Using a Relational Database for Scalable XML Search

Rebecca J. Cathey, Steven M. Beitzel, Eric C. Jensen, David Grossman, Ophir Frieder
Information Retrieval Laboratory
Department of Computer Science
Illinois Institute of Technology
Chicago, IL 60616

{cathey, beitzel, jensen, grossman, frieder}@ ir.iit.edu

June 3, 2007

Abstract

XML is a flexible and powerful tool that enables information and security sharing in heterogeneous environments. Scalable technologies are needed to effectively manage the growing volumes of XML data. A wide variety of methods exist for storing and searching XML data; the two most common techniques are conventional tree-based and relational approaches. Tree-based approaches represent XML as a tree and use indexes and path join algorithms to process queries. In contrast, the relational approach utilizes the power of a mature relational database to store and search XML. This method relationally maps XML queries to SQL and reconstructs the XML from the database results. To date, the limited acceptance of the relational approach to XML processing is due to the need to redesign the relational schema each time a new XML hierarchy is defined. We, in contrast, describe a relational approach that is fixed schema eliminating the need for schema redesign at the expense of potentially longer runtimes. We show, however, that these potentially longer runtimes are still significantly shorter than those of the tree approach.

We use a popular XML benchmark to compare the scalability of both approaches. We generated large collections of heterogeneous XML documents ranging in size from 500MB to 8GB using the XBench benchmark. The scalability of each method was measured by running XML queries that cover a wide range of XML search features on each collection. We measure the scalability of each method over different query features as the collection size increases. In addition, we examine the performance of each method as the result size and the number of predicates increase. Our results show that our relational approach provides a scalable approach to XML retrieval by leveraging existing relational database optimizations. Furthermore, we show that the relational approach typically outperforms the tree-based approach while scaling consistently over all collections studied.

1 Introduction

The Extensible Markup Language (XML) is a simple, flexible text format used for defining structured information. XML is semi-structured data, however, the structure is not as rigid, regular, or complete as typical structured data found in databases. Furthermore, since XML can be extended to include domain specific tags, information can be encoded with meaningful structure and semantics that allows rapid information sharing among devices and organizations [48]. XML was originally designed to meet the challenges of large-scale electronic publishing, however, the flexibility of XML has caused it to play an increasingly important role in the exchange of a wide variety of data [3]. Furthermore, languages to search XML are formally defined, allowing specific XML components to be retrieved regardless of an individual XML document's design.

The growing trend of using XML requires scalable technology to effectively store and search the volume and variety of data. This growth has led to the development of a wide range of XML query systems. These systems employ a variety of methods ranging from simple file systems to object relational databases. The two most common techniques for storing and searching XML are conventional tree-based approaches and relational approaches.

Tree-based approaches have typically been designed from the ground up to deal with elements, attributes, and text nodes while naturally handling document order and referential integrity issues [46]. Although, some tree-based approaches rely on tree-traversals to find specific elements within the XML tree, the majority of these methods use indexes and path join algorithms to speed up query processing. Tree-based methods are particularly effective for retrieving complex elements and searching larger sections of text.

In contrast to the tree-based approach, the relational approach seeks to utilize the power of a relational database. This method flattens the hierarchy of an XML document to store it in a relational database. XML queries are then relationally mapped to SQL to retrieve the desired results. Excluding a small engine used to parse the query and reconstruct the results, the database does the majority of the work. This method is promising because efficient access methods for relational data have been developed for over thirty years, and query planning and optimization in the relational algebra is well understood. In addition, relational systems offer key features for productive use, concurrence, transactions and safety [54]. However, relational systems sometimes need to use complex joins which can increase the time to process queries. In addition, reconstructing the XML often requires multiple calls to the database whereas a tree-based approach can extract the desired node directly from the tree hierarchy.

We compare two main approaches for XML retrieval. Both the tree-based approach and the relational approach have been developed to efficiently search XML data. We address the question of scalability by analyzing the two most common approaches to XML retrieval. Our study examines moderately sized XML collections ranging from 500MB to 8GB, similar in size to other used collections (376MB [2] and 4GB [17]). We examine not only the performance of each query on each system, but also the scaling multipliers as the collection size increase. We also examine the scalability of each approach as the size of the result set and the number of predicates increase. In this way, we can determine which approach has the potential to search large collections of XML data.

Although comparisons have been made between systems [7, 35], we are unaware of a study looking at the underlying techniques that each system uses. Our goal is to show that although relational databases are not the most common way of viewing XML, they provide increased performance over the tree-based approach. In addition we provide a framework for evaluating systems and techniques for future studies. All queries and collections used are available at `http://ir.iit.edu/collections`.

2 Background and Prior Work

2.1 XML Search

Motivated by the wide acceptance and use of XML, a rising number of XML retrieval systems are being developed. The semi-structured aspect of XML makes storing and querying it to be a challenging task. Although it has structure, the structure is not as rigid, regular, or complete as typical structured data found in databases. One of the major challenges of storing and searching XML data is preserving the hierarchy of the data. Conventional tree-based approaches use trees to represent the logical structural relationships between XML elements. Relational approaches flatten the hierarchy by shredding an XML document into edges or nodes that are stored in a relational table.

Since conventional top-down tree traversal approaches are inefficient for large document collections [33], the tree-based approach uses indexes to efficiently process queries on large document collections. Indexes are created for tree-based approaches through the use of numbering schemes. A numbering scheme assigns a unique identifier to each node in the logical document tree. The generated identifiers are then used in indexes as references to the actual nodes.

Numbering schemes range from simple level order numberings to more complex schemes. Numbering schemes optimize query processing by quickly determining structural relationships between nodes. A simple level order numbering assigns a unique identifier to every node while traversing the document in level-order. This scheme allows the ancestral relationship between nodes to be found by performing a simple calculation. Since the document is modeled as a complete *k*-ary tree, space is wasted by the insertion of spare identifiers [32]. Extensions of this scheme [36] partially drop the completeness constraint so the number of children a node may have is recomputed for every level of the tree. This reduces the number of spare identifiers needed. An alternative to level order numbering is a hierarchical numbering such as the dynamic level numbering proposed in [14]. This numbering scheme is based on variable-length identifiers and thus avoids a limit on the size of the document to be indexed [37]. Xing and Tseng propose an extended range-based labeling that combines prefix based labeling to eliminate the need to relabel nodes after arbitrary insertions [55]. Other approaches include using a variant of regions to express node-numbers [8] and extending the pre-order numbering scheme to allow ancestor-descendant relationships to be found in constant time [16]. Weigel, et al introduce the BIRD family of tree numbering schemes based on structural summaries. These schemes allow tree relations to be found and reconstructed with simple arithmetic operations [54]. Wang, et al present a method for ordering nodes based on sequences. This approach transforms structured XML data into sequences so that a structured query can be answered through subsequence matching [52]. Tatarinov, et al [49] studies three numberings: global, local, and dewey ordering. The global ordering assigns a single integral value to each node. Local ordering assigns a single value to each node, which denotes its relative order amongst its siblings in the XML document. Dewey ordering is similar to the Dewey decimal system, and stores with each node the concatenation of the local ordering identifiers of the node and its ancestors.

One of the most common ways to process queries with the tree-based approach is through the use of structural joins [13, 42]. The tree-based approach uses structural joins to resolve path expressions based on the features of the numbering scheme. Li, et al describe three algorithms for processing regular path expressions [33]. The *EE* join searches paths from an element to another, the *EA* join scans sorted elements and attributes to find elements attribute pairs, and the *KC* join finds kleene closure on repeated paths or elements. Zhang, et al [57] proposed a variation of the traditional merge join algorithm, called the multi-predicate merge join (MPMGJN) algorithm, for finding all occurrences of the basic structural relationships. Their results suggest that with some modifications, a native implementation in a relational database can support this class of query much more efficiently. Al-Khalifa, et al discusses two other families of structural join algorithms called the tree-merge and stack-tree [42]. The tree-merge algorithms are a natural extension of traditional merge joins and the multi-predicate merge joins [57], while the stack-tree algorithms have no counterpart in traditional relational join processing. Their results show that while, in some cases, tree-merge algorithms can have performance comparable to stack-tree algorithms, in many cases they are considerably worse.

In contrast to tree-based storage approaches, the relational approach flattens the hierarchy of an XML document by storing it in tables within a relational database. Yoshikawa and Amagasa classify methods for designing an XML relational database schema into two categories: structured-mapping approach and model-mapping approach [56]. In the structure-mapping approach, a database schema is defined for each XML schema or Document Type Descriptor (DTD) [18, 26, 44, 45, 50]. Storing XML documents with multiple schemas generally requires different tables to be created for each XML schema. The model-mapping approach addresses the issue of mapping XML documents without schemas. In this approach, a fixed database schema is used to store the structure of all XML documents. Examples of this include the Edge-oriented approach and the node-oriented approach. The edge-oriented approach developed by Florescu and Kossman is a simple scheme that stores all attributes in a single table [20]. Variants of the edge approach store the attribute names in another table [19] or store all associations of the same type in the same binary relation [43]. The node-oriented approach maintains nodes rather than edges [56]. With the start and end points of a node it maintains a containment relationship for ancestor-descendent relationships. Grust, et al [23] present a database structure specifically for XPath queries, where the predecessor, size and level of an element are stored and used to tailor specifically to XPath queries.

Searching XML documents stored in a database requires XML queries to be relationally mapped to SQL. Krishnamurthy, et al examine existing work for relationally mapping XML queries to SQL and describes several open problems [30]. An algorithm that handles the open problem of recursive XML schemas is proposed in [29]. Another method is presented for translation from XQuery to SQL without requiring subsequent calls to the database [24].

The expanding popularity of XML has led to XML support in several commercial databases. These XML enabled relational databases use object relational mappings to model the XML data as a tree of objects that are specific to the data in the document. Microsoft's SQL server 2005 stores the data as a BLOB and then provides a primary XML index that shreds the XML into a node table and adds some secondary indexes for improving XQuery performance [38, 41]. Oracle and IBM DB2 provide a shredded physical representation for certain schematized XML and a BLOB for the general case. [40]. Krishnaprasad, et al [31] describe how queries can be rewritten for more efficient processing on object relational frameworks with Oracle. Furthermore, Oracle translates XQuery into the same internal data structures as SQL such as sub query blocks and SQL operators which enables the same underlying optimizer and execution engine to be utilized [34].

Some methods combine the large body of work that has gone into developing a relational database with the indexing structures inherent in tree-based approaches. SystemRX is a hybrid system that does this [10]. They leverage the years of data management research to advance XML technology to the same standards expected from mature relational systems. Weigel, et al combine the indexing structures of a native XML database with the power of a relational database in [53].

Vakali examines several emerging practices for storing XML data with a particular emphasis on native XML storage approaches [51]. Furthermore, Vakali examines the features of several publicly available XML retrieval systems that use different storage techniques for XML storage. MonetDB/XQuery is an XML retrieval engine that uses the translation method described in [24] to store and query a relational database [11].

2.2 Collections

There are two types of XML documents: data-centric and topic-centric. A topic-centric document contains significantly more text than element tags, while a data-centric document dedicates more parts to tags. Our study focuses on data-centric XML retrieval .

Several collections have been used for evaluating XML search systems. The Bosak Shakespeare collection contains the complete plays of Shakespeare marked up in XML [12]. This collection is widely used when experimenting with XML search systems [25, 56], however, the size of this collection is only 7.65MB. Another collection that is often used is the digital bibliography and library project (DBLP) collection [2]. The DBLP collection provides bibliographic information on major computer science journals and proceedings in the form of XML. The size of DBLP is 376MB. Because of the need to evaluate existing XML systems, the INitiative for the Evaluation of XML retrieval (INEX) [6] has started an international effort to promote evaluation procedures for topic-centric XML retrieval [9]. The aim of INEX is to provide a large XML test collection with appropriate scoring methods. INEX uses several collections based on the Wikipedia XML corpus. The main corpus is around 4.4GB in size [17].

Several benchmarks are available for the evaluation of XML search systems. Afanasiev and Marx analyze the five most popular XML benchmarks and examine how each benchmark is used, what they measure, and what can be learned from each one [7]. Manegold compares the performance of different XQuery engines using multiple benchmarks [35]. Our study differs from previous studies because it compares specific techniques for XML retrieval rather than specific hardware. We evaluate methods rather than systems. One of the top five most popular XML benchmarks is XBench [5]. XBench generates collections between 100KB and 10GB in size. In addition, the type of XML can be specified. For our study, we generated a multiple document, data-centric XML document collection using XBench.

3 Search Methods

We examine the relational and tree-based approaches to XML retrieval. For each approach, we discuss the two primary aspects of XML retrieval. The first aspect is storage of the data. The second is searching the data. This study focuses on the search aspect of XML retrieval, however, the speed at which an approach searches XML is largely dependent on the storage method. Therefore, we also discuss the methods used for storage.

3.1 Relational Approach

The relational approach stores multiple schema XML documents in a relational database. We use the SQLGenerator [1] to show the power of the relational approach. The SQLGenerator is an XML retrieval engine that uses MySQL version 4.1.11 to store and search XML. All discussions of the storage and search techniques are the techniques employed specifically by the SQLGenerator, however, any static schema relational approach should employ similar techniques.

3.1.1 Storage

The relational approach uses the edge-oriented, model-mapping approach to store a heterogeneous collection of XML documents in a static schema relational database. This is similar to the method first described by Florescu and Kossman in [20]. Each unique XML path and that path's value are stored as a separate row in a relational table. This table also has the values in-lined in the same table. This is a static schema that is capable of storing any XML document without modification. The hierarchy of XML documents is kept intact such that any document indexed by the database can be reconstructed using only the information in the tables. We use `books.xml` (see Figure 1) as an example file to show our storage scheme and the relational approach's translation over it. All of the database tables shown in our examples will reflect the data of this file.

The four main tables in the database are the `pinndx`, `tagpathtbl`, `tagnametbl`, and `atrnametbl` tables. In Figure 2, we show the content of each table after the insertion of `books.xml`. The `pinndx` table (Figure 2(a)) stores the actual content of all the XML files that have been indexed. Each row in the `pinndx` table stores information about an XML element. The `pinndxnum` column is a unique integer assigned to each element and attribute in a document. The `collectionnum` column can be used to divide the `pinndx` table into collections. The `parent` column indicates the `pinndxnum` value of the tag that is the parent of the current tag. The `tagtype` column indicates whether the path terminates with an element or attribute. The `pinnum` column indicates the XML document this row corresponds to. The

```
<books>
    <book>
        <name>The Great Gatsby</name>
        <author>F. Scott Fitzgerald</author>
        <price currency="USD">9.99</price>
    </book>
    <book>
        <name>Cat in the Hat</name>
        <author alias="true">Dr. Seuss</author>
        <price currency="USD">14.99</price>
    </book>
    <pamphlet>
        <name>Common Sense</name>
        <author>Thomas Paine</author>
    </pamphlet>
</books>
```

Figure 1: `books.xml`

pinndxnum	collectionnum	parent	tagpath	tagtype	tagname	atrname	pinnum	indexpos	nvalue	value
1	1	0	1	E	1	1	1	0		
2	1	1	2	E	2	1	1	0		
3	1	2	3	E	3	1	1	0		The Great Gatsby
4	1	2	4	E	4	1	1	0		F. Scott Fitzgerald
5	1	2	5	E	5	1	1	0	9.99	9.99
6	1	5	5	A	5	2	1	0		USD
7	1	1	2	E	2	1	1	0		
8	1	7	3	E	3	1	1	0		Cat in the Hat
9	1	7	4	E	4	1	1	0		Dr. Seuss
10	1	7	5	E	5	1	1	0	14.99	14.99
11	1	10	5	A	5	2	1	0		USD
12	1	1	6	E	6	1	1	0		
13	1	12	7	E	3	1	1	0		Common Sense
14	1	12	8	E	4	1	1	0		Thomas Paine

(a)

vkey	value
1	[books]
2	[books, book]
3	[books, book, name]
4	[books, book, author]
5	[books, book, price]
6	[books, pamphlet]
7	[books, pamphlet, name]
8	[books, pamphlet, author]

(b)

vkey	value
1	books
2	book
3	name
4	author
5	price
6	pamphlet

(c)

vkey	value
1	-
2	currency

(d)

Figure 2: The (a) `pinndx`, (b) `tagpathtbl`, (c) `tagnametbl`, and (d) `atrnametbl` Tables

indexpos column is used for queries that use the index expression feature of XML search and indicates the position of this element relative to others under the same parent (starting at zero). This column stores the original ordering of the input XML for explicit usage in users' queries. The value column stores the textual contents of the element while the nvalue column contains the numeric representation of the value. The tagpath, tagname, and atrname correspond to primary keys in the tagpathtbl, tagnametbl, and atrnametbl tables, respectively.

The tagpathtbl (Figure 2(b)), tagnametbl (Figure 2(c)), and atrnametbl (Figure 2(d)) store the metadata (data about the data) of the XML files. The tagpathtbl and tagnametbl tables together store the information about tags and paths within the XML file. The tagpathtbl table stores the unique paths found in the XML documents. The tagnametbl table stores the name of each unique tag in the XML collection. The atrnametbl stores the names of all the attributes. In each of these tables, vkey is an integer assigned by the system and is the primary key of the table. When an XML document is indexed and an element, path, or attribute is encountered for the first time, the values are added to the tagnametbl, tagpathtbl, and atrnametbl tables. Otherwise, the indexer caches these tables in memory and uses their values to insert new rows into the pinndx table.

To optimize database performance, several indexes are built on the pinndx table. In addition to the primary key index, we built indexes on the *parent*, *indexpos*, *tagpath*, *tagname*, *atrname*, *pinnum*, and *nvalue* columns of the pinndx table. Although an index on the value column would speed up simple string matching queries, some of the values are too large to realistically index. Since an index on all the values would be too large, we build a partial index on the value column that includes the first 30 characters of each value. The index built on the parent column is useful when complex joins are necessary to enforce ancestor descendant relationships. MySQL does not support multiple indexes for a query, however, we have found that indexes built on multiple columns are useful for most of the queries. Several multicolumn indexes were also built on the *tagpath* and *parent*, *tagpath* and *value*, *tagpath* and *nvalue,* and *tagpath* and *pinnum* columns.

3.1.2 Search

In Figure 3, we illustrate the complete algorithm for relational XML search. There are three phases of the algorithm. First an XML query is translated to an SQL query. Then the SQL query is executed on the relational database. Finally, the XML is reconstructed from the database results.

Relationally Map XML Query

The first step in the algorithm relationally maps an XML query to SQL. The XML query is first examined to find the specific constructs of the query. These constructs are then translated to SQL.

The query is first parsed to find all the path expressions. A path expression consists of a set of steps that form paths. The path expression /A/B/C consists of three steps. The algorithm looks at each step and adds those steps to an address. If the path contains a predicate, then multiple addresses are created. For the path expression /A/B[C="5"], two addresses are created. The first is for the path, /A/B, the second is for the path /A/B/C. Then since the second address contains a literal, the value 5 is added to the second address. If the path expression is bound to a variable, that variable is also added to the address. If the path expression is in a return clause, the variable and address are added to the results template. The template is a document object model (DOM) template that determines the formatting of the final results. The elements in the return clause of a query are added to the template for formatting. In addition, each address in the template is also added to the set of all addresses. If the query is simple and does not contain a return clause, the element corresponding to the final step in the address that is not contained in a predicate is returned with no extra formatting. If the address is in an order by clause, the address is added to the set of elements used to order the SQL query.

Once the query is parsed, each address is examined separately. An address is resolved by querying the database and returning the set of tagpaths from the tagpathtbl where the path matches the address. Since regular expressions are used to find the path, there is no time difference between queries using '/' or '//'. An address may have multiple matching paths. For example, the path /A//D matches the path /A/C/D and /A/B/C/D. For each resolved address, an alias to the pinndx table is created. The alias is then added to the list of SQL selects. Multiple matching tag paths are processed through the use of SQL UNIONs. Predicates to enforce the tagpath and tagtype are added to the list of SQL predicates. If the address is bound to a literal, a predicate to enforce the literal value is also added to the list of SQL predicates. Furthermore, predicates are also added to enforce the hierarchy between that address and the previous address. The query /A/B[C="5"] consists of two paths /A/B/ and /A/B/C, where /A/B/C is bound to the literal 5. After the SQL predicate to enforce the value of "5" is added for the second alias, SQL predicates to

```
Algorithm
addresses={ø}, sqlSelects={ø}, sqlFroms={ø}, sqlPredicates={ø},
sqlOrderBy={ø}, resultSet={ø}, template={emptyDOMtemplate}

Step 1: Translate XML Query to SQL Query
  for each path_expression in Query do
    for each step in path_expression do
      address ← step
      if step is a literal then
        bind literal and type to address
      end if
    end for
    if path_expression bound to variable then
      add variable to address
    end if
    addresses ← address
    if path_expression in order by clause then
      SQLOrderBy ← address
    end if
    if path_expression in return clause then
      template ← address
    end if
  end for
  for each address in addresses do
    tagpaths ← all matching tagpaths from tagpathtbl
    create alias to the pinndx table
    sqlFroms ← alias
    for each tagpath in tagpaths do
      sqlPredicates ← predicate to enforce the data source
      sqlPredicates ← predicate to require presence of this tagpath
      if address is bound to literal then
        sqlPredicates ← predicate to enforce literal value
        sqlPredicates ← predicates to enforce hierarchy
      end if
    end for
  end for
  construct SQL from sqlSelects, sqlFroms, sqlPredicates and sqlOrderBy


Step 2: Execute SQL Query
  resultSet ←execution of SQL


Step 3: Reconstruct XML
  create DOM element, domResults from template
  for each row in resultSet do do
    if row is a simple element then
      domResults ← row value
    else
      domResults ← row.toDOM()
    end if
  end for
  return domResults
```

Figure 3: Relational Search Algorithm

enforce the hierarchy are also added. In this case, a predicate that says the second path alias's `parent` is equal to the `pinndxnum` of the first path since `B` is the parent of `C`. Finally, for each address in the template, the value for the corresponding `pinndx` alias is added to the list of SQL selects.

In general, the final SQL query is constructed using the list of SQL selects, SQL predicates, SQL froms and SQL order bys. The query is formed using the following format: `"SELECT [SQL selects] FROM [SQL froms] WHERE [SQL predicates] ORDER BY [sql order by]"`. The method described in Figure 3 does not go into detail on the translation of many of the advanced features of XML queries, particularly certain constructs of XQuery and XML-QL that require more advanced features of SQL.

An example of translation from an XML query to SQL is given in Figure 4. The XQuery query in this example returns the identifier and price of all items bought by a specific customer using a Visa credit card. The relational algorithm first finds all the addresses in the query. This query has five addresses: `/order/item`, `/order/item/customer_id`, `/order/item/credit_card/type`, `/order/item/id`, and `/order/item/price`. Five aliases to the pinndx table are created, one for each address: `q0`, `q1`, `q2`, `q3`, `q4`, and `q5`. These aliases are added to the list of SQL froms. Each address is resolved by querying the `tagpathtbl` table in the database. For this example we assume the `tagpath` values for each address are 1, 4, 5, 7, and 8, respectively. For each alias, the `tagpath` values are added to the list of SQL predicates. For `q0`, the predicate `q0.tagpath="1"` is added to the list of SQL predicates. This process is repeated for all of the aliases. Since `q3` and `q4` are in the return clause of the query, `q3.value`, `q3.pinndxnum`, `q4.value`, and `q4.pinndxnum` are added to the list of SQL selects. Both `q1` and `q2` appear in a predicate condition. Since `q1` is bound to a numerical value, the `nvalue` column is used. To enforce the predicate value. `q1.nvalue=3` and `q2.value="VISA"` are added to the list of SQL predicates. The results should be ordered by the item identifier where the identifier is a numerical value. To enforce this ordering in the SQL query, `q3.nvalue` is added to the list of SQL order bys. Next, SQL predicates are added to enforce the relationships between the base path `/order/item` and all other paths. The path `/order/item` is the base path because it is bound to the `$item` variable and all other paths are formed using the `$item` variable. Since `customer_id` is a child of `item`, a simple predicate to enforce the parent child relationship is added to the list of SQL predicates, `q1.parent=q0.pinndxnum`. Similar predicates are added for the `q3` and `q4` aliases. The `q2` alias is more complex. Since there are two elements between `q2` and `q0`, another alias must be created to enforce the hierarchy. This alias is called `q2_1`. The new alias is added to the list of SQL froms. Then, the predicates `q2.parent=q2_1.pinndxnum` and `q2_1.parent=q0.pinndxnum` are added to the list of SQL predicates. Finally, the SQL query is constructed from the lists of SQL selects, SQL froms, SQL order bys, and SQL predicates.

This step runs in time dependent on the size of the query. It is usually the fastest step since it is not dependent on the collection size.

```
let $item := /order/item
where $item[customer_id=3 and credit_card/type="VISA"]
order by $item/id
return
  <item>
       <id>{$item/id}</id>
       <price>{$item/price}</price>
  </item>
```

$\longrightarrow$

```
SELECT DISTINCT q3.value, q3.pinndxnum, q4.value,
q4.pinndxnum FROM  pinndx q0, pinndx q1, pinndx q2,
pinndx q2_1, pinndx q3, pinndx q4 WHERE
q0.tagpath=1 AND q1.nvalue=3 AND q1.tagpath=4
AND q1.parent=q0.pinndxnum AND q2.tagpath=5
AND q2.value="VISA" AND q2.parent=q2_1.pinndxnum
AND q2_1.parent=q0.pinndxnum AND q3.tagpath=7
AND q3.parent=q0.pinndxnum AND q4.tagpath=8 AND
q4.parent=q0.pinndxnum ORDER BY q4.nvalue
```

Figure 4: Relational Mapping of XQuery Query to SQL

SQL Execution

The second step in the relational search algorithm is to execute the generated SQL. This step is handled by the relational database and is often the most time-consuming step. An optimized database, however, can vastly improve performance. Without indexes, the database starts with the first record and reads through the whole table to find the relevant rows. If the table has an index for the columns in the query, the database can quickly determine the positions to seek to in the middle of the data without having to look at all the data. We use MySQL as our back-end database. Most MySQL indexes are stored in B-trees. If a multiple-column index exists on `col1` and `col2`, the appropriate rows can be fetched directly. If separate single-column indexes exist on `col1` and `col2`, the optimizer tries to find the most restrictive index by deciding which index finds fewer rows and using that index to fetch the rows.

Although the storage method used is simple, often many self-joins of the `pinndx` table are required to retrieve a given XML element; one join for each sub-element [47]. Sometimes multiple self-joins can be very time consuming, however simpler XML queries that do not require joins execute efficiently.

XML Reconstruction
Once the results are returned from the database, the result set is iterated through and the known values are replaced into the template obtained from the return clause. The template accepts the current result and returns a document object model (DOM) document fragment. A list of these fragments is compiled and appended to our root document. If the element is a complex element, `toDOM()` is called. `toDOM()` queries the database for all the children of the element and returns DOM objects for each child. The children are added to the template and the resulting XML fragment is added to the root document. The document is then formatted, and the resulting XML text is sent as output to the user.

3.2 Tree-Based Approach

The tree-based approach stores stores XML hierarchically as a tree. We use eXist 1.0 [4], a popular native XML database to test the tree-based approach. All discussions of the storage and search techniques are the techniques employed specifically by eXist, however, a tree-based approach should employ similar techniques.

3.2.1 Storage

Tree-based approaches are based on the XML data model, rather than the relational data model. This means that they are designed from the ground up to deal with elements, attributes and text nodes, and they naturally handle document order and referential integrity issues [46]. The tree-based approach represents XML as a tree. Every node in the tree is labeled with a unique identifier. This allows quick identification of structural relationships between a set of given nodes. It also allows direct access to nodes by their unique identifier. Furthermore, it reduces IO operations by deciding XPath expressions based on node identifiers and indexes.

Numbering schemes range from simple level order numberings to more complex schemes. In Figure 5, we show an example of the numbering scheme used by eXist, a variant of the $k-ary$ numbering schema [36], when storing `books.xml`. The tree-based approach employed uses a level order number numbering scheme where a unique integer identifier is assigned to every node while traversing the tree in level-order. Level-order numbering schemes model the document tree as a complete k-ary tree assuming that every node in the tree has exactly k child nodes. Since the actual number of children nodes varies in real documents, the remaining child identifiers are left empty before continuing on to the next nodes. Consequently, the available identifiers can run out thus limiting the maximum size of a document to be indexed. To solve this problem, eXist recently changed their numbering scheme to use hierarchical level numberings [37], however, our experiments use the previous version of eXist. Since the individual files in our collections do not exceed the maximum size, we did not have any issues indexing files with eXist.

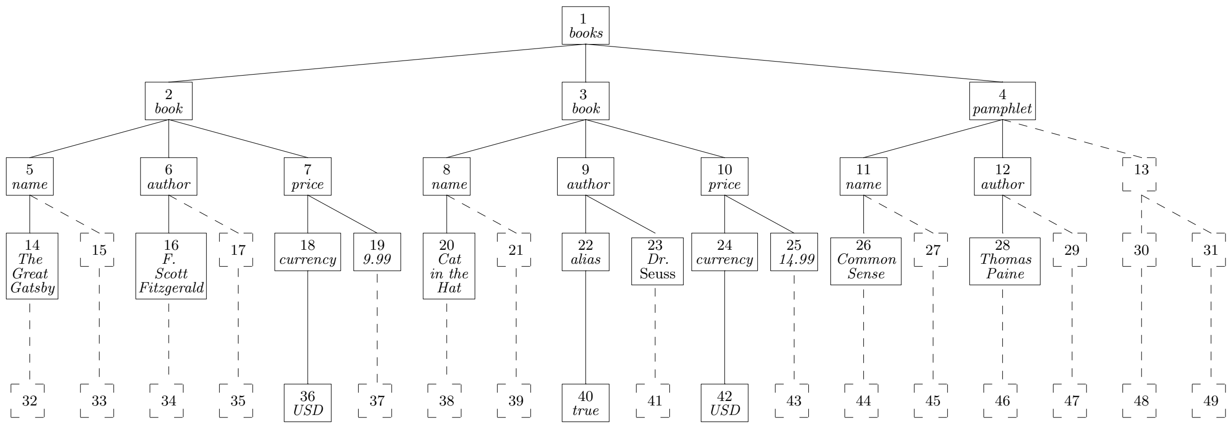

Figure 5: Tree-Based Approach Storage

A level-order numbering scheme allows the relationship between parent, sibling and possible ancestor/descendant nodes to be easily determined using a simple calculation. The completeness constraint, however, imposes a major restriction on the maximum document size. Generally, an XML document will have fewer children near the root than near the leaf nodes. Thus, in a worst case scenario, a single node at a deeply structured level may have many children. This causes a large number of spare identifiers to be inserted at all levels of the tree to satisfy the completeness rule. This causes the assigned identifiers to grow fast even for small documents. An extension to this method partially drops the completeness constraint, requiring that each node have the same number of children as the node with the maximum children on a specific level. Now, the number of children a node has is recomputed on each level. We illustrate this constraint in Figure 5. In this example, the `book` elements have three children each. Because of the completeness constraint, the `pamphlet` element must also have three children, so a spare identifier is used. This approach accounts for the fact that typical documents will have a larger number of nodes at lower levels of the document tree. Furthermore, the document size limit is raised considerably to enable indexing of much larger document. Information about the number of children each level of the tree can have is stored in a simple array.

The tree-based approach provides storage of schema-less XML documents in hierarchical collections. XML documents are stored as a multi root B+-tree using the variant of the $k-ary$ numbering schema for indexes as described in [36]. Four index files are used to store information about an XML document:. `collections.dbx`, `dom.dbx`, `elements.dbx`, and `words.dbx`. `collections.dbx` manages the collection hierarchy. `dom.dbx` contains the association of nodes with unique node identifiers. `elements.dbx` contains the indexed elements and attributes. `words.dbx` keeps track of word occurrences and is primarily used for fulltext search extensions. To reduce disk space usage, an index on specific node values is not used.

3.2.2 Search

In Figure 6, we illustrate the complete algorithm for tree-based XML search. There are three steps to the tree-based algorithm. The first step decomposes the path expressions in an XML query. The second step generates node sets corresponding to each element in the query. Finally, the third step runs a path join algorithm on the node sets to determine the relationship between the nodes in the node sets.

Decomposition of Path Expressions
The first step decomposes all path expressions in an XML query. A path is decomposed by splitting it into individual components and then forming subexpressions by combining each component with the component that comes after it and the component that comes before it. For example, the path `/A/B[C="D"]` would be split into three subexpressions `A/B`, `B[C`, and `C=D`. Like the translation step for relational search, this step is not dependent on the size of the XML collection. It runs in time proportional to the length of the path.

Generate Node sets
The next step generates node sets for each subexpression. The exact position of each element is provided in the `elements.dbx` index file. A node set is generated by loading all the root elements for all documents in the input document set. Each node set consists of <document-id, node-id> pairs, ordered by document identifiers and unique node identifiers. Each subexpression consists of two elements. The node set is only generated for the second element in each subexpression except the first subexpression for a path. This subexpression generates both node sets. For the path `/A/B/C`, two subexpressions are generated: `A/B` and `B/C`. Node sets are generated for both `A` and `B` for the first subexpression, however, only a node set for `C` is generated for the second subexpression. At the completion of this step, $m+1$ node sets have been generated, where m is the number of subexpressions.

Path-join Algorithm
The final step determines relationships between node sets using a path join algorithm. The node sets are examined to determine which elements of each set are descendants of the nodes in the next set. The path join algorithm takes two ordered sets as input. The first contains potential ancestor nodes. The second contains potential descendants. Every node in the two input sets is described by <document-id, node-id> pairs. The path join algorithm starts with the first subexpression for a path. The first node set becomes the ancestor node set and the second node set becomes the descendant node set. The ancestor node set is the list of potential ancestors for all of the nodes in the descendant node set. The actual ancestors are found by iterating through the list of descendants. Every node in the two input sets is described by <document-id, node-id> pairs. The parent of each node in the descendant node set is found and

```
Algorithm
$basicSteps=\{ø\}$, $nodeSet=\{ø\}$, $finalResults=\{ø\}$

Step 1: Decompose Path Expressions
  for each $path$ in Query do
    for each $step$ in $path$ do
      if $step$ is not last step in $path$ then
        $subexpressions_{path} \leftarrow$ createSubExpression($step_i$, $step_{i+1}$)
      end if
    end for
  end for

Step 2: Generate Node Sets
  for each $path$ in Query do
    for each $subexpression$ in $subexpressions_{path}$ do
      if $subexpression$ is first for $path$ then
        $subexpression_{nodeset1} \leftarrow$ node set for first element in subexpression
        $subexpression_{nodeset2} \leftarrow$ node set for second element in subexpression
      else
        $subexpression_{nodeset2} \leftarrow$ node set for second element in subexpression
      end if
    end for
  end for

Step 3: Path Join Algorithm
  for each $path$ in Query do
    $generatedNodeSet \leftarrow nodeSet_1$
    $descendantNodeSet \leftarrow nodeSet_2$
    for each $subexpression$ in $subexpressions_{path}$ do
      $ancestorNodeSet \leftarrow generatedNodeSet$
      $descendantNodeSet \leftarrow nodeSet_2$
      $generatedNodeSet \leftarrow \{ø\}$
      for each $desc$ in $descendantNodeSet$ do
        $parent \leftarrow$ parent of $desc$
        while $parent$ exists do
          for each $anc$ in $ancestorNodeSet$ do
            if $anc.node\_id$ equals $parent.node\_id$ then
              $generatedNodeSet \leftarrow desc$
            end if
          end for
        end while
      end for
    end for
  end for
```

Figure 6: Tree Approach Search Algorithm

compared with all of the nodes in the ancestor node set. If no match is found, the parent of the parent is found and similarly compared with the ancestor node set. The algorithm stops when there are no more parents. If the parent matches a node in the ancestor node set, the <document-id, node-id> pair from the original descendant node set is placed in a temporary node set. Once the algorithm has completed for each parent of each node in the descendant node list, the temporary set is used as the new ancestor node list for the next subexpression. For example, the path `/A/B/C` is decomposed into two subexpressions: `/A/B` and `B/C`. The resulting node set for the expression `A/B` becomes the ancestor node set for the expression `B/C`.

Bremer, et al claim that keeping both lists in document order allows for executing this kind of join operation in at most linear time with respect to the number of matching pairs of node identifiers [13].

4 Methodology

Generally speaking there are two main facets of searching XML: content based and structure-based. Content-based XML retrieval focuses on the traditional Information Retrieval (IR) notions of relevance while structure-based XML retrieval focuses on semi-structured querying of hierarchical data. The INitiative for the Evaluation of XML (INEX) [9] focuses on providing means to evaluate content-based XML retrieval. Their evaluation focuses on queries that are relevant to the results. In contrast, we perform a *scalability study* to compare our relational approach against a common XML retrieval approach.

4.1 Collections

To measure the scalability of structure-based XML retrieval, large collections of XML documents were needed. Since we are unaware of large collections of real world XML data that meet our requirements, we chose to use an XML benchmark to create synthetic XML. We use XBench [5], one of the top five most popular XML benchmarks [7], as we wanted to test the scalability over a large collection of XML documents. We modified the XBench templates to create a heterogeneous collection of multiple schema data-centric XML documents. Data-centric documents were chosen because they contain more structure to text rather than documents marked up in XML, however, we also included a subset of documents that contained primarily text. We chose to focus on data-centric XML to show how each approach handles structure when searching XML. The generated XML captures e-commerce transactional data. Three types of XML files were generated: orders, customers, and items. The generated XML files test the ability of an XML retrieval system to process queries with a large quantity of matching paths, to search large XML documents, and to perform simple text matching over large sections of free text.

We generated an 8GB collection from the modified XBench templates. Then, we created 4GB, 2GB, 1GB, and 500MB collections from random subsets of the 8GB, 4GB, 2GB, and 1GB collections, respectively. The collections were designed to test the performance of an XML retrieval system as the size of the collections exceeds the size of the memory. In Table 1, we show details for each collection.

collection	actual size	# elements	# attributes	# paths	depth
500MB	598.5MB	5,239,454	827,664	82	3-7
1GB	1.1GB	9,900,264	1,566,570	82	3-7
2GB	2.2GB	19,613,565	2,169,724	82	3-7
4GB	4.2GB	37,836,196	4,293,482	82	3-7
8GB	8.5GB	75,211,108	8,542,838	83	3-7

Table 1: Collection Details

4.2 Queries

XBench provides a set of twenty queries that challenge a system with XML-specific features as well as conventional functionalities. Since XBench generates four different types of XML, not all of the queries run on each type. We used a subset of eleven queries designed specifically for the multiple schema data-centric XML document collection. We maintain the original numberings used by XBench. The XML features covered by the queries include exact match, ordered access, quantified expressions, regular path expressions, sorting document construction, retrieving individual documents, and text search. Q1 tests shallow queries, it matches only the top level of XML document trees. Q5 returns data based on the order in a document. Q6 and Q7 test for the existence of some elements that satisfies a condition, or whether all the elements in the same collection satisfy a condition. Two queries test regular expressions. Q8 tests unknown element name and Q9 tests unknown sub-path. Q10 and Q11 test the ability of the system to efficiently sort values both in string and in non-string data types. Q12 tests the ability of a system to retrieve fragments of original documents with original structures. Q17 tests the ability of the system to search for matching text.

In addition to the XBench queries, we created a set of queries designed to compare the performance of the tree-based approach with the relational approach. These queries test large result sets and multiple predicate matching. The collections and full query sets are available and explained in more detail at `http://ir.iit.edu/collections`.

4.3 Engines

We use two XML retrieval systems to compare the performance of the relational approach with the tree-based approach: The SQLGenerator [1] and eXist [4] 1.0. The SQLGenerator uses a model mapping relational approach while eXist uses a XML-specific B+-tree indexing approach. Both systems were shown to be scalable through some initial scalability studies. eXist was compared against other XPath query engines to show its efficiency. A second experiment determined the scalability of eXist by observing linear query execution time over collections ranging in size from 5MB to 39.15MB [37]. Although the conclusion of this study showed linear execution time for eXist, the collections used were not large enough to draw reliable conclusions about the scalability of eXist on modern, real-world collections. Similarly, the SQLGenerator performed initial scalability studies by running a wide range of XML queries on increasingly large collections ranging in size from 500MB to 8GB [15].

5 Results and Analysis

All testing was performed on a ProLiant DL380 G4. This model has 4 Intel Xeon 3.4GHz with 1MB L2 Cache, 2GB RAM on 4 DIMMs, and a 178GB Hardware RAID-5 array (composed of 4 10kRPM SATA/a disks). The machine is running Redhat Enterprise Server 3.0. The primary mode of comparison is the total execution time to run the query using each system. All timings given represent the average execution time of the queries (in random order) over five runs. To ensure a cold cache, the server was rebooted between runs.

Our results focus on the examination of several key aspects of scalability. We compare the scalability of both approaches using the XBench queries as the collection size increases and as the query features change. We also compare the scalability of both methods for increasing result sets and an increasing number of predicates.

A large body of research examines the computational complexity of searching relational and tree structured data [21, 22, 27, 28, 39]. The majority of this work defines complexity classes or upper bounds. In some cases, subsets of the search are examined to find a lower expected complexity. Searching XML when stored as relational or tree-based data does not fit into the specific subsets of search described in prior work. Too many variables and internal system unknowns are involved with both approaches to adequately predict theoretical behavior. Although a theoretical analysis would be useful, given the disparity in the timing results shown later, we do not attempt to theoretically analyze or bound the relational or tree-based approaches. In short, we show the difference of each approach strictly through practical experimentation.

5.1 Query Features

We examine the scalability of different XML query features on increasing collection sizes. In Table 2, we provide the raw timings for each query over all the collections. In Figure 7, we plot the average time for each query feature over all collections. In Figure 8, we plot the average performance over all query features. In Table 3, we show the scaling multipliers for each query. The scaling multiplier for a query is calculated by comparing the time to execute on each collection with the time to execute on the 500MB collection. In Figure 9, we examine the scaling multiplier of each query feature as compared to the 500MB collection. In Figure 10, we plot the average scaling multiplier over all collections for the relational and tree-based approaches.

In Figures 7 and 9, we plot the performance of each individual query type. The exact match ((a) in both figures), ordered access ((b) in both figures), document structure preserving ((f) in both figures), and retrieving individual documents ((g) in both figures) features all exhibit similar performance for both approaches over all collections. The

	Collection Size									
	500MB		*1GB*		*2GB*		*4GB*		*8GB*	
	tree	rel	tree	rel	tree	rel	tree	rel	tree	rel
Q1	15.52	0.43	26.75	0.44	59.41	0.51	131.11	0.71	484.24	0.67
Q5	14.52	0.77	25.72	0.78	56.20	0.96	129.61	1.27	498.83	1.47
Q6	2.03	31.08	4.64	50.26	18.97	52.37	28.73	183.10	TIMEOUT	466.58
Q7	2.85	25.0	6.43	38.88	10.32	71.70	26.94	243.15	217.72	594.19
Q8	14.57	0.58	25.89	0.51	56.52	0.69	131.00	0.87	478.43	1.10
Q9	14.73	0.40	25.38	0.41	53.50	0.46	124.22	0.63	426.58	0.72
Q10	16.22	1.16	29.89	1.51	68.35	2.73	139.53	5.97	1,850.99	13.22
Q11	16.75	0.68	29.73	0.76	65.58	1.46	137.04	2.81	1,829.84	6.17
Q12	14.35	0.64	25.41	0.63	54.03	0.79	129.80	0.89	458.05	1.06
Q16	14.41	0.58	25.11	0.52	51.29	0.70	125.06	0.76	440.00	0.80
Q17	32.33	1.78	53.14	2.25	97.53	19.50	638.29	37.29	1,317.18	21.86
mean	**14.39**	**5.74**	**25.27**	**8.81**	**53.79**	**13.81**	**158.29**	**43.40**	**800.18**	**100.71**
total	**158.28**	**63.09**	**278.09**	**96.94**	**591.7**	**151.88**	**1741.33**	**477.44**	**8001.86**	**1107.84**

Table 2: Total Query Time (seconds) for XBench Queries

	Collection Size							
	1GB		*2GB*		*4GB*		*8GB*	
	tree	rel	tree	rel	tree	rel	tree	rel
Q1	1.72	1.02	3.83	1.19	8.44	1.65	31.20	1.55
Q5	1.77	1.01	3.87	1.25	8.92	1.65	34.35	1.91
Q6	2.29	1.61	9.34	1.69	14.15	5.89	–	15.01
Q7	2.26	1.56	3.62	2.87	9.45	9.73	76.39	23.77
Q8	1.78	0.88	3.88	1.19	8.99	1.5	32.84	1.90
Q9	1.72	1.03	3.63	1.15	8.43	1.58	20.96	1.8
Q10	1.84	1.30	4.21	2.35	8.60	5.15	114.12	11.40
Q11	1.77	1.12	3.91	2.15	8.18	4.13	109.24	9.08
Q12	1.77	0.98	3.77	1.23	9.05	1.39	31.92	1.66
Q16	1.74	0.90	3.56	1.21	8.68	1.31	30.53	1.38
Q17	1.64	1.26	3.02	10.96	19.74	20.95	40.74	12.28
mean	**1.75**	**1.53**	**3.74**	**2.41**	**11.00**	**7.56**	**50.55**	**17.55**

Table 3: Scale multiplier of the Relational and Tree-Based Approaches as compared to the 500MB times

tree-based approach experiences a large increase in execution time on the 8GB collection for each of these feature types. The relational approach on the other hand, has a very low scale-up as the collection size increases. The relational approach outperforms the tree-based approach for all of these features. In addition, the scaling multiplier increases at a much lower for the relational approach. The relational approach took between 0.67 and 1.47 times as long to execute the 8GB queries than the 500MB queries. In contrast, the tree-based approach took between 30.53 and 34.35 times as long to execute the 8GB queries.

Quantifier expressions ((c) in both figures) are the only feature that the tree-based approach outperformed the relational approach for most of the collections. The relational approach outperformed the tree-based approach on the 8GB collection. The tree-based approach could not complete execution of the existential quantifier query. Even though the tree-based approach outperformed the relational approach in terms of execution, the relational approach scales better than the tree-based approach. For example, the relational approach took 52.37 seconds to execute Q6 on the 2GB collection. The tree-based approach outperforms the relational approach with a time of 18.97 seconds. However, the scaling multiplier of Q6 on the 2GB collection is 1.69 with the relational approach as compared to 9.34 for the tree-based approach.

Regular expressions ((d) in both figures) experience similar behavior to other query feature features. The different types of regular expressions used are unknown sub-path (`/A//C`) and unknown element (`/A/*/C`). The timing differences between each type of regular expression differed very little with both approaches, showing that they are most likely handled similarly internally. The relational approach outperforms the tree-based approach on all collections both

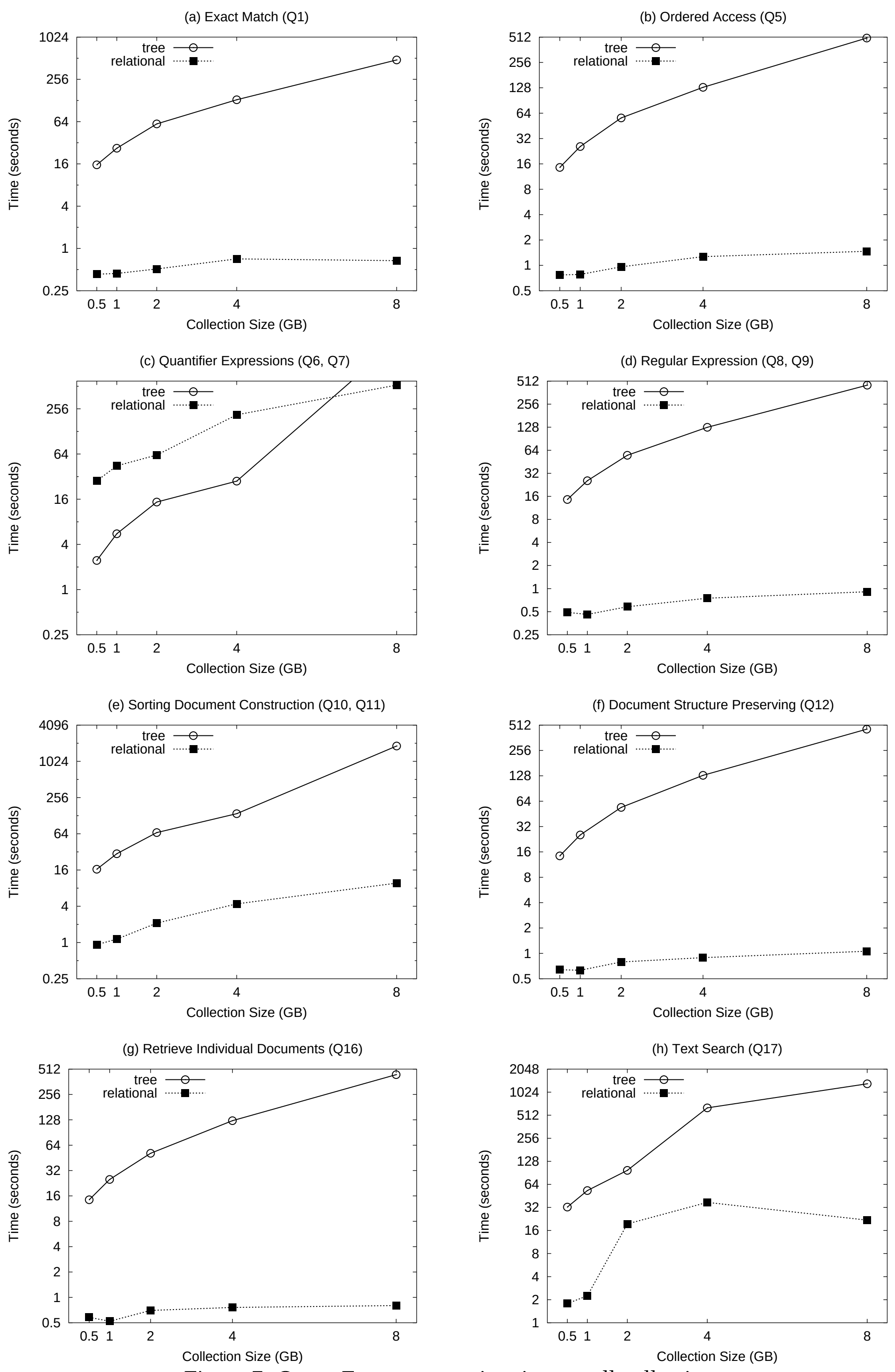

Figure 7: Query Feature execution time on all collections

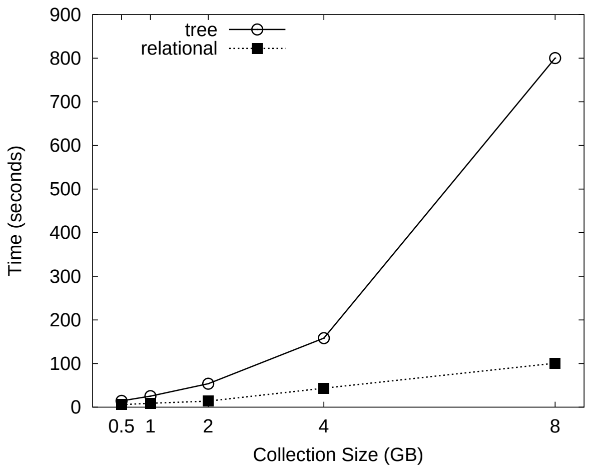

Figure 8: Average execution time on all collections

in terms of scaling multipliers and execution time. In one example, Q8 runs in 1.10 seconds on the 8GB collection using the relational approach. The tree-based approach, on the other hand, takes 478.43 seconds to execute.

Even though the generic data type of element content in XML documents is a string, users may cast the string type to other types. Therefore, an XML retrieval system should be able to efficiently sort values both in string and in non-string data types. The sorting document construction feature ((e) in both figures) tests the ability of each approach to sort by string and numeric value. Both queries return between 16 and 258 elements as the collections increase. It is interesting to note that the relational approach takes longer to sort by string type (5.97 seconds for the string type on the 4GB collection as opposed to 2.81 seconds for the numerical sort), while the tree-based approach experiences very little variation between the two types (139.53 and 137.04 seconds). Numeric sorting is faster with the relational approach since the database performs numerical comparisons faster than string comparisons. Even with the difference in time, the relational approach still outperforms the tree-based approach for both types of sorting document construction. The largest scale multiplier is 11.40 for the relational approach on the 8GB collection. In contrast, the tree-based approach has a scaling multiplier of 114.12.

Text search plays a very important part in XML search systems. The text search query searches through XML documents that contain between 5 and 80KB of free text. The text search feature ((h) in both figures) returns between 6 and 44 elements as the collection size increases. We see some odd behavior exhibited by the relational approach for this query on the 2GB and 4GB collections. Although the relational approach outperforms the tree-based approach in terms of execution time, the tree-based approach scales better on the 2GB and 4GB collections. This query is an example of the database choosing the wrong index and will be explained further in the following section.

The underlying structure is a tree with the tree-based approach and sets with the relational approach. In addition, the relational approach uses trees to retrieve elements from the relational sets. The highly optimized relational sets resulted in a much lower than expected scaling multiplier for the majority of the query features. In contrast, the tree-based approach scaled much closer to our expectations on the first four collections, but performed very poorly on the 8GB collection.

In Figures 8 and 10, we plot the average execution time over all collections as well as the average scale-up over all collection. Overall, the relational approach outperformed the tree-based approach on all five collections. The tree-based approach outperformed the relational approach for both quantifier queries, however, the relational approach scales better for these queries. On average, the relational approach took 17.5 times as long to execute the 8GB queries than the 500MB queries. On the other hand, the tree-based approach took 50.55 times as long to execute the queries on the 8GB collection.

5.1.1 Anomalous Queries

Both the tree-based and relational approaches experience variations in time for Q6 and Q7. Both Q6 and Q7 include quantified expressions which test for the existence of elements that satisfy a condition. Q6 uses existential quantifica-

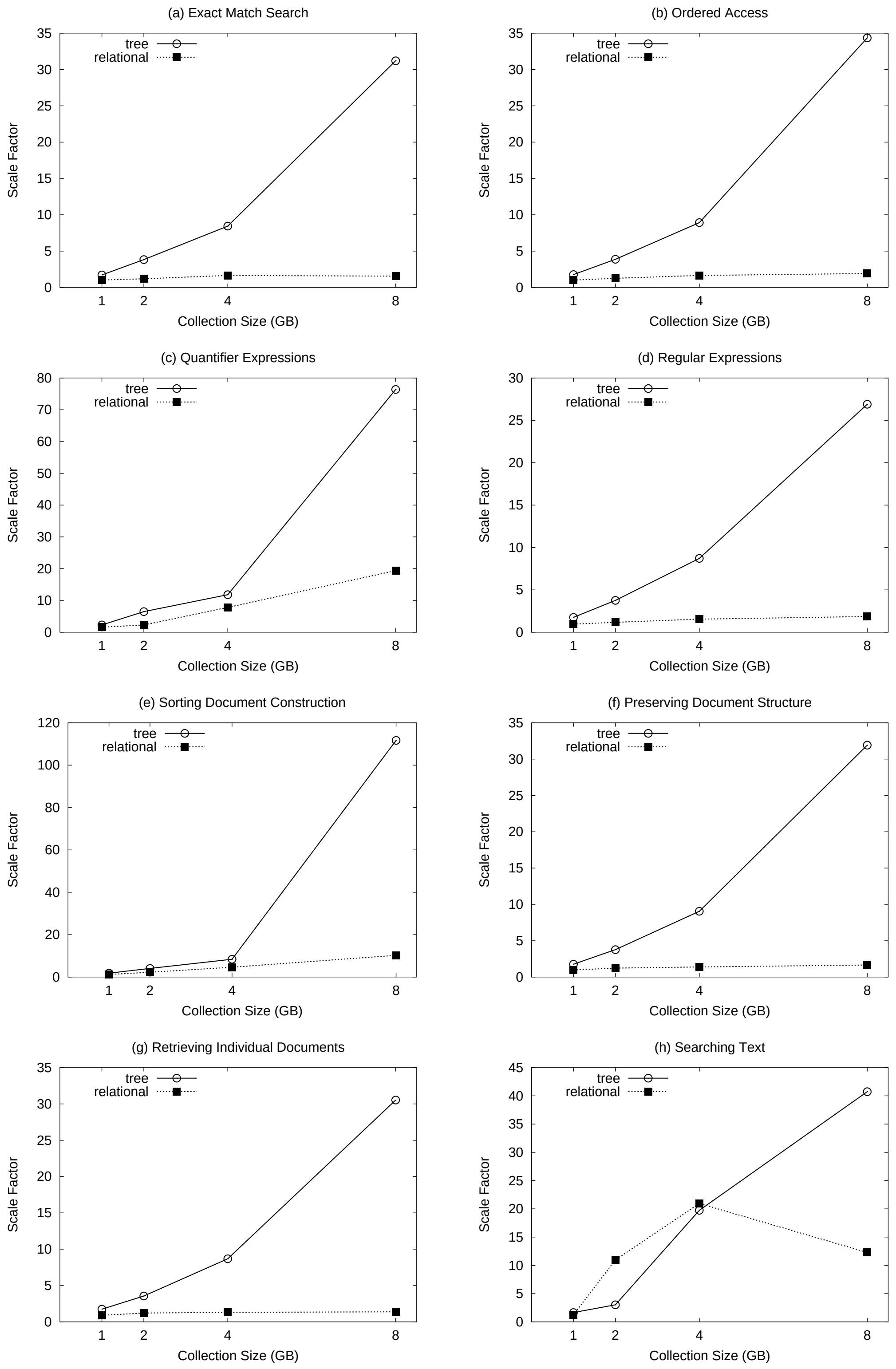

Figure 9: Query Feature Scale-up

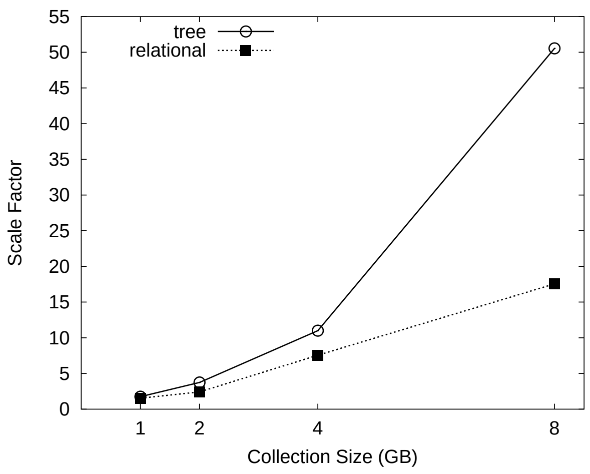

Figure 10: Average Scale-up

tion while Q7 uses universal quantification. An existential quantifier is evaluated to be true if some matching elements satisfy a specified condition. Universal quantifiers are evaluated to be true if all matching elements satisfy the specified condition. The tree-based approach executes both queries faster than any other query, however, the relational approach takes more time than any other query to execute both queries. Although the tree-based approach outperforms the relational approach in terms of execution time, the relational approach has a lower scaling multiplier for these queries as the collection size increases. The relational approach requires multiple calls to the database to process quantified expressions while the tree-based approach performs a fast check on the nodes used in the quantified expression allowing quantified expressions to be executed quickly.

The relational approach shows some odd behavior when running Q6 on the 1GB and 2GB collections. The query execution time for Q6 is close for both collections. Furthermore, Q6 experiences a large increase in execution time on the 8GB collection. Note that Q7 does not have the same problem as Q6. Q7 is near linear on all collections, however, there is an unusual increase in time for the 4GB and 8GB collections. The result set returned by Q7 is large. As the query is processed, the results are read from the database and momentarily exceed the size of memory causing memory swapping. While this is the only query that incurs swapping with the relational approach, several queries using the tree-based approach heavily use memory swapping. We observed memory swapping while informally monitoring (with the Unix utility, top) the memory and processes used during query execution.

Another query that exhibits odd behavior for the relational approach is Q17. This query takes much longer to execute on the 2GB and 4GB collections than it should. The database query optimizer first estimates the number of rows it will need to read to satisfy the query using each possible index. Then, it selects the index with the fewest number of rows. Occasionally, the MySQL optimizer will incorrectly estimate the number of rows that match specific values. Q17 is an example of the optimizer incorrectly estimating the number of rows, causing the wrong indexes to be used. We looked at the optimizer for Q17, it showed that the (*tagpath, parent*) and (*tagpath, pinnum*) indexes were being used when, in fact, the (*tagpath, nvalue*) and (*tagpath, parent*) indexes should be used. If we modified the SQL query to force the correct indexes on the 2GB and 4GB databases, the query would run in 9.14 and 14.76 seconds respectively (as compared to 19.50 and 37.29 seconds without forcing the index).

5.2 Large Result Sets

In addition to the XBench queries, we ran some queries specifically designed to measure large result sets. We use five queries that return increasingly large result sets. In Table 4, we provide the timings of each query on each collection. Also shown are the sizes of each query. In Figure 11, we plot the performance of each query as the result size increases as well as the average performance over the average result size.

Both R1 and R2 (Figure 11(a-b)) scale as the result set size increases. The scaling multiplier for the relational approach is however lower. R1 has a scaling multiplier of 7.09 for the relational approach and 10.17 for the tree-based approach. R2 has a scaling multiplier of 7.98 for the relational approach and 12.49 for the tree-based approach.

	Collection Size														
	500MB			*1GB*			*2GB*			*4GB*			*8GB*		
Type	tree	rel	size	tree	rel	size	tree	rel	size	tree	rel	size	tree	rel	size
R1	24.77	1.33	12,751	42.67	1.86	25,502	73.89	2.69	50,405	116.97	4.74	101,398	251.86	9.43	203,382
R2	15.64	1.42	12,751	28.47	1.87	25,502	60.83	3.27	63,992	89.82	5.00	101,459	195.41	11.33	206,780
R3	17.23	4.25	1,805	168.94	6.99	3,489	74.06	13.30	7,102	169.84	20.96	14,067	1,740.61	44.33	28,312
R4	30.72	14.61	2,800	72.65	27.88	5,516	156.91	59.41	11,106	882.91	124.40	22,051	MEMORY	299.53	44,228
R5	6.41	1.40	2,914	14.43	2.50	5,771	32.83	10.94	12,731	46.65	15.62	23,245	152.24	27.56	47,331
mean	**18.95**	**4.60**	**6,604**	**65.43**	**8.22**	**13,156**	**79.70**	**17.92**	**29,067**	**261.24**	**34.14**	**52,444**	**585.03**	**78.44**	**106,007**

Table 4: Total Query Time (seconds) for Queries with large result sets

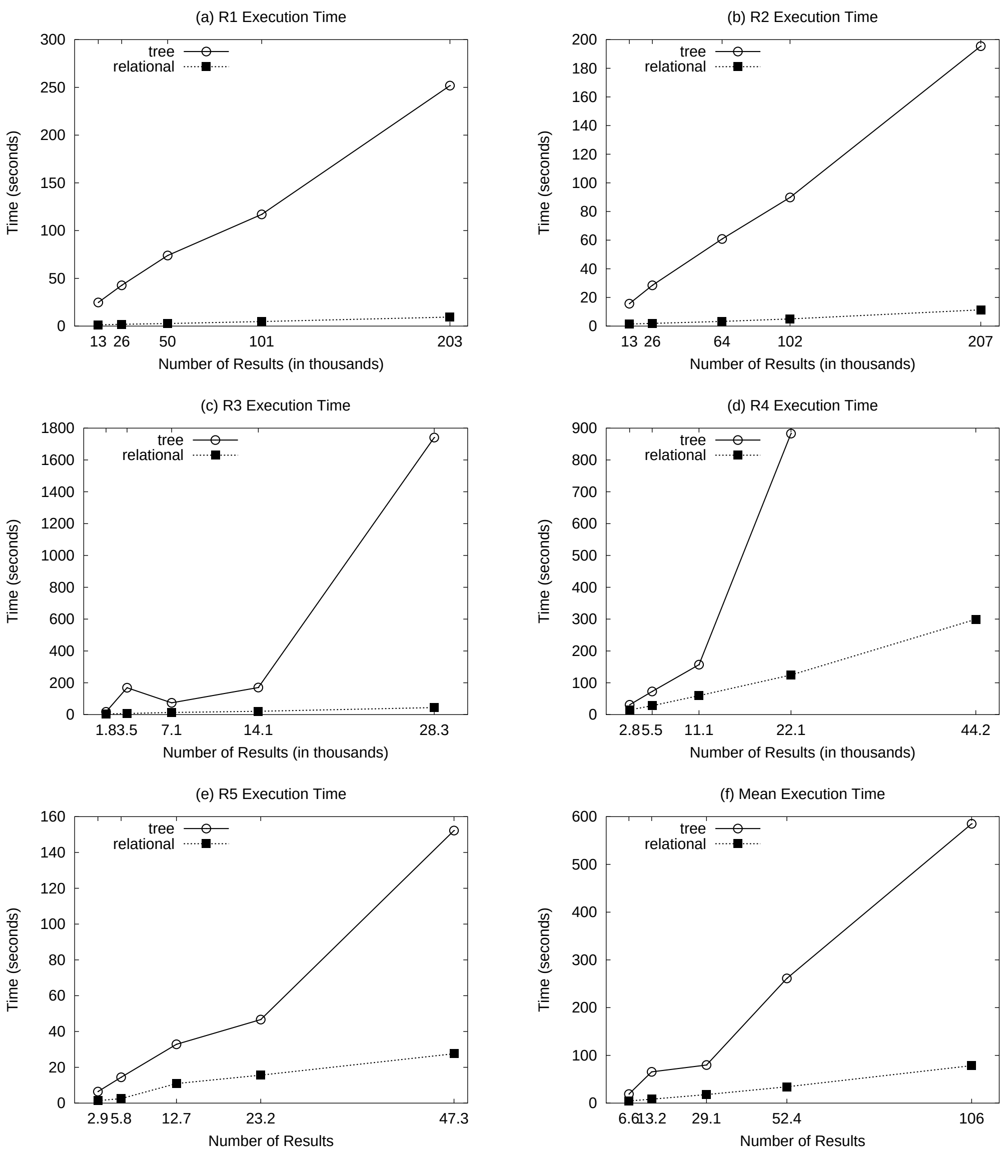

Figure 11: Query Execution time for Large Result Sets

Note that R1 and R2 have the largest result set sizes, yet R3, R4, and R5 (Figure 11(c-e)) take more time to execute. R1 and R2 return all the elements or attributes that match a given path, while R3, R4, and R5 add conditions to the query. These queries contain predicates which take more time to process for both approaches. As the result set size increases, R4 runs out of memory for the tree-based approach and fails on the 8GB collection. Although the size of the results returned from R4 are less than for the other queries, the node sets generated contain a large number of nodes causing the query to run out of memory. The relational approach's execution time outperforms the tree-based approach for all queries, however, R5 scales better for two of the result set sizes using the tree-based approach. For example, the relational approach has a scaling multiplier of 11.16 for R5 with 22,051 results while the tree-based approach has a better scaling multiplier of 7.28. All other queries scale better with the relational approach.

In Figure 11(e) we plot the average performance over the average result set size. As the result set size grows, the relational approach outperforms the tree-based approach for both query execution time and scaling multiplier. On average, the scaling multiplier on the largest result size is 17.05 for the relational approach which is much better than the tree-based approach's scaling multiplier of 30.87.

.

5.3 Multiple Predicate Matching

Finally, we examine the time to process queries with multiple predicates. Predicates are processed with the relational approach through the use of self joins. The tree-based approach generates more node sets, which are later joined in the path join algorithm. By looking at the performance of each approach as we increase the number of predicates, we can get a view of the cost of self joins versus the cost of performing the path join algorithm on more node sets.

We created four queries with one through four predicates. In Table 5, we show the time to execute each query on the 2GB collection. In Figure 12, we plot the performance of each approach as the number of predicates increase.

	Number of Predicates			
	1	2	3	4
relational	0.84	1.04	1.21	1.41
tree	57.73	93.96	139.76	182.83

Table 5: Total Query Time (seconds) for Predicates on 2GB collection

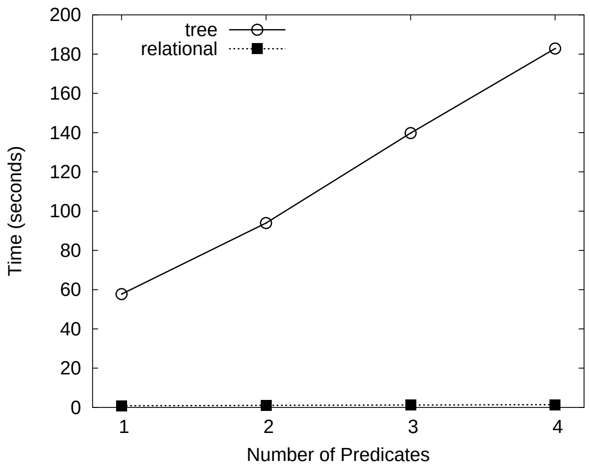

Figure 12: Query Execution time for large number of predicates

Both approaches scale linearly as the number of predicates increase, however, the relational approach outperforms the tree-based approach for all of the queries. Furthermore, as the number of predicates increase, the relational approach's execution time increases at a slower rate than the tree-based approach. The relational approach takes 1.67

times longer to execute a query with four predicates than one with one predicate. In contrast, the tree-based approach takes 3.17 times as long to execute a query with four predicates.

Our results indicate that as the number of predicates increase, the cost of relational joins is less than the cost of performing the path join algorithm in the tree-based approach.

6 Conclusions

Over the last several years, XML search has migrated from the interest of the few to the need of the many. Furthermore, XML collections previously considered large are now viewed as common-place. Thus, as always, necessity has spurred interest and innovation.

XML collections are viewed as either data or text-centric with data-centric collections containing a greater percentage of XML tags and often a greater depth in terms of the XML schema hierarchies. Our focus is on data-centric collections, and we compared the two common search approaches for XML search, namely relational database oriented and tree-based search systems.

We demonstrated the superior scalability of the relational approach for searching XML data over the tree-based approach. We generated collections of heterogeneous XML documents ranging in size from 500MB to 8GB using the commonly used XBench benchmark suite. The scalability of each method was tested by running XQuery queries that cover a wide range of XML search features on each collection. Additional queries were developed specifically for comparison between both approaches.

Our analysis shows that the relational approach is scalable to the collection size and the feature type. In addition, the relational approach is also scalable as the result set increases and for a varying number of predicates. Although complex joins for predicate matching and reconstructing large result sets do slow down the relational approach, it still provides a significant improvement over the tree-based search. Furthermore, our results support our claim that the relational mapping of XML queries not only leverages existing relational database optimizations, but typically outperforms standard tree-based indexing systems.

Future work involves further analysis of the relational and tree-based approaches. We plan to compare other methods of relational and tree-based approaches using multiple XML benchmarks. In addition, we also plan to expand our study to include text-centric data. Future work also involves extending this study to include a wider variety of XML search techniques over larger collections of XML documents. Studying different XML retrieval techniques on large collections of XML data may point to issues with currently used XML retrieval techniques.

As of recent, the SQLGenerator is in daily use, by dozens of users in each of dozens of places worldwide.

Acknowledgements

Special thanks to Fred Ingham, Tim Lewis, and Steve Condas for their assistance with the design and testing of various parts of the SQLGenerator.

References

[1] SQLGenerator. http://ir.iit.edu/projects/SQLGenerator.html.

[2] DBLP. http://www.informatik.uni-trier.de/ ley/db/.

[3] Extensible Markup Language (XML). http://www.w3.org/XML/.

[4] Open Source Native XML Database. http://exist.sourceforge.net/.

[5] XBench - A Family of Benchmarks for XML DBMSs. http://db.uwaterloo.ca/ ddbms/ projects/xbench/index.html.

[6] INitiative for the Evaluation of XML Retrieval (INEX), 2007. http://inex.is.informatik.uni-duisburg.de/2007/.

[7] L. Afanasiev and M. Marx. An Analysis of the Current XQuery Benchmarks. In *International Workshop on Performance and Evaluation of Data Management Systems (EXPDB)*, 2006.

[8] T. Amagasa, M. Yoshikawa, and S. Uemura. QRS: A Robust Numbering Scheme for XML Documents. *Proceeding of the 19th International Conference on Data Engineering (ICDE'03)*, 2003.

[9] S. Amer-Yahia and M. Lalmas. XML Search: Languages, INEX and Scoring. *SIGMOD Record*, 35(4):16–23, 2006.

[10] K. S. Beyer, R. Cochrane, V. Josifovski, J. Kleewein, G. Lapis, G. M. Lohman, B. Lyle, F. Ozcan, H. Pirahesh, N. Seemann, T. C. Truong, B. V. der Linden, B. Vickery, and C. Zhang. System RX: One Part Relational, One Part XML. In *SIGMOD Conference*, pages 347–358, 2005.

[11] P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In *SIGMOD international conference on Management of data*, pages 479 – 490, 2006.

[12] J. Bosak. Shakespeare XML Collection. http://www.ibiblio.org/bosak/xml/eg/.

[13] J. Bremer and M. Gertz. Integrating document and data retrieval based on XML. *The VLDB Journal*, 15(1):53–83, 2006.

[14] T. Bhme and E. Rahm. Supporting Efficient Streaming and Insertion of XML Data in RDBMS. In *3rd International Workshop Data Integration over the Web (DIWeb)*, 2004.

[15] R. Cathey, S. Beitzel, E. Jensen, D. Grossman, and O. Frieder. Relationally Mapping XML Queries for Scalable XML Search. *Proceeding of IEEE conference on the Intelligence and Security Informatics (ISI'07)*, May 2007.

[16] S. Chien, V. Tsotras, C. Zaniolo, and D. Zhang. Efficient Complex Query Support for Multiversion XML Documents. *Proceeding of the EDBT Conference*, 2002.

[17] L. Denoyer and P. Gallinari. The Wikipedia XML Corpus. *SIGIR Forum*, 40(1):64–69, 2006.

[18] W. Fan, J. X. Yu, H. Lu, J. Lu, and R. Rastogi. Query Translation from XPath to SQL in the Presence of Recursive DTDs. In *VLDB*, pages 337–348, 2005.

[19] D. Florescu and D. Kossman. A performance evaluation of alternative mapping schemes for storing XML data in a relational database. Technical report, INRIA, France, 1999.

[20] D. Florescu and D. Kossman. Storing and Querying XML Data using an RDBMS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[21] G. Gottlob, C. Koch, R. Pichler, and L. Segoufin. The Complexity of XPath Query Evaluation and XML Typing . In *Journal of the ACM*, volume 52, pages 284–335, 2005.

[22] M. Grohe. Parameterized Complexity for the Database Theorist. In *ACM SIGMOD Record*, volume 31, pages 86–96, 2002.

[23] T. Grust. Accelerating XPath Location Steps. In *SIGMOD international conference on Management of data*, pages 109–120, 2002.

[24] T. Grust, S. Sakr, and J. Teubner. XQuery on SQL Hosts. In *VLDB*, pages 252–263, 2004.

[25] H. Jiang, H. Lu, W. Wang, and J. X. Yu. Path Materialization Revisited: An Efficient Storage Model for XML Data. *Proceedings of the Thirteenth Australian Conference on Database Technologies*, 5:85–94, 2002.

[26] L. Khan and Y. Rao. A Performance Evaluation of Storing XML Data in Relational Database Management Systems. *Proceeding of the third international workshop on Web information and data management* , pages 31–38, 2001.

[27] C. Koch. On the Complexity of Nonrecursive XQuery and Functional Query Languages on Complex Values. In *ACM Transactions on Database Systems*, volume 31, pages 1215–1256, 2006.

[28] C. Koch. Processing Queries on Tree-Structured Data Efficiently. In *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 213–224, 2006.

[29] R. Krishnamurthy, V. Chakaravarthy, R. Kaushik, and J. Naughton. Recursive XML Schemas, Recursive XML Queries, and Relational Storage: XML-to-SQL Query Translation. *IEEE International Conference on Data Engineering (ICDE)*, 2004.

[30] R. Krishnamurthy, R. Kaushik, and J. Naughton. XML-to-SQL Query Translation Literature: The State of the Art and Open Problems, September 2003.

[31] M. Krishnaprasad, Z. H. Liu, A. Manikutty, J. W. Warner, V. Arora, and S. Kotsovolos. Query Rewrite for XML in Oracle XML DB. In *VLDB*, pages 1122–1133, 2004.

[32] Y. Lee, S. Yoo, K. Yoon, and P. Berra. Index Structures for Structured Documents. *Proceeding of the 1st ACM International Conference on Digital Libraries*, March 1996.

[33] Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. *Proceeding of the 27th International Conference on Very large Databases*, September 2001.

[34] Z. H. Liu, M. Krishnaprasad, and V. Arora. Native Xquery processing in oracle XMLDB. In *SIGMOD Conference*, pages 828–833, 2005.

[35] S. Manegold. An Empirical Evaluation of XQuery Processors. In *International Workshop on Performance and Evaluation of Data Management Systems (EXPDB)*, 2006.

[36] W. Meier. eXist: An Open Source Native XML Database. *Web, Web-Services, and Database Systems. NODe 2002 Web- and Database-Related Workshops*, October 2002. Springer LNCS Series, 2593.

[37] W. Meier. Index-Driven XQuery Processing in the eXist XML Database. In *IXML Prague*, 2006.

[38] S. Pal, I. Cseri, G. Schaller, O. Seeliger, L. Giakoumakis, and V. V. Zolotov. Indexing XML Data Stored in a Relational Database. In *VLDB*, pages 1134–1145, 2004.

[39] C. H. Papadimitriou and M. Yannakakis. On the Complexity of Database Queries. In *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 12–19, 1997.

[40] M. Rys. XQuery in Relational Database Systems. In *XML 2004*, 2004.

[41] M. Rys. XML and relational database management systems: inside Microsoft SQL Server 2005. In *SIGMOD international conference on Management of data*, pages 958 – 962, 2005.

[42] S. S. Al-Khalifa, H. Jagadish, N. Koudas, J. Patel, D. Srivastava, and Y. Wu. Structural joins: a primitive for efficient XML query pattern matching. *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 141–152, 2002.

[43] A. Schmidt, M. Kersten, M. Windhouwer, and F. Waas. Efficient Relational Storage and Retrieval of XML Documents. *Lecture Notes in Computer Science*, 1997:137+, 2001.

[44] J. Shanmugasundaram, J. Kiernan, E. J. Shekita, C. Fan, and J. E. Funderburk. Querying XML Views of Relational Data. In *VLDB*, pages 261–270, 2001.

[45] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational database for querying XML documents: Limitations and Opportunities. In Proc. of VLDB, 1999.

[46] J. Snelson. All XML Databses are Equal. *XTech 2005: XML, the Web and beyond*, 2005.

[47] D. Suciu. On database theory and XML. *ACM SIGMOD Record archive*, 30(3):39–45, 2001.

[48] N. Suizo. XML Propels Security Intelligence. *Network World*, August 2006.

[49] I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 204–215, 2002.

[50] F. Tian, D. DeWitt, J. Chen, and C. Zhang. The Design and Performance Evaluation of Alternative XML Storage Strategies. *ACM SIGMOD Record*, 31(1):5–10, 2002.

[51] A. Vakali, B. Catania, and A. Maddalena. XML Data Stores: Emerging Practices. *IEEE Internet Computing*, 9(2):62–69, 2005.

[52] H. Wang and X. Meng. On the Sequencing of Tree Structures for XML Indexing. In *ICDE*, pages 372–383, 2005.

[53] F. Weigel, K. U. Schulz, and H. Meuss. Exploiting Native XML Indexing Techniques for XML Retrieval in Relational Database Systems. In *Workshop On Web Information And Data Management (WIDM'05)*, pages 23–30, 2005.

[54] F. Weigel, K. U. Schulz, and H. Meuss. The BIRD Numbering Scheme for XML and Tree Databases - Deciding and Reconstructing Tree Relations Using Efficient Arithmetic Operations. In *XSym*, pages 49–67, 2005.

[55] G. Xing and B. Tseng. Extendible Range-Based Numbering Scheme for XML Document. *Proceeding of the International Conference on Information Technology: Coding and Computing (ITCC'04)*, 2004.

[56] M. Yoshikawa and T. Amagasa. XRel: A Path-Based Approach to Storage and Retrieval of XML Documents Using Relational Databases. *ACM Transactions on Internet Technology (TOIT)*, 1(1):110–141, August 2001.

[57] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, and G. Lohman. On supporting containment queries in relational database management systems. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 425–438, 2001.