International Journal of Scientific Research in Computer Science, Engineering and Information Technology
© 2017 IJSRCSEIT | Volume 2 | Issue 6 | ISSN : 2456-3307

Strategy List- An Efficient Querying Structure for RFX Compact Storage

Radha Senthilkumar[1], Ponsy. R. K. Sathiabhama[2], Priyaa Varshinee[3]
[1,3]Department of Information Technology, MIT Campus of Anna University, Chennai, Tamil Nadu, India
[2]Department of Computer Technology, MIT Campus, Anna University, Chennai, Tamil Nadu, India

ABSTRACT

Extensible Markup Language (XML) has been regarded as a de facto form for data exchange over the Internet. The rapid growth of XML repositories has provided the intention to design and develop systems that can store and query the XML data efficiently. The self-describing nature of XML has a great flexibility and wide acceptance but on the other hand, the huge size of the XML documents is a major drawback. The huge document size means that the amount of information that has to be transmitted, processed, and stored and queried is often larger than that of other data formats. Many researchers are working to compactly store XML document in main memory and query the same. So far researchers have implemented the compact storage structure using pointer-based approach and succinct approach for XML. Existing query processing algorithms for nested query and also querying Intra and Inter structured documents beyond simple queries are not efficient for compact storage structures. In this paper an entirely new technique is proposed which process XPath queries in the RFX (Redundancy Free Compact Storage) a non-pointer-based approach, using Strategy List. As far as know, no similar approach has been proposed for XML yet. Performance has been evaluated using the Benchmark datasets. The performance results demonstrate that the path evaluation steps for the compact storage structure are highly efficient and outperform the other tested systems.
Keywords: RFX Compact storage, strategy list, querying, XPATH classification

I. INTRODUCTION

Many markup language applications in industry, government, and academia which began as SGML [21] applications in the 1985-1998 timeframe have subsequently migrated to XML [8] using XML DTDs, W3C XML Schemas, or other schema languages. XML has emerged as a powerful format for representing data in a wide variety of fields, from technical data to finance to healthcare. Unlike traditional data formats, such as relational data, XML has a hierarchical structure that can be used to model virtually any type of data. In addition, XML is far more flexible and more tolerant to change than other formats. XML presents a number of interesting challenges and opportunities for data storage and querying. Relational databases and full-text search mechanisms that have been the backbone of many applications are not designed to manage XML content effectively. A new class of databases has emerged that is designed specifically to manage XML content. Typically called 'XML Native Databases' or just 'XML databases,' they incorporate functionality that greatly improves the searching, and manipulation and management of XML to produce the most effective XML data management solution.

The World Wide Web Consortium (W3C), the standards organization that developed XML, has also developed many standards that can be used to store, search, access and process XML data. XML databases take advantage of these standards to provide efficient and precise query, access, storage, and processing capabilities not found in traditional database technology. The result is that applications using XML databases are more efficient and better suited for managing XML data. XMill [10], XGRIND [13], XQuec [1], XQZip [3], XBW [18] are some of the XML storage systems for XML data each with their own merits and demerits. Redundancy Free Compact

XML (RFX) is one such storage or more precisely, a compression system for XML proposed by [14]. Some of the key features of RFX compact storage are its redundancy elimination and support for one to many relationships in XML documents.

1.1. One-to-many relationships in XML documents

There are three types of document relationships defined for XML documents, and any XML document falls under one of these three categories. The three types of document relationships [5] are as follows- a) Containment Relationship b) Intra Relationship document c) Inter Relationship document. Querying an XML storage system and returning results with minimum time complexity, is one of the most happening topics in which many researches are working on. RFX Compact Storage is a storage system for storing XML documents and, mainly intended to develop a unique technique for query optimization and evaluation in this system.

1.2. Why a Unique Technique is needed for Query Optimization in RFX Compact Storage

RFX Compact Storage combines the advantages of both relational and hierarchical data models. If this "self optimized" nature of RFX Compact Storage is exploited, then query evaluation time of the XPATH query can be effectively reduced. This necessitates a unique technique that leverages the advantages of RFX Compact Storage. RFX Compact Storage makes it possible to store the topology and meta data of the XML document in the main memory itself. The topology layer of RFX stores the order information of the XML Document using a novel approach. XML document has different levels of nesting and can be modeled as a k-ary tree. RFX tends to store the order information of this k ary tree in theoretic minimum of 2n bits by which it is evident that the topology is available in main memory. Zhang [11] provided a succinct approach using balanced parenthesis encoding to store blocks of data. Similar to balanced parenthesis encoding, RFX uses a novel approach to encode the order of the XML document rather than nesting information called as order encoding [14].

1.3. Organization of the article

The remainder of the paper is organized as follows: Section 2 provides an overview of related work. Section 3 states the proposed optimization technique derived from the strategy list. Section 4 describes the Querying methodology. Section 5 presents the results of experimental evaluation of proposed approach. The paper is concluded in Section 6.

II. RELATED WORKS

XMill [10], the first compressed and non-queriable XML scheme was invented to provide a means for compact storage. Although XMill achieves a good compression ratio, it does not support querying nor updating. XGRIND [13], uses top down query evaluation strategy using SAX [19] parser, which makes a depth-first-search traversal of the XML document. It maintains information about its current location in the XML document and the contents of the set of XML nodes that it is currently processing. SAX parser supports exact-match or prefix-match queries, where the query path and the query predicate are converted to the compressed form. During parsing of the compressed XML document, when the parser detects that the current path matches the query path, and that the compressed data value matches the compressed query predicate, it outputs the matched XML fragment. For range or partial-match queries, only the query path is compressed. While parsing the compressed XML document, when the parser detects that the current path matches the query path, the associated data value is decompressed and used for evaluating the match. Compared to XMill, XGRIND has a lower compression ratio but supports these types of queries.

The XPRESS [9] system is better over XQZIP [3] in terms of decompression times but XGRIND is relatively lagging behind both XPRESS and XQZIP. It supports Exact-match, Prefix-match, XPath Axes: Child and descendant attribute types of node search and navigation. Both XGRIND and XPRESS require top-down query evaluation, and do not support set-based query evaluation such as structural joins. XGRIND and XPRESS are homomorphic compressors where both support direct querying of compressed data by retaining the document structure (tags and data values separately). XPRESS separates the context and tag and both are coded respectively, then the two parts are assembled after encoding. While XGRIND uses dictionary encoding and Huffman encoding for tags and data, XPRESS adopts reverse arithmetic encoding

which maps the entire path expression to intervals for tags and diverse encoding methods for text according to the data types. The encoding technique enables XPRESS to achieve better compression ratios and higher query performance than XGRIND. However, these methods do not support the evaluation of multi-conditional queries over compressed documents.

XQZIP[3] uses several operators like select and project where first step is converting XQuery[24] expression to an internal Nested Relational Sequence similar to PAT internal form corresponding to Ozsu[7]). Queries having multiple and deeply nested predicates with mixed structure-based, value-based, and aggregation conditions done here is not supported by XGRIND. It supports a wide range of query types namely recursive path expression, aggregation, ancestor-descendant, conditional queries, conditional with recursive paths and further combinations of the same. The number of nesting levels does not affect the running time of the queries used in the experiments. This is because the structure of the XML document stored as an in-memory tree saves the time for looking up the corresponding indexes from the database. It follows both top-down and bottom-up query evaluation strategies. The use of the SIT (Structure Index Tree) in XQzip, minimizes the tree edges while the SIT, indexes the tree nodes and does not compress the textual XML data items and hence it saves space.

XQueC[1] proposed by Andrei Arion , compresses each data item individually and this usually results in a lower compression ratio (compared to XMill). An important feature of XQueC is that it supports efficient evaluation of XQuery by using a variety of structure information and other indexes. However, these structures together with the pointers pointing to the individually compressed data items, incurs huge space overhead. The query processor evaluates XQuery queries over compressed documents. The complete set of operators of XQuery allows for efficient evaluation over the compressed repository. Compression and Querying System (XCQ)[12], is a compression method which separates structure from data. It was developed based on a technique called DTD Tree and SAX Event Stream Parsing (DSP). The tree structure is compressed using the DTD information and the text is compressed using a standard method like gzip. The compressed documents in XCQ adopt a partitioned path-based data grouping which supports evaluating queries without running a full decompression. XCQ also supports querying over partially decompressed documents. Queries involving only the structure of the document can be answered without decompressing the data Stream. It has a better compression ratio than XMill at the expense of compression time, if no partitioning is made on the data streams in XCQ. XCQ also achieves, a better compression ratio and compression time than XGRIND.

Further YFILTER[4] aims to provide fast, matching of XML encoded data and transformation of the matched XML data based on specific requirements. Two extended query types are implemented here. The first type of query done in YFILTER is selection query where the predicates can be Exact Matching (equality comparison) or Range matching (inequality comparison involved).The other type is aggregation type of queries where sum, average types of functions can be used for computation. This system has no compact storage system operations involved. Xcpaqs[20] is an XML compressor with path query support .It is a hybrid compressor which separates structure and context information from XML document. It also keeps homomorphism relation between compressed and original XML document. Xcpaqs encodes path and tag respectively. But more complex operators such as aggregation and join across objects on XML document are not supported.

In [14], the author had proposed the basic structure for RFX Compact Storage and querying. [17] enhanced this storage structure to perform Intra and Inter document querying unambiguously. Query optimization techniques for Intra [15] and Inter relational [16] XML documents have been already proposed. [17] had developed a strategy to optimize any complex XPATH query that ends up in an efficient query plan for the Compact Storage which applies for all types of document relationships. Matthias Brantner [2] proposed kappa join operator to optimize a correlated query. His approach works when one query is dependent and another is independent, whereas optimization technique in [17] works even when both queries are dependent on the external producer. However, [17] does not cover exhaustive XPATH query classification and has been designed mainly for nested XPATH queries. In this paper, the efficiency of strategy list for queries are explained. Also this paper proposes a novel XPATH classification which to the best of our knowledge is exhaustive.

III. TYPES OF QUERY

A XPATH[22] query navigates within a XML document and retrieves information. It is a language for selecting nodes from a XML document. XPath 1.0 is widely implemented and used, either on its own (called via an API from languages such as Java, C# or JavaScript), or embedded in languages such as XSLT or XForms and was recommended in 1999. XPath 2.0(current version), was Recommended in 2007. A number of implementations exist but are not as widely used as XPath 1.0. Classifying XPATH queries provides valuable scope for testing the performance of query processor in different workloads. A list of XPATH functionalities have been selected which to the best of our knowledge are complete. Tabulated the functionalities to demonstrate how those functionalities are covered in different XPATH queries. The functionalities have been derived from XPATH use cases [23].

3.1 XPATH functionalities

Single phrase

These are the simplest queries which are a single word or sequence of words.

Eg. //title , /books/book/title

Axes and Predicate

An axes is either a forward axes or a reverse axes. An axes that contains the context node or nodes that are after the context node in the document order is a forward axes. An axes contains the context node or nodes that are before the context node in the document order is a reverse axes. A predicate filters is a node-set with respect to an axes to produce a new node-set. For each node in the node-set to be filtered, the Predicate Expression is evaluated with that node as the context node.

Eg., para[position()=3]. => PREDICATE

Descendant, parent, following-sibling, preceding-sibling => AXES

Query Across the Content

This functionality is used to query across XML element boundaries. These Boundaries include XML tags: Start-Tags, End-Tags, and Empty-Element Tags. Descendant XML tags and attribute values are removed from the string to be queried by tokenization before the query. At the XQuery Data Model level tags are a syntactic element.

Eg., Find all book chapters containing the phrase "one of the best known lists of heuristics is Ten Usability Heuristics".

/books/book[count(.//chapter ftcontains "one of the best known lists of heuristics is Ten Usability Heuristics")>0]

Wildcard matching

This functionality illustrates queries which use wildcards to substitute for any other character or sequence of characters to a word or a part of a word..

Eg. Find all books with the word "test" with a one character suffix in the text.

/books/book[count(./content ftcontains "test." with wildcards)>0]

Prefix-Infix-Suffix matching

Character wildcards may be prefix (appended before the first character), infix (inserted into a word), or suffix (appended after the last character).

Eg.(suffix query) Find all books with the word "test" with a three to four character suffix in the text.

/books/book[count(./content ftcontains "test.{3,4}" with wildcards)>0]/(@number|./content)

Ordered (Query)

Distance

Ordered distance queries finds sequences of words allowing up to a specified number of intervening words.

Eg. Find all books with information on "software developers". /books/book[count(.//content ftcontains "software" ftand "developer" with stemming distance at most 3 words)>0]

Window

This functionality enables ordered queries to search within a window, within a sentence or within a paragraph.

Eg. Find all books about "users feeling well-served". /books/book[count(.//content ftcontains "users" ftand

"feeling" ftand ("well served" ftor "well-served") with stemming ordered window 15 words)>0]

Unordered (Query)

Distance

These are same as ordered distance queries except that the words need not be in the specified sequence.
Eg. Return all books, listing books with text on "software" first. /books/book[count(.//content ftcontains "software" ftand "developer" with stemming distance at most 3 words)>0]

Window

Similar to ordered window queries , unordered window queries enable unordered queries to search within a window, sentence or within a paragraph.
Eg Find all books about users feeling “well served” /books/book[count(.//content ftcontains "users" ftand "feeling" ftand ("well served" ftor "well-served") with stemming unordered window 15 words)>0]

Stemming

This functionality invoke a stemming algorithm which returns noun, verb, adjective, and adverb forms of a word or root of a word in singular and plural.
Eg. Find all books with the word "test" in the text.
/books/book[count(.//content ftcontains "test" with stemming)>0]

Thesaurus

This functionality illustrates queries which return synonyms or related words identified by thesauri, dictionaries, and taxonomies.
Eg. Find all introductions which quote someone.
/books/book[count(.//introduction ftcontains "quote" with thesaurus at http://bstore1.example.com/UsabilityThesaurus.xml" relationship "synonyms")>0]

Stop word

These use cases query a phrase, one word of which has been identified as a stop word via a stop word list.
Eg. Find all books with the phrase "planning then conducting" in the text where "then" is treated as a stop word./books/book[count(.//content ftcontains "planning then conducting" with stop words at "http://bstore1.example.com/StopWordList.xml")>0]

Diacritics

The main use of diacritics is to change the sound value of the letter to which they are added. XPATH implements this functionality to differentiate.
Eg. Verify the existence of a "résumé" in the papers of John Wesley Usabilityguy.
doc("http://bstore1.example.com/full-text.xml")/books/book[count(.//content ftcontains "résumé." with wildcards diacritics sensitive)>0]

Aggregation

An aggregation operation computes a single value from a collection of values. An example of an aggregation operation is calculating the average daily temperature from a month's worth of daily temperature values.
Eg. Find all book titles containing the word "usability".
/books/book[count(./metadata/subjects/subject ftcontains "web site" ftand "usability")>0]

Logics

This functionality includes queries containing logical expressions: or, and, the unary not, and not, etc
Eg Find all books which do not belong in a collection on "usability testing".
/books/book[count(. ftcontains ftnot "us.* testing" with wildcards)>0]

Quantification

There are two types of quantification- existential and universal quantification. Existential quantification is used to find a word and a phrase in any instance of an element across the siblings of the same element. Universal quantification finds two words in every instance of an element.
Eg. Existential quantification: Find all books with the phrase "web site" and the word "usability" in any subject.
/books/book(some $s1 in ./metadata/subjects/subject satisfies ./metadata/subjects/subject ftcontains "web site") and (some $s2 in ./metadata/subjects/subject satisfies ./metadata/subjects/subject ftcontains "usability")
Eg. Universal quantification:
/books/book(every $pub in .//publisher satisfies ($pub ftcontains “ersatz” ftand “publications”))

Multi Lingual

As implied by the name, this functionality enables multilingual content in the XML document to be queried.

Eg. Find all book subjects containing the phrase (n-gram) "网站".

/books/book/metadata/subjects/subject[. ftcontains "网站" language "zh"]

3.2. XPATH Functionalities Spreadsheet

The table1 provides insight to variety of queries by the number and combination of functionalities they cover. Some functionality always occurs in conjunction with one more functionality. For example prefix matching always occurs with wild card functionality. Such a relationship can be specified as "functionality1 *is dependent* on functionality2".

Table 1. Coverage of XPATH functionalities in Queries

S No		1	2	3	4	5	6		7		8	9	10	11	12	13	14	15
							a	b	a	b								
1		-	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
2		✔	-	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
3		✖	✔	-	✔	✔	✔		✔	✔	✔				✔			
4		✖	✔	✔	-	✔					✔	✔						
5		✖	✔	✔	✖	-										✔	✔	
6	a	✖	✔	✔			-	✔							✔	✔	✔	
	b	✖	✔	✔			✔	-							✔	✔	✔	
7	a	✖	✔	✔					-	✔					✔	✔		
	b	✖	✔	✔					✔	-					✔	✔		
8		✖	✔	✔	✔		✔	✔	✔	✔	-	✔						
9		✖	✔						✔	✔	✔	-		✔	✔	✔		
10		✖	✔										-		✔	✔		
11		✖	✔									✖		-				
12		✖	✔	✔		✔	✔	✔	✔	✔			✔		-	✔		
13		✖	✔			✔	✔	✔	✔	✔		✔	✔	✔	✔	-	✔	
14		✖	✔		✔	✔	✔	✔		✔			✔	✔	✔	✖	-	
15		✖	✔													✔		-

✖ - is dependent on ✔ – Possible combination

Query Examples

Query 1

Functionalities: phrase query, stemming, unordered distance (0 to 2 intervening words)

Abstract Query: Find all books with "improve" "web" "usability" in the short title.

XPATH:

/books/book[count(./metadata/title/@shortTitle ftcontains "improve" ftand "web" ftand "usability" with stemming distance at most 2 words)>0]/metadata/title

Query 2

Functionalities: phrase query, wildcard (suffix) (1)

Abstract Query: Find all books with the word "test" with a one character suffix in the text.

XPATH: /books/book[count(./content ftcontains "test." with wildcards)>0]

Query 3

Functionalities: phrase query, and query, existential quantification

Abstract Query: Find all books with the phrase "web site" and the word "usability" in any subject.

XPATH:/books/book(some $s1 in./metadata/subjects/subject satisfies ./metadata/subjects/subject ftcontains "web site") and (some $s2 in ./metadata/subjects/subject satisfies ./metadata/subjects/subject ftcontains "usability")

Query 4

Functionalities: phrase queries, stemming, and query

Abstract Query: Find all books with the phrase "manuscript guides" in the short title and the phrase "user profiling" in a component title

XPATH: none

IV. STRATEGY LIST

Query processor is the most important component of any storage system. Without the ability to retrieve the stored data, any storage strategy is despicable. Also, efficiency in terms of time followed by space has to be considered while querying. In this paper, novel data structure called Strategy List to support efficient

querying in RFX Compact Storage is proposed. Strategy List is based on divide and conquer approach. The basic approach, architecture, supported functions and design of the strategy list are described in the forthcoming subsections.

4.1 Divide and Conquer

XML query languages like XQuery and XPATH provide rich set of features to retrieve information from XML documents. Each XML storage system has its own querying methodology to implement querying in the same. RFX storage system is one such system wherein we implement strategy list querying and discuss its efficiency with respect to existing methodologies in other storage systems. The basic idea behind strategy list is "divide and conquer". As a matter of fact, any complex query is built with more than one simple query. Hence, the complex query can be broken down into simple queries, which take less time to execute. Two possible forms of simple queries is shown in figure 1a and 1b.

([/ | //]a([condition])*)+
book[/bookstore/@specialty=@style]
**[@specialty]*
book[@style]
author[last-name = "Bob"]
author[degree and award]

Figure 1a. Simple query-form1

([/ | //]a)+
/students/student/name
/dblp//author/first name
bookstore//book/excerpt//emph
price/@exchange
book//last-name*

Figure 1b. Simple query-form2

Where, 'a' is any element node of the XML document.

The strategy is to break any complex query into more than one simple query in one of the above stated forms and order the functionalities involved in the query. This strategy particularly applies for existentially quantified queries where the XPATH query is correlated [kappa join], though it works well with other query types too. Here the effectiveness of our strategy with some examples are described. Query 1 has been selected from [XPATH usecases].

Query 1: /books/book[count(.//content ftcontains "users" ftand "feeling" ftand ("well served" ftor "well-served") with stemming ordered window 15 words)>0]
This query selects those books whose content contains "users feeling well-served (or) well served" within a window of 15 words. The overhead in this query is due to the join operations that occur at conjunctions and disjunctions. This query includes both independent and guide operations.

Definition 1: Guide operations are those operations which cannot be performed independently and are always performed in conjunction with some other independent operation.

Definition 2: Independent Operations are those operations which can be executed independently irrespective of the occurrence of some other operation in the query.
For example, "stemming" is a guide operation guiding the independent operation " ftcontains".
The independent operations involved in Query 1 includes ftor(OR), ftand(AND), ftcontains(CONTAINS) and count. The guide operations include stemming and ordered window.
This query can be executed efficiently in RFX by retrieving all the positions of the result node without condition and then filtering the positions step by step. The independent operations specified sequentially are applied on the results of previous step. The general querying method in RFX for simple queries is described in section 4.2.

The following is the list of steps after query 1 has been broken.

1. /books/book
2. .//content
3. ftcontains "users"
4. ftand feeling
5. ftand well-served/well served
6. count

7. (check)result>0

GUIDE OPERATIONS for steps 3,4,5 – stemming, ordered window 15 words.
Once the order of operations has been known, the operations are queued according to the order in the strategy list and then executed.

Query 2 : Nested Query
//restaurants//food[/price>
//standards/food/price]/availability

The above is a correlated query [kappa join]. While /price is the dependent query (dependent on the path //restaurants//food) "//standards/food/price" is the independent query (because it is independent of the preceding path in the XPATH expression"//restaurants//food"). However, in usual query execution for each price value in the dependent query, the independent query is executed once which is not necessary.

Solving the independent and dependent queries separately is the scope for efficient execution in this query. Strategy list does this with the aid of node queues which will be described shortly.

After the query is broken, the execution steps contain queries

1. S=//standards/food/price
2. T=//restaurants//food/price
3. R= S intersection T
4. A=//availability
5. Select those positions of availability (A) which are nearer to positions in R

It should be clear by now, how divide and conquer approach applied to XPATH querying, improves query efficiency without any optimization explicitly being employed.

Query 3 :/books/book
This query is evaluated in RFX as follows
1. ID of the books is fetched from Element Table
2. ID of the 'book' is found from ET(Element Table)
3. Parent Id of the 'book' is verified
4. Since book is the last node of the query, the location of 980ccurrences of 'id' of 'book' in element structure mapping is noted.
5. Filter the locations that which belong to element alone using order encoding.(A data may have same id as 'book'. The locations corresponding to data id should be omitted).
6. Select all element, attribute , text nodes whose level is greater than book node (those which come under book node).

```
<xml>
 <books>
  <book isbn="118970">
   <name>Geetanjali</name>
   <author>Rabindranath Tagore</author>
   <price>Rs 2000</price>
  </book>
  <book isbn="112345">
  <name>Life Divine</name>
  <auhtor>Sri Aurobindo</author>
  <price>Rs400</price>
  </book>
 </books>
</xml>
```

Figure 2. Sample XML document

```
OUTPUT for Query 3

<book isbn="118970">
<name>Geetanjali</name>
<author>Rabindranath Tagore</author>
<price>Rs 2000</price>
</book>
<book isbn="112345">
<name>Life Divine</name>
<auhtor>Sri Aurobindo</author>
<price>Rs400</price>
</book>
```

```
OUTPUT for Query 4

<book isbn="112345">
<name>Life Divine</name>
<auhtor>Sri
Aurobindo</author>
<price>Rs400</price>
</book>
```

Figure 3. Output for Query4 and Query5

Query 4 : //book[@isbn=112345]
This query is evaluated in RFX as follows
1. ID of the book is fetched from ET*
2. ID of 'isbn' is fetched from AT* and is checked if it belongs to the book element.
3. ID of the data '112345' is found from ADT*
4. Check if '112345' belongs to 'isbn' by seeing if the attribute id of the data '112345' and the id of 'isbn' are one and the same.
5. If they are same search for the location of id '112345' in the element structure mapping
6. Find the location of book id in element structure mapping that which is nearest to location of '112345'.
7. Select all element, attribute , text nodes whose level is greater than the selected book node (those which come under the selected book node).
Results from the sample XML document in Figure 2 are given in Figure 3

4.2. Query Processing Architecture in RFX Compact storage

Query processing architecture for RFX is given in Figure 4. The query analyzer analyzes the input XPATH query and validates its syntax. If the query is syntactically valid, the query is unnested, semantic validation is done and the query is split up according to the operator precedence table in Appendix B. Then for each sub query, a node queue is created and the operations in the sub query are queued. The list is processed and the results are returned. Each node queue corresponds to a particular sub query, which is a simple query. Example for query split up and ordering is given below.

4.3. General Querying In RFX Compact Storage

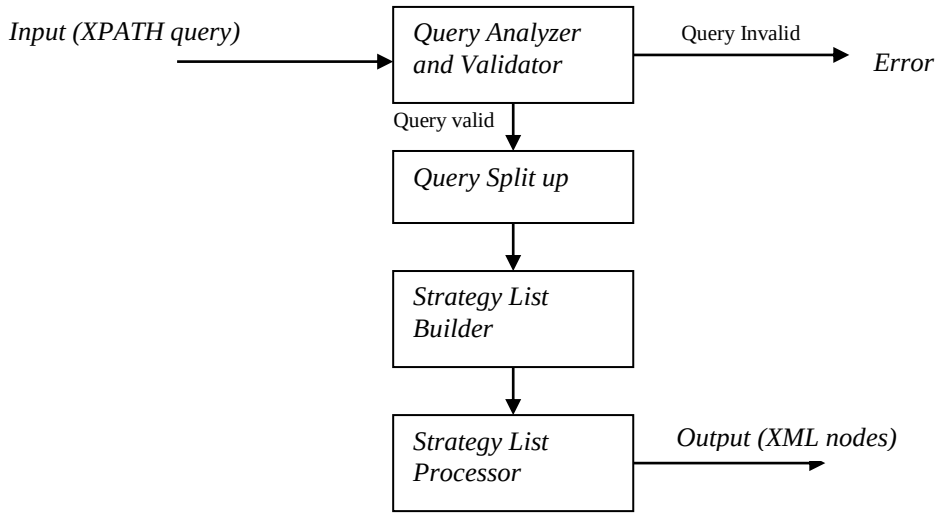


Figure 4. Query Processing Architecture in RFX

Query 5 : /books/book[count(.//content ftcontains "users" ftand "feeling" ftand ("well served" ftor "well-served"))>0]

Split Up phase

After Step 1: Split point- '['

1. $result = $result1[$result2]
2. $result1=/books/book
3. $result2= count(.//content ftcontains "users" ftand "feeling" ftand ("well served" ftor "well-served"))>0

After Step 2: Split point – '('

1. $result = $result1[$result2]
2. $result1=/books/book
3. $result2= count($result3)>0
4. $result3= .//content ftcontains "users" ftand "feeling" ftand ($result4)
5. $result4="well served" ftor "well-served"

After Step 3: Split point – 'ftcontains'

1. $result = $result1[$result2]
2. $result1=/books/book
3. $result2= count($result3)>0
4. $result3= .//content $result5 ftand "feeling" ftand ($result4)
5. $result4="well served" ftor "well-served"
6. $result5= ftcontains "users"

After Step 4: Split point – 'ftand'

1. $result = $result1[$result2]
2. $result1=/books/book
3. $result2= count($result3)>0
4. $result3= .//content $result5 $result6 $result7
5. $result4="well served" ftor "well-served"
6. $result5= ftcontains "users"
7. $result6= ftand "feeling"
8. $result7= ftand ($result4)

After Step 5: Split point – 'ftor'

1. $result = $result1[$result2]
2. $result1=/books/book
3. $result2= count($result3)>0
4. $result3= .//content $result5 $result6 $result7

5. $result4="well served" $result8
6. $result5= ftcontains "users"
7. $result6= ftand "feeling"
8. $result7= ftand ($result4)
9. $result8= ftor "well-served"

The algorithm for Query Analysis and Query split up is given in figure 5a and 5b.

Algorithm 1: sub_query_list_Analysis (Query)

4. Read I = Query
5. if (query valid)
   a. return error
6. if (I is correlated)
   a. then for each $q = independent query in I
      i. L= empty set
      ii. O_i = find_step($q)
      iii. L = L U O_i
   b. return L
7. else
   a. return(find_steps($q))

endif

Algorithm 2: ordered_steps find sub_query_list_Analysis (Query)

1. Read I = Query
2. if (query valid)
   a. return error
3. if (I is correlated)
   a. then for each $q = independent query in I
      i. L= empty set
      ii.

Algorithm 2 : ***ordered_steps*** **find_steps(*query*)**

1. if there are more operators in *query* && visited(operator)=false
   a. if(o=highest_precedence_operator && not_visited)
      i. $split_point=o;
      ii. visited(o)=true;
   b. split the query at each $split_point into two queries
2. if the $query_part after splitting does not have any unvisited operator
   a. Add the $query_part at the end of operation sequence
3. Repeat step 1 till operators are visited.
4. return operation_sequence

Figure 5a. Algorithm for analysis phase

4.4 Stepwise Optimization and Evaluation Using Strategy List

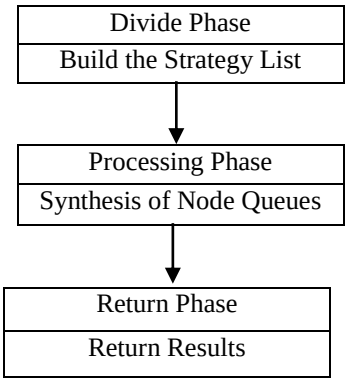


Figure 6. Query Phases

During the "divide phase", the query is broken down and the operation sequence is found to build the strategy list. Then each node queue is synthesized and results are combined according to the linkers to find the final query result. The query phases have been given in order in Figure 6.

4.5. Strategy List Structure

Divide and conquer approach of querying is implemented with Strategy List. Strategy List is a heterogeneous list containing node queues and linkers. Each node queue contains in it a simple query or a sequence of operations with a set of guide operations. As described in section 4.1., the sequence of operations which are ordered, speed up the query execution. Also all the node queues are independent of each other and can be run in parallel, which is the most highlighting feature of the Strategy List.

4.5.1. Node Queue

The node queue is a queue in which each element represents an independent operation. For example if an element of the node queue is /book, then the operation is to select all book nodes from the current node. If the element in node queue is "ftcontains "user"", then the operation is to select only those nodes with the term "user" from current result nodes. A query may be composed of one to many node queues depending upon its structure. A nested or correlated query contains alteast two node queues, one for dependent and the other for independent query.

4.5.2. Linkers

Linkers connect two node queues. They determine the relationship between the node queues they connect. The following are the five types of linker nodes which are self explanatory.

1. Correlation_starts
2. Correlation_ends
3. Conjunction
4. Disjunction
5. Query_ends

The linkers Correlation_starts and Correlation_ends are used when the query is correlated or nested. These linkers are used to connect node queues of dependent and independent sub-queries. Conjunction and disjunction are used for conjunctive nested queries like *//student[examination/@id= //exam[grade < 'B']/@id* ***and*** *@id* = /lecture[title='NCT']/helpers/helper/@student]/name and disjunctive nested queries like //student[examination/@id=//exam[grade< 'B']/@id **or** @id = /lecture[title='NCT']/helpers/helper/@student]/name. Query_ends marks the end of execution

4.6. Query Evaluation

Examples for strategy list building

Query6
/books/book[count(./metadata/title/@shortTitle ftcontains "improve" ftand "web" ftand "usability" with stemming distance at most 2 words) > 0]/metadata/title

This query retrieves all books with words "improve", "usability", and "web". The independent operations in the query are 'ftcontains', ' ftand'. Guiding operations are 'stemming' , 'distance 2 words'. Strategy List for Query6 is given in figure 7

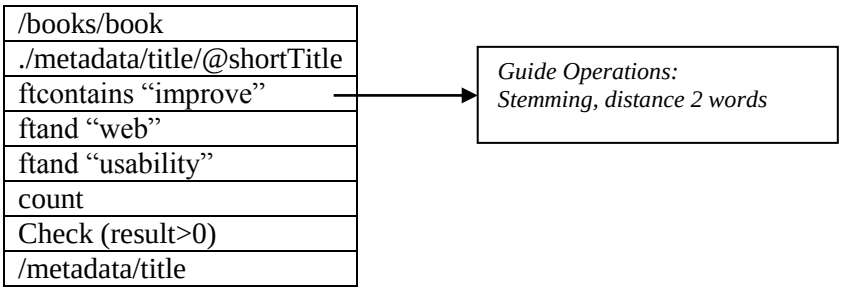


Figure 7. Strategy list for Query4

In case of Query6, the query contains only a single node queue. The guide operations which apply to fcontain also apply to subsequent ftand and ftor.

Query7
/books/book(some $1 in ./metadata/subjects/subject satisfies ./metadata/subjects/subject ftcontains "web site") and (some $2 in ./metadata/subjects/subject satisfies ./metadata/subjects/subject ftcontains "usability")

This query finds all books with "web site" and "usability" in any subject. Clearly this is an existential quantification query. Strategy list for the same is given below.

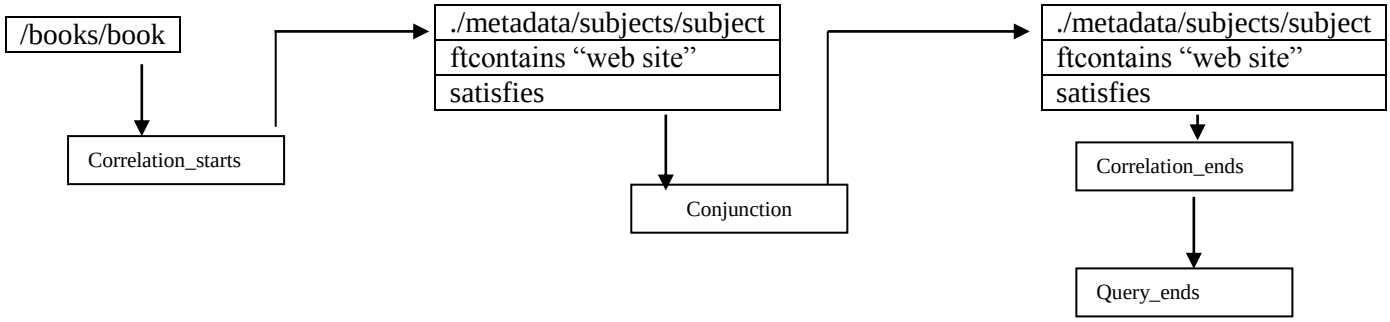


Figure 8. Strategy List for Query 7

The main context of the query lies in the node queue before the linker "correlation_starts". After the linker "correlation_ends", no node queues are there. Hence, the resultant nodes will be 'book' nodes of XML document. The strategy List for ~~for~~ Query 7 is shown in Figure 8.

V. EXPERIMENTAL RESULTS

We had implemented our approach in JAVA. All of the experiments were performed on a PC with Pentium4, 3 GHz CPU, 1GB memory and 80 GB hard disk. The Operating System is Windows XP.

5.1. Execution Time

The first metric we chose for performance evaluation is query execution time which includes the time to build the strategy list followed by scalability. We had compared the querying time with XQZip+ and *XGRIND* . As XQZip is not an open source software, we had adopted the values of its query execution time from [XQzip].

The test queries has been listed in Appendix A. The set of queries that have been selected are from [XQZip]. The test queries cover a wide spectrum of queries, ranging from simple to complex join queries. As XQZip does not support complex join and order based predicates, Q4 and Q5 of XMark fail for the same.

The test queries are run on various benchmarks including

XMark[1] the XMark documents model an auction database with deeply-nested elements. The XML document instances of the XMark benchmark are produced by the *xmlgen* tool of the XML benchmark project. For our experiments, we generated three XML documents using three increasing scaling factors.

DBLP[2] presents the famous database of bibliographic information of computer science journals and conference proceedings.

Lineitem[3] is an XML representation of the transactional relational database benchmark (TPC-H).

Shakespeare[4] represents the gathering of a collection of marked-up Shakespeare plays into a single XML file. It contains many long textual passages.

Treebank[5] is a large collection of parsed English sentences from the Wall Street Journal. It has a very deep, non-regular and recursive structure.

Characteristics of listed datasets have been given in table 2

Table 2. Benchmark datasets and their chracteristics

Data Source	Size (MB)	Depth	Tags/Attrs	E_num	A_num
XMark	111	11	86	1666315	381878
DBLP	148	6	41	3883112	471124
Treebank	82	36	252	2437666	1
Shakespeare	7.3	6	23	179072	0
Lineitem	30.8	3	19	1022976	1

Query Execution Time (in seconds) for XQZip+, RFX and *XGRIND* is given in table 3. The cases where the query fails have been left blank.

Table 3. Query Execution Time for XQZip+, RFX and XGRIND

Dataset	Query	XQzip+	RFX-List	XGRIND
Shakespeare	Q1	0.014	0.0036	1.311
	Q2	0.016	0.0039	1.62
	Q3	0.016	0.0038	2.312
	Q4	0.005	0.0016	
	Q5	0.014	0.0041	
LineItem	Q1	0.011	0.003	2.336
	Q2	0.012	0.004	2.89
	Q3	0.014	0.0045	3.21
	Q4	0.032	0.015	
	Q5	0.007	0.0045	
Treebank	Q1	0.674	0.235	
	Q2	0.177	0.076	
	Q3	0.453	0.159	
	Q4	1.003	0.785	
	Q5	0.985	0.491	
XMark	Q1	0.349	0.234	35.012
	Q2	0.118	0.068	
	Q3	1.544	1.266	
	Q4		2.547	
	Q5		4.938	
DBLP	Q1	0.034	0.015	19.582
	Q2	0.029	0.009	26.108
	Q3	1.543	0.564	50.344
	Q4	0.001	0.001	
	Q5	0.642	0.546	

[1]http://monetdb.cwi.nl/xml/
[2]http://kdl.cs.umass.edu/data/dblp/dblp-info.html
[3]http://www.cs.washington.edu/research/xmldatasets/
[4]http://www.ibiblio.org/xml/examples/shakespeare/
[5]http://www.cis.upenn.edu/~treebank/

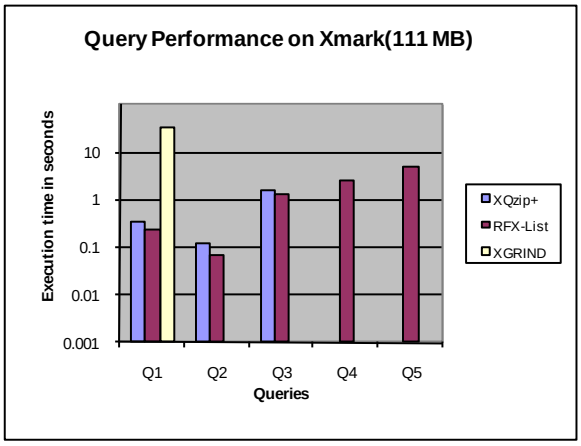


Figure 9a. Query Performance on XMark

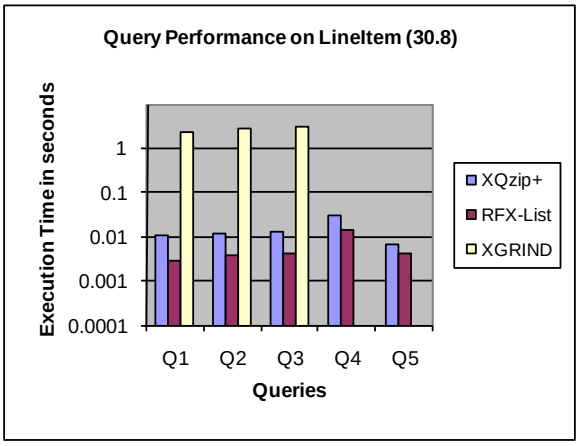
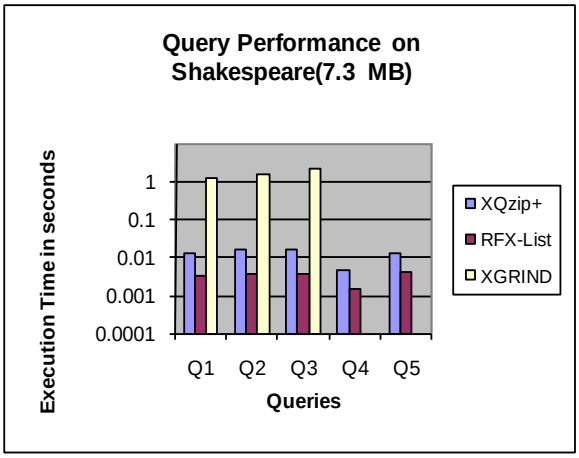


Figure9b. Query Performance on Lineitem



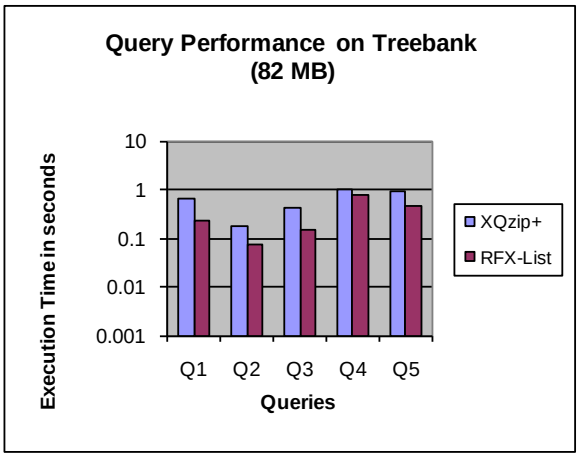
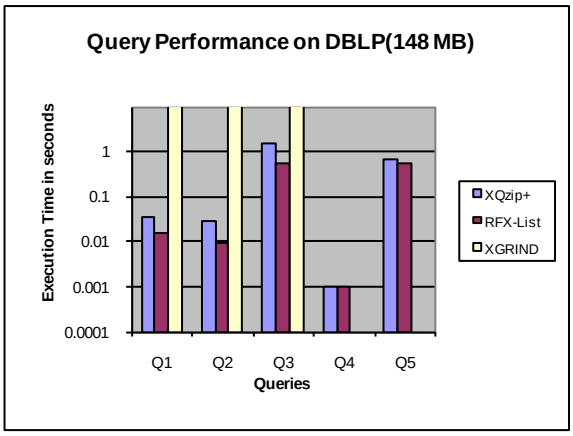


Figure9c. Query Performance on Shakespeare

Figure9d. Query Performance on Treebank



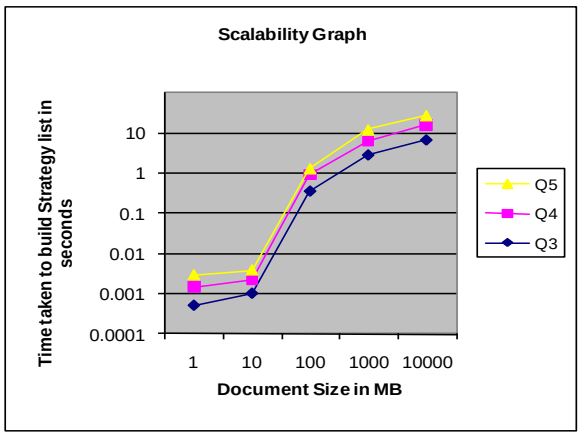


Figure9e. Query Performance on DBLP

Figure10. Scalability graph

The graphs in Figure 9a, 9b, 9c, 9d, and 9e illustrate the query performance of Strategy List in RFX compact storage for each data set. The corresponding data values have been tabulated in Table 3. Q4 and Q5 are join queries which fail in both XQZip and XGRIND.

5.2. Scalability

The building time of strategy list for XMark Q3, Q4 and Q5 has been recorded for various document sizes. The resulting scalability graph is shown in Figure 10. The figure clearly portrays the scalability of our approach. The values of list building time has been listed in table 4

Table4. Time taken to build strategy list on various document sizes

Doc size (MB) / **Query**	**1**	**10**	**100**	**1000**	**10000**
Q3	0.0005	0.001	0.361	2.9	6.89
Q4	0.001	0.0014	0.588	3.56	9.63
Q5	0.0015	0.0016	0.369	5.79	11.23

VI. CONCLUSION AND FUTURE WORK

In this paper, we had designed a unique data structure to query RFX Compact Storage. Our proposed querying scheme has scored well in terms of execution time as well as scalability. One of the main advantages of strategy list is the parallel execution of query parts (node queues).In contrast to [17] which mainly concentrates on nested queries and XML document relationships, in this paper we give attention to various XPath functionalities and structural complexity of the queries.

In future, we plan to test our technique in combination with PAT optimization proposed in [7]. We also plan to extend out work in the direction of multi query processing.

VII. REFERENCES

[1]. Arion and et. al. XQueC: Pushing Queries to Compressed XML Data. In (Demo) Proceedings of VLDB, 2003.

[2]. M. Brantner, S. Helmer, C-C. Kanne and G. Moerkotte. Kappa-join:Efficient Execution of Existential Quantification in XML Query languages, in the Proceedings of 4th International XML Database Symposium, XSym 2006 Seoul, Korea, September 10-11, 2006, in: Lecture Notes in Comput. Sci., vol. 4156, Springer, 2006, pp. 1-15.

[3]. Cheng, J., Ng, W.: XQzip: Querying compressed XML using structural indexing.In: EDBT. (2004) 219-236

[4]. Y. Diao, P. Fischer, M.J. Franklin, and R. To. YFilter: Efficient and Scalable Filtering of XML Documents. In Proc. of the 18th Intl. Conf. on Data Engineering, pages 341--342, San Jose, California, February 2002.

[5]. Document Relationships in XML : www.developer.com/xml/article.php/1575731

[6]. Dunren Che, Karl Aberer, Tamer Ozsu M, "Query Optimization in XMLStructured document databses", The VLDB Journal , 2006.

[7]. Dunren Che, Karl Aberer, Tamer Ozsu M, "Query Optimization in XML Structured document databses", The VLDB Journal , 2006.

[8]. Extensible Markup Language (XML) 1.0 (2nd Edition) W3C Recommendation, October 2000. http://www.w3.org/TR/REC-xml/

[9]. Jun-Ki Min, Myung-Jae Park, and Chin-Wan Chung. XPRESS: A Queriable Compression for XML Data. In Proceedings of the 2003 ACM SIGMOD international conference on Management of data, pages 122-133. ACM Press, 2003.

[10]. H. Liefke and D. Suciu. XMill: an efficient compressor for XML Data. Proc. of ACM SIGMOD Conf. on Management of Data, p. 153-164, 2000

[11]. Ning Zhang, Varun Kacholia, and M. Tamer Ozsu. A Succinct Physical Storage Scheme for Efficient Evaluation of Path Queries in XML. In Proceedings of the 20th International Conference on Data Engineering (ICDE), pages 54-65. IEEE Computer Society, 2004

[12]. W. Ng, W. Y. Lam, P. T. Wood and M. Levene. XCQ: A Queriable XML Compression System. An International Journal of Knowledge and Information Systems, (2005).

[13]. Pankaj M. Tolani and Jayant R. Haritsa. XGRIND: A query-friendly XML compressor. In Proceedings of the 18th International Conference on Data Engineering (ICDE),pages 225-234. IEEE Computer Society, 2002.

[14]. Radha Senthilkumar, Priyaa Varshinee, Dr. A. Kannan Designing and Querying a Compact Redundancy Free XML Storage, The Open Information Systems Journal, 2009 (Journal Under Publication)

[15]. Radha Senthilkumar, A. Kannan, M. Bhuvaneswari, "Query Optimization for Inter Document Relationships in XML Structured Document," iccima,pp. 25-32, International Conference on Computational Intelligence and Multimedia Applications - Vol.2 (ICCIMA 2007), 2007

[16]. Radha Senthilkumar, A. Kannan, V. Prasanna, P.Hindumathi , "QueryOptimization for Intra Document Relationships in XML StructuredDocument", International conference on open-source systems &Technologies (ICOSST 2007).

[17]. Radha Senthikumar , S. Priyaa Varshinee, S. Manipriya, M. Gowrishankar, A. Kannan, "Query Optimization of RFX Compact Storage using Strategy List", ADCOM December 2008

[18]. Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Compressing and searching xml data via two zips. In WWW, pages 751-760, 2006.

[19]. SAX : http://www.saxproject.org/

[20]. H. Wang, J. Li, J. Luo, and Z. He. XCpaqs: Compression of XML Document with XPath Query Support. In proceedings of the 2004 IEEE International Conference on Information Technology: Coding and Computing (ITCC'04).

[21]. World Wide Web Consortium :Standard Generalized Markup Language (SGML) : www.w3.org/MarkUp/SGML/

[22]. World Wide Web Consortium. XML Path Language (XPath) Version 1.0. http://www.w3.org/TR/xpath/, W3C Recommendation 16 November 1999.

[23]. World Wide Web Consortium. XPATH use cases: http://www.w3.org/TR/xpath#section-Node-Set-Functions

[24]. World Wide Web Consortium. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/, W3C Working Draft 22 August 2003.

Appendix A

XMark :

Q1: /site/people/person/pro_le[@income[[. >= \100"] & [. <= \1000"]]]

Q2: (.. [//open auctions[//bidder]]//closed auction[[buyer] & [seller]](//buyer/@person + //seller/@person + //description))

It retrieves the buyer/seller IDs and the description of those closed auctions which have
both a buyer and a seller, if the document contains open auctions which have a bidder.

Q3: //people/person[@id = ..//closed auction[[buyer/@person=nnsite/*/open auction [[bidder/$C>=100] j [[initial <= \30"] & [current <= \80"]]]/seller/@person] & [price >= \50"]]/seller/@person]/emailaddress

It retrieves the email addresses of those people who are sellers of some items which
were bidden at the price of >= $50 and the buyers of these items are the sellers of
some bidding items which have either >= 100 bidders or have an initial price <= $30
and a current price >= $50.

Q4 People with an income equal to the current price of some item {join on values}
/site/people/person[profile/@income=/site/open_auctions/open_auction/current]/name

Q5 Categories that are reachable from a given category in i steps, for i >= 1. {parametric join on keys} generator
Y(i)/id(.)/name for i = 1,2,...
Y(1) = /site/catgraph/edge[@from = "category0"]/@to
Y(i) = /site/catgraph/edge[@from = Y(i-1)]/@to for i >= 2

DBLP:

Q1: /dblp/inproceedings/booktitle

Q2: /dblp/inproceedings[booktitle = \SIGMOD Conference"]

Q3: /dblp/inproceedings[year[[. >= 1998] & [. <= 2000]]]

Q4: //inproceedings[[[publisher] & [editor]] & [author]]/journal

Q5: (//proceedings[[[publisher $=\IBM"] & [editor ?=\van"]] j [author/$C < 3]] (isbn
+ journal[. ! = \TKDE"]))
It retrieves the ISBN and non-TKDE journal of those proceedings which have either (1)
a publisher whose name starts with ($=) \IBM" and an editor whose name contains
(?=) \van"; or (2) less than 3 authors.

Lineitem:

Q1: /table/T/L TAX

Q2: /table/T[L TAX = \0.02"]

Q3: /table/T[L TAX[[. >= \0.02"] $ [. <= \0.04"]]]

Q4: (//*(L ORDERKEY + L PARTKEY + L SUPPKEY + L LINENUMBER + L QUANTITY + L DISCOUNT + L EXTENDEDPRICE + L TAX + L RETURNFLAG + L COMMENT + L LINESTATUS + L SHIPDATE + L COMMITDATE + L RECEIPTDATE + L SHIPINSTRUCT + L SHIPMODE))

Q5: //L DISCOUNT/$U

Shakespeare:

Q1: /PLAY/ACT/SCENE/SPEECH/SPEAKER

Q2: /PLAY/ACT/SCENE/SPEECH[SPEAKER = \PHILO"]

Q3:/PLAY/ACT/SCENE/SPEECH[SPEAKER[[. >= \MARK ANTONY"] & [. <\PHILO"]]]

Q4: // SPEECH[![STAGEDIR]]/SPEAKER/$C

Q5: [//SPEECH/SPEAKER[[. ! = \LUCE"] & [n_ [[![n_/STAGEDIR]] j [LINE/$C > 5]]]]]
It returns true if there is a speech, whose speaker is not \LUCE", that either (1) does
not have a STAGEDIR or (2) has more than 5 lines. It returns false otherwise.

Treebank:

Q1: //VP//NP//VP//NPnnVP.nVP

Q2: //PP//PP//PP//PP//PP//PP//PP

Q3: //PP/.PP//PP/.PP//PP/.PP//PP/.PP

Q4: //PPnnPP//NP

Q5: //PP[//PP]//NP

The five queries test Strategy list's performance on very nested data. They test a mixture of
axes: descendant (//), descendant-or-self (/.), ancestor (nn) and ancestor-or-self (.n).

Appendix B

The table consists of XPATH operators in decreasing order of their precedence.