The *RankGroup* Join Algorithm: Top-*k* Query Processing in XML Datasets

Sunetra Saha, Jianmei Fan, Nicholas Cohen, and Russell Greenspan
University of Illinois
April, 2004

ABSTRACT

This project investigates top-*k* queries in XML datasets. We propose a syntactical addition to XQuery to accommodate top-*k* XML queries. We then propose a 3-step process to realize these top-*k* XML queries using a relational database and a new join operator, *RankGroup*. Our preliminary implementation shows promise in dramatically reducing the running time and number of tuples accessed during such query processing.

1. INTRODUCTION

Top-*k* query processing has become an essential part of the internet today and extensive work has been done to optimize the speed of such ranked queries. XML's format is ideally suited to the web, and there has been a rapid increase of XML documents on the web, thus ensuring that we have countless top-*k* queries directed at XML documents.

To facilitate running top-*k* queries over XML datasets, we present an extension to the XML Query Language XQuery. XQuery is designed to be flexible enough to query a broad spectrum of XML information sources, including both databases and documents, and presented a good fit for our top-*k* syntactical additions.

We further present a procedure for processing such top-*k* XML queries. The procedure involves three steps:

1. Shredding the XML data to relational tables for Relational Database (RDBMS) storage and convenient access
2. Converting the top-*k* XQuery queries into top-*k* relational queries
3. Running a new join operator, *RankGroup*, over the relational tables to compute the top-*k* results. *RankGroup* is designed to take into account the inherent structure of the XML document and return results as they are processed

The rest of this paper is organized as follows. Section 2 describes related work in the field of XML in relational databases and top-*k* relational queries. Section 3 describes our proposal for syntactical additions to XQuery to support top-*k* queries. Section 4 describes each of the three steps outlined above in great detail, including implementation particulars and the steps involved along the way. In Section 5, we outline our benchmark XML dataset and our top-*k* benchmark queries. In Section 6 we show initial experimental results using our *RankGroup* join technique and compare its performance to alternative techniques.

2. RELATED WORK

2.1 Structured XML Queries in RDBMS

[5] describes traditional relational database engines for processing XML documents conforming to Document type Descriptors (DTDs). They reveal a number of limitations in current RDBMS's that in some instances make use of relational technology for XML queries either awkward or inefficient.

[1] proposes algorithms that allow an RDBMS to efficiently perform XML containment and proximity queries. The algorithms utilize an inverted list system for converting XML documents to and from relations and are complementary to our work.

2.2 Top-*k* Relational Queries

[3] presents an extension to the Ripple join operator for processing top-*k* queries that allows the system to avoid a final sort (generally necessary to pick the top-*k* results) by ensuring that results are sorted in the desired final sort order along the way. The algorithm takes *m* ranked inputs, a join condition, a monotone combining ranking function and the number of desired ranked join results *k*. The algorithm reports the top-*k* ranked join results in descending order of their combined score. Our *RankGroup* algorithm closely follows this technique.

In a middleware environment, [4] introduces the first efficient set of algorithms (FA) to answer ranking queries. Later they introduced the TA algorithm which uses a threshold value to improve the cost. The NRA and CA algorithm is used for the scenario where random access is forbidden or expensive.

3. TOP-*K* XML QUERY SYNTAX

3.1 Introduction

In order to apply top-*k* queries to XML data, a few expressions must be defined. XQuery (http://www.w3.org/TR/xquery/) has two constructs which are of interest to top-*k* queries. The *for* expression supplies output from an XML input in a similar manner as a recordset supplies output from an SQL query. The *for* expression provides a method of ordering output data with the *order by* clause. Since XQuery provides a manner to define custom functions, top-*k* queries can be implemented by specifying a custom function to the *order by* clause in a *for* expression.

3.2 Definitions

XQuery has an expression called *for* that can iterate over a given input and generate a sequence of tuples as output.

```
 1 for $d in fn:doc("depts.XML")//deptno
 2 let $e :=
  fn:doc("emps.XML")//emp[deptno = $d]
 3 where fn:count($e) >= 10
 4 order by fn:avg($e/salary) descending
 5 return
 6    <big-dept>
 7       {
 8       $d,
 9
  <headcount>{fn:count($e)}</headcount>,
10
  <avgsal>{fn:avg($e/salary)}</avgsal>
11       }
12    </big-dept>
```

In the example above, *for* will iterate over all deptno in the depts..XML (line 1). For each iteration, the variable $d will represent the department of the current iteration, and the variable $e will be set to the number of employees in the department $d (line 2). Only departments with 10 or more employees will be included in the output (line 3). The final ordering of the tuples will depend on the average salary of all employees of the department (line 4). Finally, tuples of data will be generated in <big-dept> tags containing the relevant information from all iterations (lines 5-12).

In order to apply the *for* expression to the concept of top-*k* queries, the ability to define a function must be available. Luckily, XQuery provides syntax for defining functions.

```
 1 declare function local:summary($emps
  as element(employee)*)
 2    as element(dept)*
 3 {
 4    for $d in fn:distinct-
  values($emps/deptno)
 5    let $e := $emps[deptno = $d]
 6    return
 7       <dept>
 8          <deptno>{$d}</deptno>
 9          <headcount> {fn:count($e)}
  </headcount>
10          <payroll>
  {fn:sum($e/salary)} </payroll>
11       </dept>
12 };
13+
14
  local:summary(fn:doc("acme_corp.XML")//
  employee[location = "Denver"])
```

In the example above, a summary function is defined (lines 1-2). The definition entails both a function name and a namespace (local). The function one parameter $emps corresponds to the type (employee)* essentially meaning multiple employees. The function is composed of a *for* expression that returns the number of employees in each department and the average salary of the employees in the department.

The syntax for top-*k* queries in XQuery is basically the combination of the *for* expression, custom-defined function, and a new expression called *stop after$x*. Basically the *for* expression will stop producing results after it has produced the $x results. The syntax will generally look similar to the following:

```
let $x := 10

for input
  order by fn:f(...)
  stop after $x

define funciton fn:f(...) as ... {
  ...
}
```

4. THE XML SHREDDER

4.1 Introduction

Traditionally, the shredding of XML documents into relations is done in a lossless way with the help of keys to maintain the parent-child relation. As such, all elements with the same tag are stored in a different relational table, and the attributes of the element make up the columns of the table. Each element's parent ID is also stored to maintain the document's structure. But for XML queries, we can leverage the inherent tree structure of XML to potentially create inter-table indexes. This will make the parent-child relationship easier to navigate and query. We utilize a distinct method described in [2] of shredding the XML into blocks and storing each block's relative position by recording the attributes (StartPos, EndPos, Level). XML-specific queries can then be asked using the RDBMS because the comparable position of each element is attainable.

Checking for the presence of structural relationships in the XML tree, like ancestor-descendant and parent-child, amounts to checking that certain range conditions hold between the components of the positions of these elements. For example, to check if an element A contains element B, we would check the following condition:

```
WHERE (A.StartPos < B.StartPos AND
  B.EndPos < A.EndPos)
```

4.2 Purpose

For the purpose of evaluating the performance of top-*k* queries against a shredded XML database, a utility to perform the shredding was needed. After initial discussions it was decided that we wanted to shred the XML documents such that the structural information is recorded as the start position of the XML tag, its end position and the level of nesting of the element inside the document. This sort of information was important because this allowed us to do structural joins of the parent-child as well as the ancestor-descendant format. However none of the commercially available XML to relational converters seemed to shred the XML as we wanted. So we decided to develop our own version. In order to utilize the technical expertise in the group the shredder was written in java and as the database size was potentially huge the parsing was done with the Xerces Sax parser.

4.3 Design Decisions

As the structural information of the tags was important, but the actual position of the tag in the document was relatively unimportant, the start and end position was not the actual start and end positions in the document, but the count of the number of start and end tags in the document. This allowed us to perform our structural joins but saved considerable development effort.

We were implementing our algorithm as a stand alone piece of code, and not as part of an existing DBMS implementation. So our shredder did not need to actually insert the data in a DBMS, but we needed to make it available to our implementation of the new algorithm. (A similar implementation of an existing top-*k* algorithm was run against the same format for comparison.). After some discussions, we decided to use a Relation-Row format – with our shredded XML also presented in an XML format.

For an example of an XML document and its shredded equivalent, please see Appendix B.

4.4 Implementation/Architecture

Though it was initially decided that sax would be our parsing method of choice, as we went ahead with the implementation, we came across problems. This was because we needed to record the ending position for a particular element – which meant that we could not write out a row till its end has been reached, or the row information of an element would have to be stored till the element came to an end. Also we needed to keep all rows of a particular relation at one place and could not just write out rows as and when we encountered the elements in the document. Dom parsing seemed to be implied here or a sax based memory intensive version. Both versions were developed and experimented with till the problems became more apparent as the size of the XML document increased. In order to counter this problem, we decided to store rows of individual relations into separate files during parsing and then join them together after parsing is complete – to save memory intensive processing. Of course this resulted in slower performance, but as performance was not a very important side of the shredding we decided to go ahead with this implementation.

4.4.1 System Requirements/Usage

To run this shredder, we use the java runtime environment and the Xerces for java library files. The shredder consists of a single file called Shredder.java, which is complied to produce Shredder.class.

If the Classpath is set to include both the Shredder.class directory and the xerces jar files, the Shredder can be run with the command:

```
java Shredder <input file name>
```

If the input file is of a reasonable size, the application would create a number of files of the name db_<tag name>, which would be deleted once processing is complete. The output would be stored in a file called Shredded_<input_file_name> in the runtime directory.

4.5 Converting The Xml Queries To Relational Queries

The basic procedure to convert a XQuery into an SQL query involving structural joins is as follows: the XQuery will basically consist of a *for* construct with an *order by* clause. The *for* construct will iterate over a certain set of parent elements and select certain elements from the XML tree. These elements correspond to items in a relational table with name equal to the name of the element. The *order by* clause will order the *for* output with respect to the relevant function.

Since our queries all involve structural joins between elements, the start position and end element position are utilized in the SQL query to efficiently infer existence of structural (ancestor-decendant or parent-child) relationships. As XQuery is a higher abstraction, it does not involve the use of start and end element position. The "order by" clause in each query is essentially identical. *Item* is included in each of the SQL FROM clauses because it is the parent element and will be returned as output from each query. The other relations in the FROM clause are inferred from each element contained in the query.

5. THE *RANKGROUP* JOIN ALGORITHM

5.1 Theory

Once we have the XML shredded to relational tables and the top-*k* XML queries converted to SQL queries over the shredded data, the final step is to process the SQL queries. Consider the following simple query that returns the cheapest items by combined price:

```
SELECT TOP-k
FROM A, B
WHERE A.Type = B.Type
ORDER BY A.Price + B.Price
```

There are generally two ways to approach this query. The first is to join A and B on *A.Type = B.Type*, sort on *A.Price + B.Price* after the join has been realized, and then return just the top *k* from this resultset. This is potentially expensive because many of the processed joins could be at a very expensive combined price, and would therefore not be reflected in the top-*k*. This also requires a sort, another potentially expensive operation.

The second approach is to consider A and B sorted by *Price* and attempt to join on *A.Type = B.Type* in the order of *Price*. This ensures that joins are processed in the final sort order (*A.Price + B.Price*), avoiding any final sort operation. Simultaneously, we know that any successful join belongs in the top-*k* and we can stop joining after *k* joins.

5.2 The *RankGroup* Algorithm

The problem, then, is how to consider the tuples from each relation while ensuring that the final sort order is kept during join processing. Consider the following data for (*Price*, *Type*) in *A* and *B* in the required sorted order:

```
A {(5, 2), (5, 4), (6, 1), (8, 1), (8,
3)}
B {(3, 1), (3, 2), (5, 2), (6, 3), (6,
4)}
```

The technique we use, nicknamed *RankGroup*, is very similar to the Ripple join variation described in [3]. We first attempt joins on the minimally priced items in *A* with the minimally priced items in *B*. This is *A*{(5, 2), (5, 4)} and *B*{(3, 1), (3, 2)}, where *A.Price + B.Price* = 8. After processing joins within this group on *A.Type = B.Type* (finding a successful match on (5, 2) from *A* and (3, 2) from *B*), we need to intelligently select the next rank groups to join within.

To do this, we keep a priority queue of the possible next-best group combinations. In this example, we would add A{(6, 1)} and B{(3, 1), (3, 2)}, where *A.Price + B.Price* = 9, and A{(5, 2), (5, 4)} and B{(5, 2)}, where *A.Price + B.Price* = 10, to our queue. Since the queue is sorted by this combined ranking, we are ensured that joins are processed in order of minimal price point, which is our desired final sort order.

The pseudocode for this algorithm:

```
Initialize Queue;
Add item to queue for first RankGroup
from each relation;

WHILE total output rows < k OR queue is
empty
{
      Pop the first item off the queue;
      Process all joins within the sets
of the item popped;

      Add item to queue for current left
and next right;
      Add item to queue for next left and
current right;
}
```

For an illustrative example of the algorithm in action, please see Appendix C.

5.3 Distinction from Ripple Join

Such a technique is commonly used in the family of joins known as Ripple joins, specifically [3]. However, our technique involves an important distinction: using a multi-dimensional index, we are able to perform a merge-sort join within each Rank group, whereas Ripple join would have performed a nested-loop join. This yields dramatically different performance on large datasets.

To realize this approach, we consider that the RDBMS system keeps multi-dimensional indexes on all attributes involved in the query (*Price* and *Type* in the previous example). To be sure, keeping such indexes for every possible combination of attributes is unfeasible in a real-world situation, but for the limited scope of this algorithm we ignore this issue.

5.4 Additional Uses

The algorithm can be used with indexes sorted in a descending direction as needed to achieve a final sort in descending order (i.e. [*A.Price + B.Price*] DESC).

Similarly, the algorithm can be used to compute the maximum difference between two columns (i.e. [*A.Price - B.Price*] DESC, which can be viewed as [A.Price + (– B.Price)] DESC). To do this, we consider *A* in *Price* descending order and *B* in *Price* ascending order. The result is that as the algorithm proceeds, the maximum difference between the two columns is considered at each step.

5.5 Implementation

Our initial proof-of-concept implementation was written in C# and works as a stand-alone application. It takes input from flat files and computes the desired query using four custom-implemented join techniques: traditional Nested-Loop, traditional Merge-Sort, Ripple, and *RankGroup*.

The C# implementation relies on two C# data structure classes, *Data.DataTable* and *Collections.SortedList*. *Data.DataTable* can really be thought of just as a two-dimensional array (matrix) with convenient *Row[RowID]["ColumnName"]* access to the data. The *Collections.SortedList* class is more complex than a simple priority queue we would prefer to use, but offers the needed functionality of keeping the cheapest item at the front of the list.

5.6 Isolated Experimental Results

To test the implementation, the simple query described above was run over varying sized datasets with different values for *k* from 10 to 10,000. For this simple test, only the running time in milliseconds was measured. The test machine was a 3Ghz Pentium 4 with 1GB RAM.

As can be seen from the results shown in Appendix D, *RankGroup*'s performance is more or less flat as the number of tuples increases, whereas the traditional nested-loop and merge join perform horribly as the size of input relations increases. This means that *RankGroup*'s performance is the same for top-*50* joins with 100 x 100 tuples (i.e., 100 tuples in Relation A and 100 tuples in Relation B) as for 10,000 x 10,000 tuples. This makes sense intuitively, as the number of tuples in each relation has no effect on the *RankGroup* algorithm. However, the traditional join must first perform the 10,000 x 10,000 tuple join and then sort, yielding expensive operations.

Another interesting note is that as *k* increases, there is a slight decrease in *RankGroup*'s performance while traditional joins remain the same. This allows traditional joins to outperform *RankGroup* for very small numbers of tuples and large *k*.

6. BENCHMARK DATASET AND QUERIES

6.1 Benchmark Dataset: XBench

XBench (http://db.uwaterloo.ca/~ddbms/projects/xbench) is a family of benchmarks that capture different XML application characteristics. The data-centric schema is an e-commerce catalog of available books, the books' authors, the books' pricing and cost, etc. See Appendix A for the complete schema of the XBench dataset used.

XBench was chosen as the dataset because it is relatively deep (lots of parent-child and anscestor-decendant relationships) when compared to other benchmarks available. Even though XBench is relatively deep, it may not necessarily be deep enough to fully exploit the features of XML. In hindsight, we decided that perhaps generating our own benchmark dataset would have been ideal, allowing us to tailor the dataset to be more XML friendly. This was not completed due to time constraints but would be added in the future.

6.2 Benchmark Queries

The following are examples of queries over the data-centric schemas defined by XBench.

6.2.1 Query #1: Items that were available and released the longest ago

XQuery Syntax

```
1   let $q := 10
2
3   for $i in inputItems
4     let $id = $i/id
5     let $d := $i//date_of_release
6     let $w := $i//when_is_available
7     order by fn: score-available-
    long-ago($d,$w,$h) ascending
8     stop after $q
9     return
10      <item id=$id>
11      </item>
12
13  declare function fn:score-
    available-long-ago (
14
    $d as xs:date,
15
    $w as xs:date)
16
    as xs:decimal {
17    return ($w - $d)
18  }
```

SQL Syntax

```
19  SELECT i.id
20  FROM ITEM i, DATE_OF_RELEASE d,
    WHEN_IS_AVAILABLE w
21  WHERE (i.StartPos < d.StartPos AND
    d.EndPos < i.EndPos)
22    AND (i.StartPos < w.StartPos AND
    w.EndPos < i.EndPos)
23  ORDER BY (w.Text + d.Text)
```

This query uses multi-dimensional indexes:

```
DATE_OF_RELEASE(Text, StartPos, EndPos)
WHEN_IS_AVAILABLE(Text, StartPos, EndPos)
```

6.2.2 Query #2: Bulkiest items

XQuery Syntax

```
1   let $q := 10
2
3   for $i in inputItems
4       let $id = $i/id
5       let $l := $i//length
6       let $w := $i//width
7       let $h := $i//height
8       order by fn:score-bulky-items
    ($l,$w,$h) ascending
9       stop after $q
10    return
11      <item id=$id>
12      </item>
13
14  declare function fn:score-bulky-
    items($l as xs:decimal,
15
    $w as xs:decimal
16
    $h as xs:decimal)
17
    as xs:decimal {
18    return ($l + $w + $h)
19  }
```

SQL Syntax

```
20 SELECT i.id
21 FROM ITEM i, LENGTH l, WIDTH w,
   HEIGHT h
22 WHERE (i.StartPos < l.StartPos AND
   l.EndPos < i.EndPos)
23   AND (i.StartPos < w.StartPos AND
   w.EndPos < i.EndPos)
24   AND (i.StartPos < h.StartPos AND
   h.EndPos < i.EndPos)
25 ORDER BY (l.Text + w.Text +
   h.Text)$user-page := 100
```

This query uses multi-dimensional indexes:

```
LENGTH(Text DESC, StartPos, EndPos)
WIDTH(Text DESC, StartPos, EndPos)
HEIGHT(Text DESC, StartPos, EndPos)
```

6.2.3 Query #3: Items with greatest profit margin

XQuery Syntax

```
1  let $q := 10
2
3  for $i in inputItems
4    let $id = $i/id
5    let $p :=
   $i//suggested_retail_price
6    let $c := $i//cost
7    order by fn:score-greatest-
   profit($s,$c) ascending
8    stop after $q
9    return
10        <item id=$id>
11        </item>
12
13    declare function fn:score-bulky-
   items($p as xs:decimal,
14
   $c as xs:decimal)
15
   as xs:decimal {
16      return ($p - $c)
17    }
```

SQL Syntax

```
18 SELECT i.id
19 FROM ITEM i, SUGGESTED_RETAIL_PRICE
   p, COST c
20 WHERE (i.StartPos < p.StartPos AND
   p.EndPos < i.EndPos)
21   AND (i.StartPos < c.StartPos AND
   c.EndPos < i.EndPos)
22    ORDER BY (p.Text - c.Text) DESC
```

This query uses multi-dimensional indexes:

```
SUGGESTED_RETAIL_PRICE(Text DESC,
StartPos, EndPos)
COST(Text, StartPos, EndPos)
```

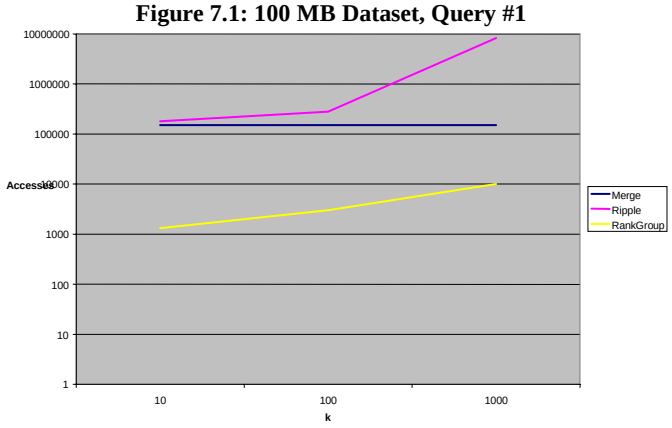
7. EXPERIMENTAL RESULTS

Using the shredded XBench dataset, we then ran each of our benchmark queries in the same testing environment as described in Section 4.3.6. For these tests, both the running time in milliseconds and the number of tuples accessed was measured. Please note that because the operators were not implemented in a pipelined fashion, accesses for the initial Cartesian join in each benchmark query (between ITEM and the second relation) were not counted. Additionally, because there is no distinction between disk accesses and memory accesses, potential hash-table accesses in Ripple and *RankGroup* joins are counted the same as other accesses. This yields potentially exaggerated access counts because these joins look at the same tuple multiple times.

Each of the benchmark queries was run using datasets of sizes 100KB, 1MB, 10MB, and 100MB and values for *k* of 10, 100, and 1000. Complete results of all tests can be seen in Appendix E.

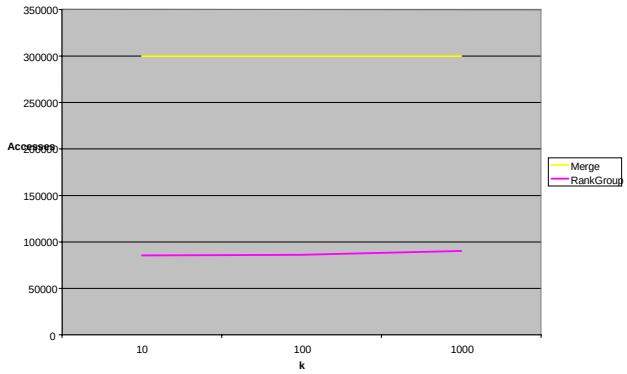
For large datasets and small values of *k*, *RankGroup* truly flexes its muscle. Running Query #1 (items that were available and released the longest ago) with the 100MB dataset (which involved 25,000 tuples per shredded relation) and *k* of 10, *RankGroup* requires just 1313 accesses, as opposed to 149,998 for merge join and 178,394 for Ripple join.

Figure 7.1: 100 MB Dataset, Query #1



For Query #2 (bulkiest items), *RankGroup* again showed its promise, requiring just 85,323 accesses as opposed to 299,996 for merge join.

Figure 7.2: 100 MB Dataset, Query #2



However, for Query #3 (items with greatest profit margin), our implementation of *RankGroup* proved to be inferior to merge join, requiring 443,027 accesses, while merge join required 149,998 accesses. The reason for this has to do with the inherent nature of the query. In Query #3, the idea is to find the items with the greatest profit margin, i.e. the greatest difference between RETAIL_PRICE and COST. We use one index DESC (RETAIL_PRICE) and the other ASC (COST). We then try to complete the join by looking at high->low RETAIL_PRICEs and low->high COSTs. But the reality of the data is that an item has a COST that is very close to its RETAIL_PRICE, and so the very nature of the query forces us to consider RETAIL_PRICE tuples with many COST tuples that will produce no join. This causes unnecessary extra work, and processing the query by first joining and then sorting on the computed column is more efficient.

In general, the results in processing our benchmark dataset were promising but also displayed some limitations:

- Our initial tests were run on a dataset customized to *RankGroup*'s strengths, as *RankGroup* performs best for large datasets and small *k*, while our real-world benchmark dataset did not allow us to fully exploit *RankGroup.*
- Our largest benchmark dataset contained just 25,000 tuples per relation, and this meant that the dataset contained very few items with the same value for the ordering attribute; this allowed Ripple join to perform similarly to *RankGroup* in some situations.
- Real structural XML queries require a more complex join condition (A.StartPos < B.StartPos AND B.EndPos < A.EndPos) than what was originally tested (A.Type = B.Type), and this allow merge join a slight edge in some situations.
- Multiple-relation queries require unimplemented optimizations to allow a limited number of tuples to be returned at each step in the query plan for *RankGroup* to truly outperform.

8. FUTURE WORK

8.1 XML Benchmark Dataset

In the future, we will seek to analyze queries with other XML-specific structural semantics such as recursive queries and wildcard queries. Our benchmark dataset will then need a structure that supports such predicates. We would also alter the dataset to include additional depth and fan-out, plus attributes inside the elements. Our experience indicates that a dataset which supports such queries is hard to find, so in the future either manual synthesis or combination of several benchmark datasets would be necessary.

8.2 Implementation

Future work will involve implementing the *RankGroup* operator as a physical query operator within a RDBMS. This goal would most likely be attainable by working with PostGreSQL and implementing a modification to the query processor to include *RankGroup* for applicable top-*k* queries.

Additionally, optimizations for processing multiple relation joins such as that described in [6] should be considered. So that only the minimal number of tuples is returned at each step in the plan, pipelined techniques would be implemented. It might also be possible to implement non-blocking operators so that each part of the plan could operate simultaneously or nearly simultaneously.

8.3 Experimental Results And Refined Cost Model

Additional metrics should be measured to gather a greater understanding of the performance of each join technique. We would seek to distinguish a cost factor for different types of accesses, such as memory, CPU, and disk. Measurements should take into consideration disk head moving cost, scan cost, how many tuples in a page, whether the tuples are clustered or not, etc. Since we are considering indexes, the disk access of fetching index pages should also be considered. Where the index is clustered or not is also a factor. We should also consider the memory size allocated to the database, the different algorithms, exploiting hashing, and different sorting algorithms. All of these factors play into total cost, and in the future it will be necessary to construct a truly comprehensive cost model.

9. CONCLUSIONS

In this paper we outline a new syntax for top-*k* XML queries and describe our implementation of a process for realizing these top-*k* queries. We found several strengths and limitations of our technique by experimenting with XBench datasets and custom queries. Combined with future efforts outlined in Section 7, our implementation and benchmark tests provide a good starting ground for additional studies in processing top-*k* XML queries.

10. REFERENCES

[1] C. Zhang, J.F. Naughton, D.J. DeWitt, Q. Luo, and G. Lohman. *On supporting containment queries in relational database management systems*. In Proceedings of the ACM SIGMOD Conference, 2001.

[2] Shurug Al-Khalifa, H. V. Jagadish, Jignesh M. Patel, Yuqing Wu. *Structural Joins: A Primitive for Efficient XML Query Pattern Matching*. 18th International Conference on Data Engineering (ICDE' 02).

[3] Ihab F. Ilyas, Walid G. Aref, Ahmed K. Elmagarmid. *Supporting Top-k Join Queries in Relational Databases*. VLDB 2003: 754-765.

[4] Ronald Fagin. Combining Fuzzy Information from Multiple Systems. PODS 1996: 216-226.

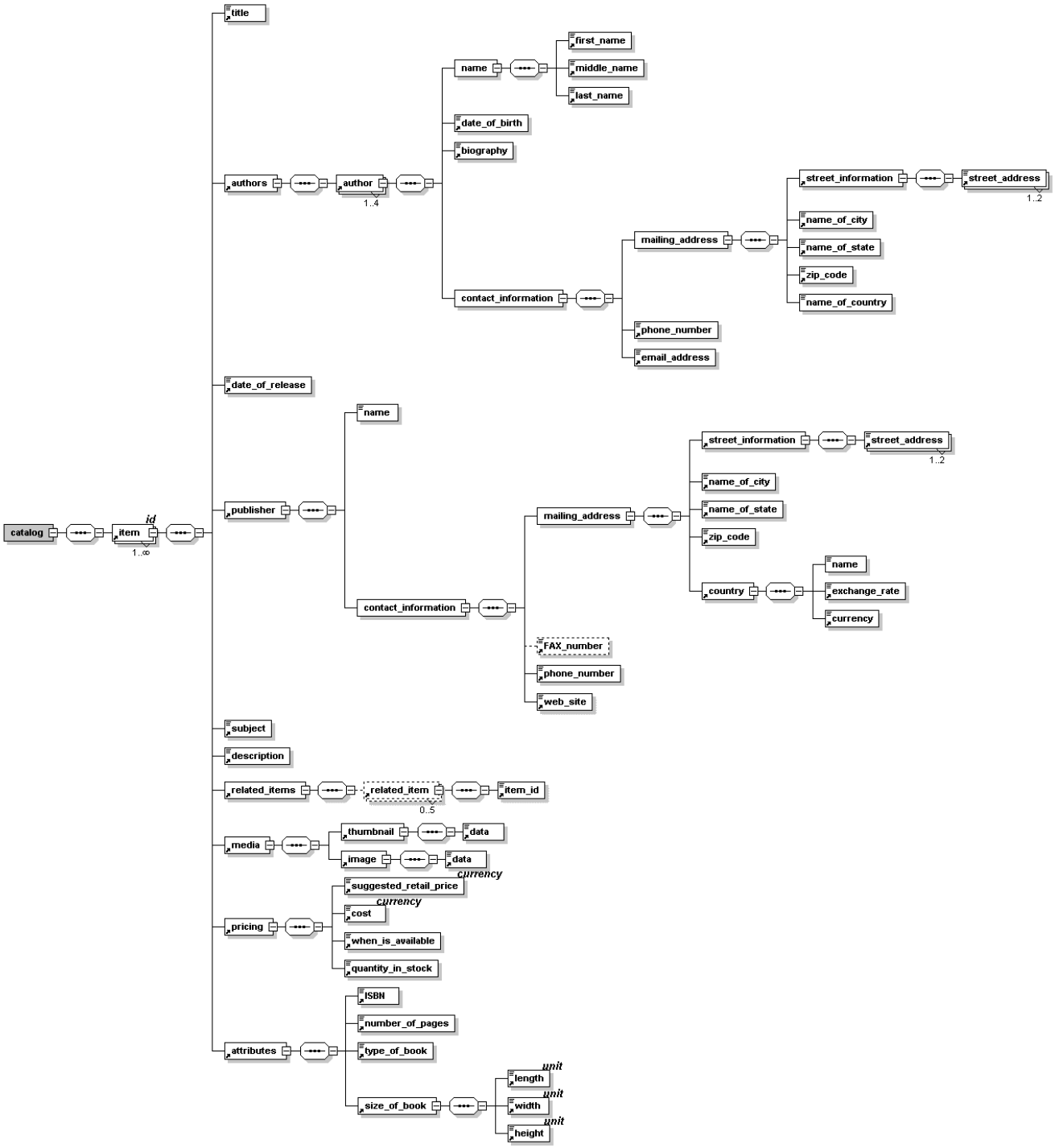
[5] Shanmugasundaram, J., Tufte, K., He, G., Zhang, C., De Witt, D., Naughton, J. *Relational Databases for Querying XML Documents: Limitations and Opportunities*, Proc. of 25th VLDB, pp. 302-314, Edinburg, Scotland, 1999.

[6] M. J. Carey and D. Kossmann. *On saying "enough already!" in SQL*. 1997 SIGMOD Conference: 219-230, 1997.

[7] Surajit Chaudhuri, Luis Gravano. *Evaluating Top-k Selection Queries*. VLDB 1999: 397-410, 1999.

[8] K.C. Chang, S. Hwang. *Minimal Probing: Supporting Expensive Predicates for Top-k Queries*. SIGMOD Conference 2002: 346-357, 2002.

11. APPENDIX A: XBENCH BENCHMARK DOCUMENT STRUCTURE



12. APPENDIX B: XML SHREDDER EXAMPLE

To illustrate the shredding we can show the following sample XML and its shredded equivalent.

XML Document

```
<?XML version="1.0" encoding="UTF-8"?>
<catalog>
  <item title="" subject=""
date_of_release="" ISBN=""
number_of_pages="" cubic_inches=""
rating="">
    <authors>
      <author first_name="" last_name=""
date_of_birth="" rating="">
        <biography/>
         <contact_information
email_address="" phone_number="">
           <mailing_address street=""
city="" state="" zip="" country=""/>
         </contact_information>
      </author>
    </authors>
    <publisher name="">
       <contact_information
email_address="" phone_number="">
             <mailing_address
street="" city="" state="" zip=""
country=""/>
      </contact_information>
    </publisher>
    <pricing suggested_retail_price=""
cost="" when_available="" quantity=""/>
   </item>
</catalog>
```

Shredded version

```
<?XML version="1.0" encoding="UTF-8"?>
<Relations>
<publisher>
       <row name = "" StartPos = "13" level =
"3" EndPos = "18" />
</publisher>
<contact_information>
       <row email_address = "" phone_number =
"" StartPos  = "7" level = "5" EndPos = "10"
/>
       <row email_address = "" phone_number =
"" StartPos  = "14" level = "4" EndPos = "17"
/>
</contact_information>
<item>
       <row title = "" subject = ""
date_of_release = "" ISBN = ""
number_of_pages = "" cubic_inches = "" rating
= "" StartPos  = "2" level = "2" EndPos =
"21" />
</item>
<author>
       <row first_name = "" last_name = ""
date_of_birth = "" rating = "" StartPos  =
"4" level = "4" EndPos = "11" />
</author>
<authors>
       <row StartPos = "3" level = "3" EndPos
= "12" />
</authors>
<biography>
       <row StartPos = "5" level = "5" EndPos
= "6" text = "" />
</biography>
<catalog>
       <row StartPos = "1" level = "1" EndPos
= "22" />
</catalog>
<pricing>
       <row suggested_retail_price = "" cost
= "" when_available = "" quantity = ""
StartPos  = "19" level = "3" EndPos = "20"
text = "" />
</pricing>
<mailing_address>
       row street = "" city = "" state = ""
zip = "" country = "" StartPos  = "8" level =
"6" EndPos = "9" text = "" />
       row street = "" city = "" state = ""
zip = "" country = "" StartPos  = "15" level
= "5" EndPos = "16" text = "" />
</mailing_address>
</Relations>
```

13. APPENDIX C: RANKGROUP JOIN ILLUSTRATIVE EXAMPLE

Here is an illustrative example of the algorithm in action using the same sample dataset (with items added at each step in red font, **->** pointing to the current rank group being considered, and *f(A, B)* referring to the ordering function):

(A1, B1, *f(A, B)* = 8)

RELATION A

	RankGroup	Price	Type
->	1	5	2
		5	4
	2	6	1
	3	8	1
		8	3

RELATION B

	RankGroup	Price	Type
->	1	3	1
		3	2
	2	5	2
	3	6	3
		6	4

Queue: {(A2, B1, *f(A, B)* = 9), (A1, B2, *f(A, B)*= 10)}

(A2, B1, *f(A, B)*= 9)

RELATION A

	RankGroup	Price	Type
	1	5	2
		5	4
->	2	6	1
	3	8	1
		8	3

RELATION B

	RankGroup	Price	Type
->	1	3	1
		3	2
	2	5	2
	3	6	3
		6	4

Queue: {(A1, B2, *f(A, B)*= 10), (A2, B2, *f(A, B)*= 11), (A3, B1, *f(A, B)*= 11)}

(A1, B2, *f(A, B)*= 10)

RELATION A

	RankGroup	Price	Type
->	1	5	2

RELATION B

	RankGroup	Price	Type
	1	3	1

->	1	5	2
	2	6	1
	3	8	1
		8	3

		3	2
->	2	5	2
	3	6	3
		6	4

Queue: {(A2, B2, *f(A, B)*= 11), (A3, B1, *f(A, B)*= 11), (A1, B3, *f(A, B)*= 11)}

(A2, B2, *f(A, B)*= 11)

RELATION A

	RankGroup	Price	Type
	1	5	2
		5	4
->	2	6	1
	3	8	1
		8	3

RELATION B

	RankGroup	Price	Type
	1	3	1
		3	2
->	2	5	2
	3	6	3
		6	4

Queue: {(A3, B1, *f(A, B)*= 11), (A1, B3, *f(A, B)*= 11), (A2, B3, *f(A, B)*= 12), (A3, B2, *f(A, B)*= 13)}

(A3, B1, *f(A, B)*= 11)

RELATION A

	RankGroup	Price	Type
	1	5	2
		5	4
	2	6	1
->	3	8	1
		8	3

RELATION B

	RankGroup	Price	Type
->	1	3	1
		3	2
	2	5	2
	3	6	3
		6	4

Queue: {(A1, B3, *f(A, B)*= 11), (A2, B3, *f(A, B)*= 12), (A3, B2, *f(A, B)*= 13)}

(A1, B3, *f(A, B)*= 11)

RELATION A

	RankGroup	Price	Type
->	1	5	2
		5	4

RELATION B

	RankGroup	Price	Type
	1	3	1
		3	2

	2	6	1
	3	8	1
		8	3

	2	5	2
->	3	6	3
		6	4

Queue: {(A2, B3, *f(A, B)*= 12), (A3, B2, *f(A, B)*= 13)}

(A2, B3, *f(A, B)*= 12)

RELATION A

	RankGroup	Price	Type
	1	5	2
		5	4
->	2	6	1
	3	8	1
		8	3

RELATION B

	RankGroup	Price	Type
	1	3	1
		3	2
	2	5	2
->	3	6	3
		6	4

Queue: {(A3, B2, *f(A, B)*= 13), (A3, B3, *f(A, B)*= 14)}

(A3, B2, *f(A, B)*= 13)

RELATION A

	RankGroup	Price	Type
	1	5	2
		5	4
	2	6	1
->	3	8	1
		8	3

RELATION B

	RankGroup	Price	Type
	1	3	1
		3	2
->	2	5	2
	3	6	3
		6	4

Queue: {(A3, B3, *f(A, B)*= 14)}

(A3, B3, *f(A, B)*= 14)

RELATION A

	RankGroup	Price	Type
	1	5	2
		5	4
	2	6	1
->	3	8	1
		8	3

RELATION B

	RankGroup	Price	Type
	1	3	1
		3	2
	2	5	2
->	3	6	3
		6	4

14. APPENDIX D: RANKGROUP ISOLATED EXPERIMENTAL RESULTS

This table displays the results from running the following simple query over different sized datasets and varying *k*:

```
SELECT TOP-k
FROM A, B
WHERE A.Type = B.Type
ORDER BY A.Price + B.Price
```

All results are in milliseconds.

Relation A Tuples	Relation B Tuples	Rank Groups	Types per Rank Group	
1771	1242	1 to 100	50 to 100	
k	**10**	**100**	**1000**	**10000**
NestedLoop	1150	1460	1460	1780
Merge	78	78	93	140
Ripple	1	15	62	687
PriceGroup	1	1	15	171

Relation A Tuples	Relation B Tuples	Rank Groups	Types per Rank Group	
2226	3431	1 to 100	100 to 500	
k	**10**	**100**	**1000**	**10000**
NestedLoop	4343	4484	4484	4390
Merge	31	31	46	93
Ripple	31	46	250	2703
PriceGroup	1	15	31	171

Relation A Tuples	Relation B Tuples	Rank Groups	Types per Rank Group	
5510	6471	100 to 500	50 to 100	
k	**10**	**100**	**1000**	**10000**
NestedLoop	40171	40171	40171	39125
Merge	2700	2218	2310	2215
Ripple	15	15	187	1156
PriceGroup	1	15	15	250

Relation A Tuples	Relation B Tuples	Rank Groups	Types per Rank Group	
12351	15798	500 to 1000	50 to 100	
k	**10**	**100**	**1000**	**10000**
NestedLoop	219718	219718	219718	227703
Merge	12930	11578	11500	11671
Ripple	1	15	187	1218
PriceGroup	1	1	15	250

Relation A Tuples	Relation B Tuples	Rank Groups	Types per Rank Group	
45413	45061	500 to 1000	100 to 500	
k	**10**	**100**	**1000**	**10000**
Merge	30703	30703	30703	30468

Ripple	31	93	562	5250
PriceGroup	1	1	15	171

Relation A Tuples	**Relation B Tuples**	**Rank Groups**	**Types per Rank Group**
77931	88054	1000 to 5000	50 to 100

	k	**10**	**100**	**1000**	**10000**
Ripple		15	15	140	1187
PriceGroup		1	15	15	187

Relation A Tuples	**Relation B Tuples**	**Rank Groups**	**Types per Rank Group**
288039	349109	1000 to 5000	100 to 500

	k	**10**	**100**	**1000**	**10000**
Ripple		46	62	656	6156
PriceGroup		15	15	46	218

Relation A Tuples	**Relation B Tuples**	**Rank Groups**	**Types per Rank Group**
421397	500000	1000 to 5000	500 to 1000

	k	**10**	**100**	**1000**	**10000**
Ripple		187	203	1390	13843
PriceGroup		1	1	31	187

15. APPENDIX E: BENCHMARK QUERY EXPERIMENTAL RESULTS

Please note:

- Because operators were not implemented in a pipelined fashion, accesses for the initial Cartesian join in each benchmark query (between ITEM and the second relation) were not counted.
- Because there is no distinction between disk accesses and memory accesses, potential hash-table accesses in Ripple and *RankGroup* joins are counted the same as other accesses. This yields potentially exaggerated access counts because these joins look at the same tuple multiple time.

		Query #1: Oldest items released and available							**Query #2: Bulky Items**							**Query #3: Items with greatest profit margin**					
		k = 10		**100**		**1000**			**k = 10**		**100**		**1000**			**k = 10**		**100**		**1000**	
		msecs	**accesses**	**msecs**	**accesses**	**msecs**	**accesses**		**msecs**	**accesses**	**msecs**	**accesses**	**msecs**	**accesses**		**msecs**	**accesses**	**msecs**	**accesses**	**msecs**	**accesses**
100KB (25 tuples per relation)																					

NestedLoop	10MB (2,500 tuples per relation)	RankGroup	Ripple	Merge	NestedLoop	1MB (250 tuples per relation)	RankGroup	Ripple	Merge	NestedLoop
3359		15	15	15	46		15	15	15	15
6255000		398	2605	1498	63000		234	1072	148	675
3359		31	46	15	46		15	15	15	15
6255000		2066	33914	1498	63000		536	2553	148	675
3359		78	93	15	46		n/a	n/a	n/a	n/a
6255000		6005	109189	1498	63000		n/a	n/a	n/a	n/a
6765		109	140	31	93		31	15	15	15
12510000		9605	130500	2996	126000		702	3197	296	1350
6765		156	187	31	93		31	15	15	15
12510000		13757	173039	2996	126000		1082	5011	296	1350
6765		250	265	31	93		n/a	n/a	n/a	n/a
12510000		21507	257473	2996	126000		n/a	n/a	n/a	n/a
3359		156	156	15	46		15	15	15	15
6255000		16063	79618	1498	63000		370	1576	148	675
3359		406	359	15	46		15	15	15	15
6255000		31033	153143	1498	63000		473	1993	148	675
3359		453	437	15	46		n/a	n/a	n/a	n/a
6255000		37945	187143	1498	63000		n/a	n/a	n/a	n/a

RankGroup	Ripple	Merge	NestedLoop	RankGroup	Ripple	Merge	NestedLoop	RankGroup	Ripple	Merge
15	62	31	n/a	31	156	203	n/a	984	4546	78
602	38923	14998	n/a	1313	178394	149998	n/a	18254	6818778	29996
46	78	31	n/a	578	1734	203	n/a	859	4593	78
1350	82979	14998	n/a	2996	277530	149998	n/a	19787	7028848	29996
250	1468	31	n/a	1828	5460	203	n/a	1250	6312	78
5906	2290541	14998	n/a	10179	8310741	149998	n/a	29470	9566112	29996
40750	40281	31	n/a	15703	n/a	187	n/a	1259688	7094386	14998
922798	5199878	14998	n/a	443027	n/a	149998	n/a	n/a	n/a	187
66796	65640	31	n/a	n/a	n/a	187	n/a	n/a	n/a	149998
1259688	7094386	14998	n/a	n/a	n/a	149998	n/a	n/a	n/a	149998
n/a	n/a	31	n/a	n/a	n/a	187	n/a	n/a	n/a	149998
n/a	n/a	14998	n/a	n/a	n/a	149998	n/a	n/a	n/a	149998

100MB (25,000 tuples per relation)