathcal{P}([\![path]\!], c)) \\
\mathcal{P}([\![]\!], c) &= c
\end{aligned}
$$

Translation of FLWOR for/let bindings: *FL*

$$
\begin{aligned}
\mathcal{F}([\![\underline{\text{for}}\ \$v\ \underline{\text{in}}\ e\ FL]\!], c, s) &= \mathcal{F}([\![FL]\!], \mathcal{T}([\![e]\!], c), s) && \text{(first for-loop)} \\
\mathcal{F}([\![\underline{\text{for}}\ \$v\ \underline{\text{in}}\ e\ FL]\!], c, s) &= \mathcal{F}([\![FL]\!], \mathbf{Assign}(w, \mathbf{Step}(), \mathbf{Loop}(c, \mathbf{Var}(w), \mathcal{T}([\![e]\!], \mathbf{Var}(w)))), s) \\
\mathcal{F}([\![\underline{\text{let}}\ \$v := e\ FL]\!], c, s) &= \mathbf{Assign}(v, \mathbf{Clone}(\mathcal{T}([\![e]\!], s), n), \mathcal{F}([\![FL]\!], c, s)) && \text{(clone } n \text{ times)} \\
\mathcal{F}([\![]\!], c, s) &= c
\end{aligned}
$$

Translation of IR search predicates: S, S_1, S_2

$$
\begin{aligned}
\mathcal{R}([\![\text{“phrase”}]\!], c) &= \mathbf{Phrase}(c, \text{“phrase”}) \\
\mathcal{R}([\![S_1\ \underline{\text{and}}\ S_2]\!], c) &= \mathbf{Conjunction}(\mathcal{R}([\![S_1]\!], c), \mathcal{R}([\![S_2]\!], c)) \\
\mathcal{R}([\![\underline{\text{not}}\ S]\!], c) &= \mathbf{Negation}(\mathcal{R}([\![S]\!], c))
\end{aligned}
$$

Fig. 1. Translation from XQuery to Stream Iterators (w are fresh variables)

bindings *FL* in a FLWOR expression from left to right. The parameter *s* is the stepper used by the **Return** iterator of the FLWOR expression.

The Assign/Var operators do not correspond to any iterator. Instead, they are used for local assignments: Assign binds a local variable to an iterator and Var returns the value of the local variable.

As a simple example of translation, the following XQuery:

```
document("cs.xml")/department/gradstudent[gpa/text()="3.5"]/name
```

is translated into:

```
Child(name,Assign(0,Step(),
   Predicate(Child(gradstudent,Child(department,Document("cs.xml"))),
        Var(0),
        Assign(1,Var(0),Call(eq,Text(Child(gpa,Var(1))),Constant("3.5"))))))
```

6 Conclusion

Our language extensions are very simple, yet powerful enough to capture many emerging proposals for document-centric retrieval of XML data with relevance ranking. The XQueryRank evaluation engine is made out of pipelines of stream iterators that avoid materialization of intermediate results when possible and use efficient merge joins to evaluate path expressions and search predicates. Its modular approach of building pipelines to evaluate XQuery scales up to any query complexity because the pipes can be connected in the same way complex queries are formed from simpler ones.

The Java source of the prototype system is available at

`http://lambda.uta.edu/XQueryRank.tar.gz`

Acknowledgments: This work is supported in part by the National Science Foundation under the grant IIS-0307460.

References

1. S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural Joins. A Primitive for Efficient XML Query Pattern Matching. In ICDE 2002.
2. S. Al-Khalifa, C. Yu, and H. V. Jagadish. Querying Structured Text in an XML Database. In SIGMOD 2003, pp 4-15.
3. S. Amer-Yahia, C. Botev, and S. Shanmugasundaram. TeXQuery: A Full-Text Search Extension to XQuery. In WWW 2004.
4. S. Amer-Yahia, L. V. S. Lakshmanan, and S. Pandit. FleXPath: Flexible Structure and Full-Text Querying for XML. In SIGMOD 2004.
5. S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML Query Language. W3C Working Draft. November 2003. At `http://www.w3.org/TR/xquery/`.

6. C. Botev, S. Amer-Yahia, and J. Shanmugasundaram. A TexQuery-Based XML Full-Text Search Engine (Demo Paper). In SIGMOD 2004.
7. J. Bremer and M. Gertz. XQuery/IR: Integrating XML Document and Data Retrieval. In WebDB 2002, pp 1-6.
8. N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In SIGMOD 2002, pp 310-321.
9. T. Chinenyanga and N. Kushmerick. An Expressive and Efficient Language for XML Information Retrieval. JASIST 53(6): 438-453 (2002).
10. N. Fuhr and K. Grojohann. XIRQL: A Query Language for Information Retrieval in XML Documents. In SIGIR 2001.
11. T. Grabs and H.-J. Schek. PowerDB-XML: a Platform for Data-Centric and Document-Centric XML Processing. XSym 2003.
12. G. Graefe. Query Evaluation Techniques for Large Databases. ACM Computing Surveys 25(2): 73-170 (1993).
13. L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked Keyword Search over XML Documents. In SIGMOD 2003, pp 16-27.
14. J. Kamps, M. Marx, M. de Rijke, and B. Sigurbjornsson. Best-Match Querying from Document-Centric XML. WebDB 2004.
15. H. Meyer, I. Bruder, G. Weber, and A. Heuer. The Xircus Search Engine. 2003. At `http://www.xircus.de`.
16. M. Porter. An Algorithm for Suffix Stripping. Program 14(3):130-137 (1980).
17. A. Theobald and G. Weikum. Adding Relevance to XML. In WebDB 2000.
18. A. Theobald, G. Weikum. The Index-based XXL Search Engine for Querying. In EDBT 2002, pp. 477–495
19. A. Theobald, G. Weikum. The XXL Search Engine: Ranked Retrieval on XML Data using Indexes and Ontologies (Demo Paper). In SIGMOD 2002.
20. F. Weigel, H. Meuss, K. U. Schulz, and F. Bry. Content and Structure in Indexing and Ranking XML. WebDB 2004.
21. C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On Supporting Containment Queries in Relational Database Management Systems. In SIGMOD 2001.
22. http://www.w3.org/TR/xmlquery-full-text-use-cases/