Performing Grouping and Aggregate Functions in XML Queries

Huayu Wu, Tok Wang Ling, Liang Xu, and Zhifeng Bao
School of Computing
National University of Singapore
wuhuayu@comp.nus.edu.sg, lingtw@comp.nus.edu.sg, xuliang@comp.nus.edu.sg, baozhife@comp.nus.edu.sg

ABSTRACT

Since more and more business data are represented in XML format, there is a compelling need of supporting analytical operations in XML queries. Particularly, the latest version of XQuery proposed by W3C, XQuery 1.1, introduces a new construct to explicitly express grouping operation in FLWOR expression. Existing works in XML query processing mainly focus on physically matching query structure over XML document. Given the explicit grouping operation in a query, how to efficiently compute grouping and aggregate functions over XML document is not well studied yet. In this paper, we extend our previous XML query processing algorithm, *VERT*, to efficiently perform grouping and aggregate function in queries. The main technique of our approach is introducing relational tables to index values. Query pattern matching and aggregation computing are both conducted with table indices. We also propose two semantic optimizations to further improve the query performance. Finally we present experimental results to validate the efficiency of our approach, over other existing approaches.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems

General Terms

Algorithms

Keywords

grouping, aggregate function, query processing, XML

1. INTRODUCTION

XML is an important standard format for data storage and exchange over the Internet. As a result, how to efficiently process queries over XML databases attracts lots of research interests [15]. Existing works on XML query processing mainly focus on how to efficiently match the query pattern to XML document, which is considered as a core operation to process queries in most standard XML query languages (e.g. XPath [2] and XQuery [5]). As more and more business data are represented in XML format, analytical queries involving grouping and aggregate operations have become more popular. To process an analytical query with grouping, existing pattern matching techniques are no longer effective. A new technique is required to handle the grouping operation in queries.

Similar to relational databases, most analytical queries over XML documents contain a main operator *group-by* and a set of aggregate functions such as max( ), min( ), sum( ), count( ), avg( ), etc. In most XML query languages, aggregate functions are syntactically supported; however, the shortcoming is the lack of explicit support for grouping. E.g. XQuery 1.0 is a widely adopted version by most XQuery engines, however, grouping in XQuery 1.0 can only be expressed implicitly using nesting. This nested expression for grouping can be neither well understood by users, nor easily detected by query optimizer, as pointed out by [4].

There are many efforts [6, 3, 18] on extending the expressive power for XQuery to support grouping, until W3C publishes the latest version of XQuery, XQuery 1.1 [11], to introduce a new construct to explicitly express grouping in FLWOR expression. For example, consider the bookstore document shown in Fig. 1, and a query to find the average book price for each publisher in each year. This query can be expressed in XQuery 1.1 as follows:

```
FOR $p IN distinct-values(doc("bookstore.xml")//book/publisher),
      $y IN distinct-values(doc("bookstore.xml")//book/year),
LET $pr :=
      doc("bookstore.xml")//book[publisher=$p and year=$y]/price
GROUP BY $p, $y
ORDER BY $p, $y
RETURN
   <book publisher="{$p}" year="{$y}">
      <average_price>{avg($pr)}</average_price>
   </book>
```

Although the work of XQuery 1.1 has just started, it reflects the importance of grouping operations in XML queries. As a result, how to efficiently process XML queries with grouping becomes a new research direction. Since RDBMS is the dominant model for structured data, in the early stage there are many works [29, 28, 25] on storing and querying XML data using RDBMS. In these relational approaches, they normally shred XML documents into tables and convert XML queries into *SQL* to query the database. This sort of approaches can handle grouping in XML queries with the group-by function in *SQL*. However, *SQL* has difficulty supporting multi-level (nested) grouping, which often appears in analytical XML queries. Also the primeval drawbacks of relational approaches in query structural search, such as the inefficiency to answer "//"-axis queries over XML document with recursively appearing elements [7], are a big concern. A recent work [14] proposed an algorithm to compute *group-by* queries natively over XML document. They scan the document for each query and prune out irrelevant nodes. For the relevant nodes, they merge and count the analytical attributes for each group so that aggregate function can be easily performed.The major problem is that their approach is only suitable for queries with simple predicate. They find the relevant nodes in documents by scanning the document for each query. However, if a query contains complex predicates as selection conditions and the document schema is complex (e.g. "//"-axis query and documents with recursively appearing tags), file scan is neither efficient, nor effective to return correct answers. That also explains why many twig pattern matching techniques, e.g. *TwigStack* [7], attract lots of research attention.

To solve the problem in structural search in existing work for XML query processing with grouping, we extend our previous algorithm *VERT* [26], to efficiently compute *group-by* operators in XML queries with complex predicates. Given a group-by query, we match the query pattern to the document based on query predicates using *VERT*. *VERT* can handle both structural search and content search in an XML query efficiently, thus it is suitable for queries with complex predicates. After that, we use the table indices to get the values of relevant properties and compute the aggregate functions in different levels of nested grouping by scanning the resulting tuples.

The contribution of this paper is summarized as follows:

- We propose an extended algorithm $VERT^G$ based on our previous algorithm, *VERT*, to process *group-by* queries over XML document efficiently. $VERT^G$ inherits all the advantages of *VERT*, including the efficiency in matching complex query pattern.
- We propose two optimizations based on semantic information like object and property, which can further enhance the query performance.
- We conduct experiments to compare the efficiency of our approach to existing works, to validate the benefit of our approach.

Some background, as well as related work is presented in Section 2. In Section 3 and 4 we describe a format of queries with grouping and aggregation, which is used in our system, and design the algorithm $VERT^G$ to efficiently process queries. In Section 5, we optimize our algorithm with semantics of object and property. We present experimental results in Section 6 and conclude this paper in Section 7.

2. BACKGROUND AND RELATED WORK

2.1 XML data model and queries

XML documents are normally modeled as ordered trees without considering ID/IDREF, in which nodes represent elements, attributes or values in document, and edges represent the relationships between element, attribute and value.

The predicate in a query are the constraint to filter the query results. It is similar to the *where* clause in *SQL* to specify the selection condition. An XML query with predicate can be represented by a tree structure, which is called a *twig pattern*. Finding all occurrences of a twig pattern query in the tree structure of the XML document is the core operation for XML query processing.

2.2 Grouping and aggregate function

Aggregate function is used to perform analytical computation and summarization. Some common aggregate operators include max( ), min( ), sum( ), count( ), avg( ), etc. Since usually we need to apply the aggregate functions to each of a number of groups, another operator *group-by* is commonly used. Furthermore, an optional operator *having* normally comes with *group-by* to specify the qualifications over groups.

2.3 Related work

Grouping and aggregation are well supported by SQL in relational databases. There are also research works [16, 19] to generalize or optimize such analytical operations in RDBMS. Since XQuery 1.0 lacks functions to explicitly support grouping, processing queries with grouping in XML is addressed by researchers in recent years. Intuitively, the relational approaches [29, 28, 25] to store and query XML data can support grouping and aggregation because those approaches shred XML data into relational tables and convert queries involving grouping operation into SQL to query the database. These sort of approaches have limitations such as the inefficiency to answer "//"-axis queries over the document with recursively appearing tags [7]. Also [29] proves that the relational approaches are not efficient as native approaches for most cases.

The research in XML grouping in native XML databases mainly focuses on three directions. The first direction is on how to support grouping by either providing logical grouping operators [13, 9, 22], or detecting grouping in nested queries and rewriting queries [10, 12, 20, 23, 24]. Particularly, in [13] they provide algebraic operators for grouping, and achieve efficient construction of XML elements using their algebra. [9, 22] focus on designing a graphical query language supporting grouping, and eventually the query will be translated to XQuery expression to process. The works [10, 12, 20, 23, 24] detect the potential grouping from nested queries and using different rewriting rules to transform the queries into a new structure with explicit *group-by* operator. However, this approach has a bottleneck, which is the difficulty of detecting grouping in nested queries. Sometimes it is even not possible to detect such potential grouping [3].

Due to the limitation of detecting grouping in nested queries, some researchers focus on a second direction, which is extending XQuery 1.0 to explicitly support grouping in queries. In [6, 3, 18] they defined extra operator to complement FLWOR expression in XQuery for grouping. In this case, the query optimizer does not need to detect potential grouping in an XQuery expression. Based on these research efforts, W3C published the new version of XQuery, XQuery 1.1, in which a grouping construct is introduced as a core requirement, though the work has just started.

Since none of the works mentioned above focuses on physically computing *group-by* and aggregate function over XML documents, a new research direction works on algorithmic support for processing grouping and aggregation. [14] proposes an algorithm to directly compute *group-bys*. However their method did not consider the case that an XML query may contain complex predicate and the document may also have a complex schema such as containing recursively ap-

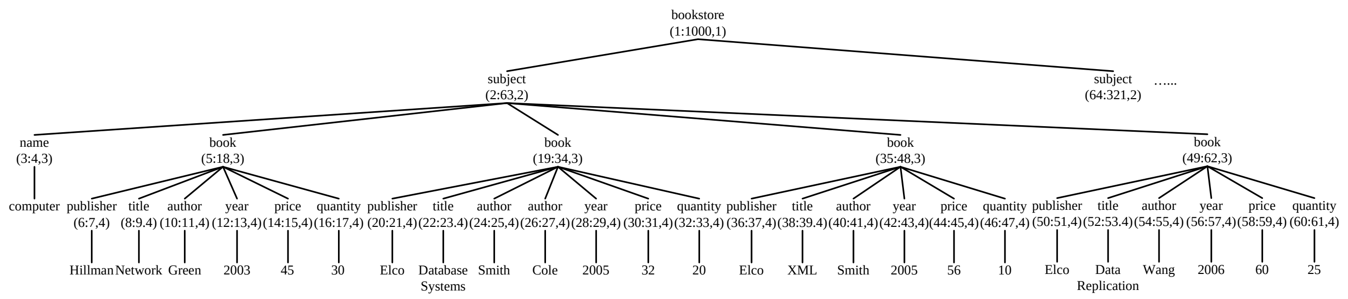


Figure 1: An example document *bookstore.xml*

pearing elements. For such documents and queries, the file scan to select relevant nodes in [14] may fail to work, and this motivates many pattern matching techniques ([15]). There are also works ([21]) to eliminate duplicates during grouping computation so that better performance can be retrieved.

3. QUERY EXPRESSION

In this section, we describe the general form of XML queries with grouping, which is used in our $VERT^G$ algorithm. The general query form is shown in Fig. 2 below.

```
Expr        ::= "PATTERN:" XPath_expression
                Group-by*
Group-by    ::= "GROUP BY:" group-by_attribute+
                ("ORDER BY:" group-by_attribute+)?
                ("HAVING:" condition+)?
                "RETURN: {" aggregate_function+
                            Group-by* "}"
Occurrence Indicator: + 1 or more  * 0 or more  ? 0 or 1
```

Figure 2: Query form used by $VERT^G$

Pattern: The grouping operation and aggregation function are built on twig pattern queries as mentioned in Section 1. We use XPath expressions to represent twig patterns. The nodes in a twig pattern should include all the predicate nodes, group-by nodes and output nodes in the given query.

Grouping: Grouping is explicitly expressed using the keyword *group by*. *Group by* indicates the query nodes by which the results are grouped, and an optional *order by* clause indicates the order to output each group. Without indicating the grouping order, we will output the result based on the ascending order of the group-by nodes by default. Grouping often comes with optional *having*, which is used to specify the aggregate conditions. Grouping can be parallel, which means the results are grouped in multiple ways by different properties. Grouping can also be nested, which means the results within each *return* clause can be further grouped.

Return: The *return* clause specifies the aggregate functions in each group. As mentioned above, grouping can recursively appear in a query, so the output information following the *return* clause can be the value of an aggregate function, or a nested grouping operation with another *return* clause.

EXAMPLE 1. *Consider a query to find first all the computer books grouped by publisher to output the total number of books of each publisher whose average book price is greater than 40, and then group all books under each of these publishers by year and price separately to find the total quantity of books in each subgroup. This query can be expressed as Q1 in Fig. 3. Note that the pattern in Q1 is an XPath expression in which all relevant nodes to the query are included.*

```
Q1: PATTERN: subject[name="computer"]/book[publisher][year][price][quantity]
      GROUP BY: publisher
      ORDER BY: publisher
      HAVING: avg(price)>40
      RETURN: { count(book),
          GROUP BY: year
          RETURN: { sum(quantity) }
          GROUP BY: price
          RETURN: { sum(quantity) } }
```

Figure 3: Example query Q1

4. ALGORITHM

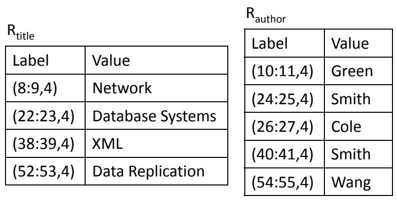
In this section, we introduce the algorithm $VERT^G$ to perform grouping as well as aggregate function in XML queries with complex predicate. Our algorithm contains two phases. In the first phase, we perform pattern matching to find all the relevant nodes that satisfy the query predicates in XML document. In our implementation we use our previously proposed algorithm *VERT* to match query pattern because: (1) *VERT* solves content problems existing in many other algorithms, such as the inefficiency of content management, content search and content extraction. (2) *VERT* makes use of relational tables to index values, which is more compatible with the algorithm proposed in this paper, and (3) VERT is a very efficient algorithm to process twig pattern queries. After that, in the second phase we use the table indices on values, together with the result from pattern matching, to perform grouping and compute aggregate functions. Multi-level grouping can be efficiently supported in $VERT^G$. First of all, we briefly review *VERT* with table indices.

4.1 VERT and table index

Traditional twig pattern matching techniques suffer from problems dealing with contents, such as difficulty in data content management and inefficiency in performing content search. We proposed *VERT*, to solve these content problems, by introducing relational tables to index values. Most XML query processing algorithms assign labels to each document node so that the parent-child relationship or ancestor-descendant relationship between two nodes can be easily identified using node labels. In *VERT*, we use tables to store labels of properties and their values. The term *property* used in this paper refers to the property of each object, irrespective of whether it appears as an element type or an attribute type in an XML document. When we parse an XML document, we label only elements and attributes, and put the labels into corresponding streams. A stream for each element or attribute is used to store labels for that element or attribute in document order. Values in documents are not labeled; instead we put them into relational tables together with the corresponding property node labels. The schema of each table is:

$$R_{property}(\text{Label, Value})$$

In this schema, the subscript *property* in the table name indicates for which type of property this table is used. The two fields "Label" and "Value" store the label of the property node and its child value. There are different labeling schemes for static or dynamic[1] XML documents. Suppose we adopt containment labeling scheme [29] in our implementation, the labels assigned to each document node is shown in Fig. 1 and the example tables for property "title" and "author" in the the same document is shown in Fig. 4.



R_{title}

Label	Value
(8:9,4)	Network
(22:23,4)	Database Systems
(38:39,4)	XML
(52:53,4)	Data Replication

R_{author}

Label	Value
(10:11,4)	Green
(24:25,4)	Smith
(26:27,4)	Cole
(40:41,4)	Smith
(54:55,4)	Wang

Figure 4: Table indices for "title" and "author"

To process a twig pattern query with value comparison in the predicate, *VERT* performs content search first on the value comparisons, and then rewrites the query by removing those value comparisons and performs structural search on the new query pattern using any efficient structural join algorithm, e.g. *TwigStack*. For example, to process the query shown in Fig. 5(a), *VERT* first refers to the *title* table to get the labels for property *title* whose value is "XML". It constructs a new *title* stream for this query using the selected *title* labels. Now we just need to process the rewritten query in Fig. 5(b), ignoring the value comparison and using the new stream for *title*. The reason why we can simplify the original query like this is that the value of the property *title* for all the labels in the new title stream is "XML" and actually we have already handled the predicate based on *title*. Comparing *VERT* using a relational table to handle values with the pure structural matching algorithms, we can see *VERT* significantly reduces the size of the stream for *title*, and reduces the number of structural joins by 1. We also propose optimizations for *VERT*. Details can be found in [26].

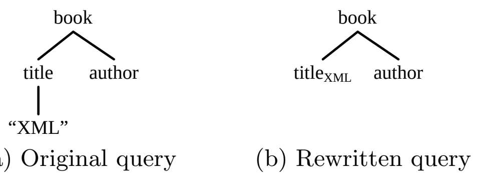


Figure 5: Example query in *VERT* processing

4.2 Data structures and output format

We define the query format in Section 3. In this section, we discuss how we store the query information into relevant data structures, which will be used during query processing with $VERT^G$. Since we adopt $VERT$ to process pattern matching for queries, we need to maintain the index tables for each type of property, as mentioned in last section. The tables are also used to extract actual values for each property when we perform grouping and aggregation.

[1]Static document is the document which is seldom updated, whereas, dynamic document is frequently updated.

Besides the table indices, we also need two tree structures in $VERT^G$. One is a query structure tree, named QT, and the other one is a grouping structure tree, named GT. QT is used to represent the XPath expression in a query, and it is also named as *twig pattern*. $VERT^G$ matches QT to the document. This pattern matching process can be considered as a selection based on predicates. In GT, each node stands for a grouping operation. Thus within a GT node we record the group-by property[2], the order-by property, the grouping constraint and the output aggregate function. Each GT node has two pointers: *child* and *next sibling*. The *child* points to a nested grouping operation, and the *next sibling* points to a parallel grouping operation in the same grouping level as the current node.

Example 2. *Consider Q1 in Fig. 3. The structures* QT *and* GT *for Q1 are shown in Fig. 6. In* GT*, the four entries in each node stand for group-by property, order-by property, grouping constraint and output aggregate function in order. The* child *pointer reflects the nested relationship between the two levels of grouping, and the* next sibling *pointer reflects the parallel relationship between the two grouping operations in the same level.*

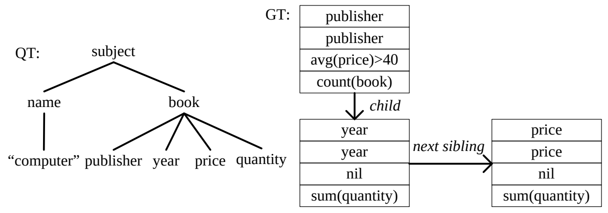


Figure 6: Structures for Q1

The format of the output results can be easily generated by analyzing the query. The result format for Q1 is shown in Fig. 7. Due to the space limitation, the details of generating QT, GT and output format from the query are omitted.

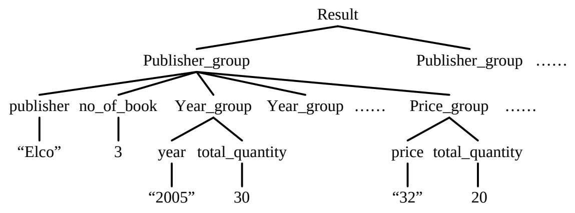


Figure 7: Output format for Q1

4.3 Query processing

To process a query with grouping using $VERT^G$, we first perform a pattern matching for the query to the XML document. After that in the second phase we perform grouping and aggregation based on the matching results.

Pattern matching: As mentioned previously, we adopt *VERT* for pattern matching. The output of this pattern matching phase is tuples of labels for relevant nodes, which is considered as intermediate result set, named as $RS_{intermediate}$. The relevant nodes means the nodes which are searched by

[2]To simplify the explanation, we assume there is one group-by property in each grouping operation. The data structures can be easily extended to support multiple group-by properties. The same assumption is made for grouping constraint and output aggregate function.

the query, used as group-by properties, or involved in aggregate functions. For example, to process Q1, we match the path expression following PATTERN to the document. Since nodes "book", "publisher", "year", "price" and "quantity" appear in GROUP BY, HAVING and RETURN clauses, $VERT$ will output the labels for these nodes in each matched segment. The intermediate result set for Q1 is shown in Fig. 8, where each tuple contains the node labels in each twig pattern occurrence in document.

$RS_{intermediate}$

book	publisher	year	price	quantity
(5:18,3)	(6:7,4)	(12:13,4)	(14:15,4)	(16:17,4)
(19:34,3)	(20:21,4)	(28:29,4)	(30:31,4)	(32:33,4)
(35:48,3)	(36:37,4)	(42:43,4)	(44:45,4)	(46:47,4)
(49:62,3)	(50:51,4)	(56:57,4)	(58:59,4)	(60:61,4)

Figure 8: Pattern matching result for Q1

Performing grouping: In the second phase, we perform grouping, as well as aggregate functions. We first construct RS_{final} by extracting actual values for the properties in the intermediate result set $RS_{intermediate}$ using table indices for each property. After that we traverse the GT for the query according to a child-first fashion. The recursive method for GT traverse is shown in Algorithm 1. We start with **traverse** ($GT.root$) and the global variable *level*, which indicates the grouping level that we start performing grouping with, is initialized to be 1. When we visit a node, we attach the group-by property, order-by property, grouping constraint and aggregate function in that node to the end of the corresponding global lists GL, OL, CL and AL. If a node does not have a child, we begin to perform grouping in RS_{final} with current GL, OL, CL, AL and *level*. We also consider the parallel grouping within the same level by checking the next sibling of each GT node. The *level* value is set to be the level of the node which has a next sibling.

Algorithm 1: **traverse** (*node*)

```
1 attach the group-by property, order-by property, grouping
  constraint and aggregate function in node to the end of the lists
  $GL$, $OL$, $CL$ and $AL$ separately
2 if node.getChild == null then
3     perform ($RS_{final}$, $GL$, $OL$, $CL$, $AL$, level)
4     delete the last entry of $GL$, $OL$, $CL$ and $AL$.
5 else
6     traverse (node.getChild)
7 if node.getNextSibling != null then
8     level=node.getLevel
9     traverse (node.getNextSibling)
```

To process the query Q1, we traverse the GT in Fig. 6. By Algorithm 1, we perform grouping twice for Q1: one is for properties "publisher" and "year" with *level*=1, and the other one is for "publisher" and "price" with *level*=2. Now we move to the algorithm to perform grouping, which is shown in Algorithm 2. Note that although the RS_{final} is in relational table format, we cannot use SQL to compute all the group-by clauses, because SQL cannot support nested grouping due to the flat format of relational table. We partition RS_{final} in line 1. The function $partition(RS_{final}, GL, OL)$ sorts the table RS_{final} based on all the properties in GL, following the order by which the properties appear in OL if it is different from that in GL. Sorting by multiple properties works in the way that the system sorts tuples by the first property, and if two or more tuples have the same value on the first property, then it sorts them by the second property, and so forth. Now the tuples can be partitioned into different groups for different levels.

Algorithm 2: **perform** (RS_{final}, GL, OL, CL, AL, $level$)

```
1  partition($RS_{final}$, $GL$, $OL$)
2  let $n$ = $GL$.length
3  foreach $i$ = level to $n$ do
4      initialize $cv$[i] = $RS_{final}$[$GL$[i]]
5      initialize lists count[i][ ], sum[i][ ], max[i][ ], min[i][ ] for
       relevant properties in $RS_{final}$, which are used to compute
       aggregate functions
6  foreach tuple $t$ in $RS_{final}$ do
7      foreach $i$ = level to $n$ do
8          if $t[GL[i]]$ != $cv[i]$ then
9              foreach $j$ = $i$ to $n$ do
10                 check the constraints in $CL$[j]
11                 if $CL[j]$ holds then
12                     compute aggregate functions in $AL$[j]
13                     put $cv$[j] and the aggregate results into the
                       appropriate position in result tree
14                 $cv$[j] = $t$[$GL$[i]]
15                 reset count[j][ ], sum[j][ ], max[j][ ], min[j][ ]
16             break
17         else
18             update count[j][ ], sum[j][ ], max[j][ ], min[j][ ]
19 foreach $i$ = level to $n$ do
20     check the constraints in $CL$[i]
21     if $CL[i]$ holds then
22         compute aggregate functions in $AL$[i]
23         put $cv$[i] and the aggregate results into the appropriate
           position in result tree
```

EXAMPLE 3. *Consider Q1 in Fig. 3 with the intermediate result set shown in Fig. 8. Using the index tables $R_{publisher}$, R_{year}, R_{price} and $R_{quantity}$ we can get the exact values for each field. When the* **perform** *function is first called in Algorithm 1, we partition the RS_{final} based on properties "publisher" and "year". The result is shown in Fig. 9. The bold lines in the RS_{final} show the partition.*

Level 1 partition ↓ Level 2 partition ↓

RS_{final}

publisher	year	book	price	quantity
Elco	2005	(19:34,3)	32	20
Elco	2005	(35:48,3)	56	10
Elco	2006	(49:62,3)	60	25
Hillman	2003	(5:18,3)	45	30

Figure 9: Example RS_{final} with partition for Q1

In lines 2-5, we initialize the lists used in this algorithm. Particularly, cv[i] stores the current value of the group-by property in the ith level group, while statistic lists count[i][ ], sum[i][ ], max[i][ ] and min[i][ ] store the corresponding current statistic values for the ith level group. In lines 6-23, we update these lists to get aggregate results. We check each tuple in RS_{final} to see whether any new partition in the different levels begins at this tuple. This is done by checking whether the value of the group-by property in each level is changed in line 8. If any new partition begins in a certain grouping level, for every lower level a new partition also begins. Then we check the HAVING constraint in these levels and compute the aggregate functions using the corresponding statistic lists, as shown in lines 10-13. After that we reset the current group-by property value and the statistic lists for each of these levels, in lines 14-15. If in a tuple, some grouping level does not end, we simply update the statistic lists in line 18. In many cases, we do not need to maintain all the statistic lists as the query may be only interested in some of them. To simplify the presentation, we use all the statistic lists in the pseudo-code. Lines 19-23 finalize the query processing by outputting the result for the last group in each grouping level.

EXAMPLE 4. *When the* **perform** *function is first called during* GT *traverse for Q1, the* RS_{final} *with partition is shown in Fig. 9. We start with* level=*1, and initialize the current value and the necessary statistic lists for each grouping level, as shown in Fig. 10. The list cv[ ] contains two entries since there are two levels of grouping. The statistical list, saying count[1][ ], stores the total number of each target property in the first level, e.g. count[1][2] is the count of the second property "price" in level 1 grouping.* Nil *in some entries of each list means the corresponding statistic value is not asked by the query and we do not need to maintain it.*

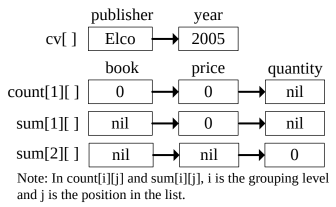
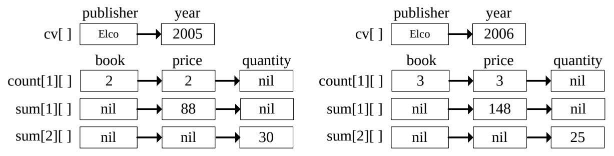

publisher
year
cv[ ]
Elco
2005
book
price
quantity
count[1][ ]
0
0
nil
sum[1][ ]
nil
0
nil
sum[2][ ]
nil
nil
0
Note: In count[i][j] and sum[i][j], i is the grouping level and j is the position in the list.

Figure 10: Example initial lists for Q1

When the system reads the third tuple in RS_{final}, *the value in cv[1] is the same as the "publisher" value in the third tuple. That means the current level 1 group does not end at this tuple. Thus it updates the lists count[1][ ] and sum[1][ ]. However, the value "2005" in cv[2] is different from the "year" value "2006" in the third tuple, which means current level 2 group ends. It then follows lines 10-13 in Algorithm 2 to compute the aggregate function in level 2 grouping based on current statistic list for this level, e.g. sum[2][ ], and puts the value "2005" for the group-by property "year" and the result "30" for aggregate function* sum(quantity) *into appropriate position in the result tree as shown in Fig. 7. After that the system resets cv[2] and the statistic list sum[2][ ] for level 2 grouping and continues reading the next tuple. The relevant lists before and after reading the third tuple is shown in Fig. 11.*



(a) Before the third tuple　(b) After the third tuple

Figure 11: Example lists before and after reading the third tuple in RS_{final} for Q1 processing

4.4 Early pruning

Anti-monotonic constraint is defined as the constraint which will never be true once it becomes false. Some aggregate constraints that appear in HAVING clauses, such as count( ) $\leq$ *num*, max( ) $\leq$ *num*, min( ) $\geq$ *num* or sum( ) $\leq$ *num* (*num* is a numeric value), are anti-monotonic constraints. E.g. for the constrain max(price) $\leq$ 100, once we get a price greater than 100 in a group, we can never turn the constraint to be true, no matter how many more prices are checked in the same group. Motivated by anti-monotonic constraints, some early pruning can be done to enhance the query performance. When we read tuples in RS_{final}, we can check the anti-monotonic constraint first, rather than checking all constraints after meeting the end tuple of the group. If any anti-monotonic constraint is violated by a certain tuple, all other tuples in the same group can be skipped.

4.5 Extension flexibility

The query form and query processing algorithms presented in Section 3 and Section 4.2 are built on basic aggregation. Sometimes the user may issue queries involving keyword constraints *distinct*, or some other aggregate functions, or even moving windows following the group-by properties. In this section, we explain briefly how our algorithm is flexible to be extended to support these advanced features.

Distinct: Some aggregate function aims to find aggregate results on distinct values in the group. In this case, we need to introduce keyword *distinct*. There are two types of parameters that can be used by *distinct* constraint. The first type is property. E.g. count(*distinct* name) counts the number of different names distinguished by name values. To support this type of *distinct*, we can maintain a sorted list to store different values for the corresponding properties. When a value comes, we can know whether it is a *distinct* value or not by check the sorted list.

The second type of parameter following *distinct* constraint is object, e.g. count(*distinct* book). This function is not easy to compute as "book" is an object class rather than property, and there is no child value for "book" to explicitly distinguish each "book" object. One way to distinguish objects under the same class is to discover more semantics on object ID [8]. As long as the ID of an object class is clear, we can easily perform aggregate functions on distinct objects by introducing ID to RS_{final} for the relevant object.

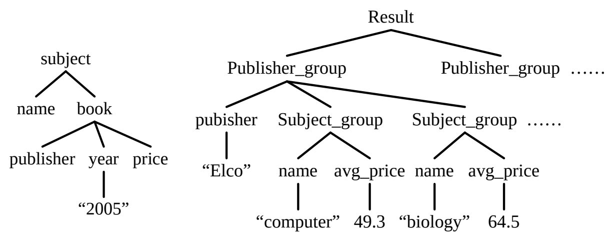
Other aggregate functions: We discuss four more aggregate functions that are frequently asked, namely, *maxN( )*, *minN( )*, *median( )* and *mode( )*. The function *maxN( )* and *minN( )* are top N functions to find the N maximum or minimum values. *Median( )* returns the value that separates the higher half of a set of values from the lower half, and *mode( )* is used to find the value that occurs most often in a set. In the discussion about *distinct* keyword above, we mentioned that we can maintain an additional sorted list to store different values for particular property. To compute *maxN( )*, *minN( )*, *median( )* and *mode( )*, we not only need the sorted list for the distinct values for relevant properties, but also need a frequency list in which each entry stores the number of occurrences of the value in the corresponding entry in the sorted list. Using these two lists, these aggregate functions can be easily computed.

Moving windows: Moving windows are used to group answers by ranges of values on a certain property. E.g. a query needs to find the total quantity of books group by range of 5 years with a moving step of 3 years, beginning at 2008. In this query, we need to put books with year in [2008, 2012] together, with year in [2011, 2015] together and so on. The general approach to handle moving windows is, we first do grouping and aggregation as usual for each distinct value, and after that we perform a post-aggregation that aggregates the results from the previous step based on each window range. Consider the query mentioned above. First we get the sum(quantity) for each year, and then in the post-aggregation step, we just sum up the quantity for years from 2008 to 2012, and from 2011 to 2015, etc. If there are nested grouping operations inside each window group, the post-aggregation is also effective. For example, continuing with the above query, suppose for each year window, we need to find the number of books grouped by publisher. In the first step, we group books by each different year, and then in each group, we do a secondary grouping on publisher and count the books in each subgroup. The post-aggregation will integrate all subgroups in the five groups with year value from 2008 to 2012, from 2011 to 2015, etc, by summing up the results under the same publishers.

5. SEMANTIC OPTIMIZATION

In Q1, we group "book" by its descendant properties. Actually our algorithm also supports grouping by the properties appearing in other places in document, rather than descendants of an object.

EXAMPLE 5. *Consider the query Q2 to find the average price of books published in 2005, group by publisher first and then group by subject name. In this query, subject name appears as a property of the parent node of "book". To answer this query, we just match the twig pattern shown in Fig. 12(a) to the document tree, and extract values for each property using table index to form RS_{final}. Then grouping operation and aggregate function can be done normally in RS_{final}. The result structure is shown in Fig. 12(b).*



(a) Twig pattern (b) Result tree

Figure 12: Query Q2 and result tree

By investigating analytical queries, we find many of them group objects by their own properties. E.g. in Q1, we group books by publisher and then by year and price. *Publisher*, *year* and *price* are all the properties of *book*. With the semantic information on grouped object, and the relationship between grouped object and group-by properties, we can optimize tables to further improve the query performance.

5.1 Optimization 1: object/property table

Recall the table index we used in $VERT^G$ (e.g. the example tables shown in Fig. 4), we can see the "Label" field in each table stores the label of the corresponding property, while the "Value" field stores the value of the property. If we have knowledge on the object to which each property belongs, e.g. the object for properties "title" and "author" is "book", we can optimize the table to be object/property table, instead of the previous property table. The schema of the object/property table is:

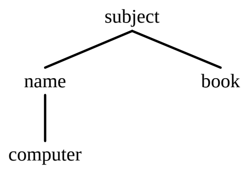
$$R_{object/property}(\text{Label, Value})$$

In the object/property table, the table name indicates which object and property the table is for. The "Label" field stores the label for the object, instead of the property, and the "Value" field stores the corresponding property value.

EXAMPLE 6. *For the bookstore document in Fig. 1, we can optimize the index property table to be object/property table. The table for object "book" and property "title" is shown in Fig. 13(a). Comparing the "book/title" table in Fig. 13(a) to the table for "title" shown in Fig. 4, we can find that the stored information in "Label" field is changed from "title" labels to "book" labels, whereas the "Value" field is not changed. Now when we process Q1 using the new table indices, we can simplify the twig pattern as shown in Fig. 13(b), by reducing quite a number of structure nodes and structure joins. After getting the labels for "book" during twig pattern matching, we can use the corresponding object/property tables to get values for "publisher", "year", "price" and "quantity", to form RS_{final}.*

$R_{book/title}$

Label	Value
(5:18,3)	Network
(19:34,3)	Database Systems
(35:48,3)	XML
(49:62,3)	Data Replication



(a) Object/property table for "book/title" (b) Twig pattern for Q1 with new table indices

Figure 13: Example for Optimization 1

5.2 Optimization 2: object table

In Optimization 1, for each object we maintain different table indices for different properties. E.g. for the object "book" in the bookstore document in Fig. 1, we have tables $R_{book/title}$, $R_{book/author}$, etc. Processing a group-by query involving different properties on the same object requires accessing multiple table indices. Those tables have the same "Label" value as there are for the same object. If we merge all object/property tables for the same object and single-valued properties to get object table, we can save the cost on the access and the search in multiple tables for the same object. Motivated by this, we have the second optimization. The schema for object table used in Optimization 2 is:

$$R_{object}(\text{Label, } Property*)$$

The table name indicates for which object the table is, the field "Label" stores the labels of each object and the rest fields store the names of each belonging single-valued property and the corresponding values. For multi-valued properties, we cannot merge them with other properties, so we keep the object/property table for multi-valued properties.

R_{book}

Label	Publisher	Title	Year	Price	Quantity
(5:18,3)	Hillman	Network	2003	45	30
(19:34,3)	Elco	Database Systems	2005	32	20
(35:48,3)	Elco	XML	2005	56	10
(49:62,3)	Elco	Data Replication	2006	60	25

$R_{book/author}$

Label	Value
(5:18,3)	Green
(19:34,3)	Smith
(19:34,3)	Cole
(35:48,3)	Smith
(49:62,3)	Wang

Figure 14: Example object table in Optimization 2

EXAMPLE 7. *Consider the bookstore document shown in Fig. 1. The index tables for "book" under Optimization 2 are shown in Fig. 14. We merge all the single-valued properties for "book" to R_{book}, and for the multi-valued property "author" we keep the object/property table. With the new optimization, for Q1 we only need to join $RS_{intermediate}$ in Fig. 8 with R_{book} once to get all the property values.*

Intuitively the parent node of each value is its property, while the parent node of each property can be considered as the corresponding object, but it is not always correct. E.g. in an XML document, "person" has property "name", and "name" is a composite property having two children "firstName" and "lastName". In this case, "firstName" and "lastName" should be the properties of object "person", though they are not the children of "person". Normally the semantics of object can be inferred from domain knowledge. Without such semantics, we can still process queries using Optimization 2, in which the parent node of each property is simply considered as an object, e.g. in the above example, we consider "name" as an object with two properties "firstName" and "lastName". Once we have more semantic information, we can include it to the table index and further improve the query performance. E.g. in a query to find the person whose "firstName" is "John". When we consider "name" as an object of property "firstName", we need to find the "name" whose property "firstName" has value of "John", and then join it with each "person". If we know the actual object for "firstName" is "person", we can directly find the "person" whose property "firstName" has value of "John".

6. EXPERIMENTS

In this section we present experimental results. First we conduct experiments to compare the query performance using $VERT^G$ without optimization, with Optimization 1 and with Optimization 2. Then we use $VERT^G$ Optimization 2 to compare with other approaches including relational approach, XQuery engine, and a recently proposed algorithm N-GB [14] on group-by query processing.

6.1 Experimental settings

We implemented all algorithms in Java. The experiments were performed on a dual-core 2.33GHz processor with 4G RAM. We used real-world data sets DBLP (91MB) and NASA (23MB), and a well known synthetic data set XMark [27] in our experiments. Note that DBLP data has a simple schema, while NASA data has a complex schema. The characteristics of queries used is shown in Fig. 15.

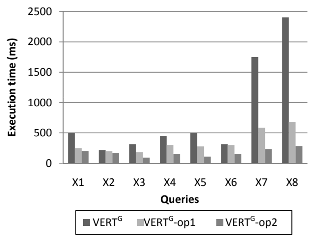
Query	Grouping levels	Grouping properties	Query	Grouping levels	Grouping properties
X1, N1, D1, XM1, NM1	1	1	XNR1, XNS1	1	2
X2, N2, D2, XM2, NM2	1	1	XNR2, XNS2	1	2
X3, N3, D3, XM3, NM3	1	2	XNR3, XNS3	2	3
X4, N4, D4, XM4, NM4	1	2	XNR4, XNS4	2	4
X5, N5, D5, XM5, NM5	2	3	XNR5, XNS5	3	5
X6, N6, D6, XM6, NM6	2	3	XNR6, XNS6	3	6
X7, N7, D7, XM7, NM7	2	4	DN1, DN2, DN3	1	1-2
X8, N8, D8, XM8, NM8	2	4	DN4, DN5, DN6	2	2-4
SX, SN, SD	1-6	1-6	DN7, DN8, DN9	3	3-6

Figure 15: Experimental queries with No. of grouping levels and No. of grouping properties

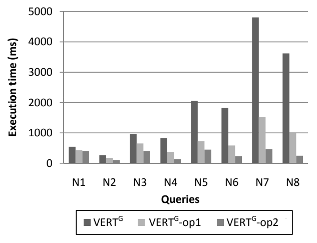
6.2 Comparison between $VERT^G$ without and with optimizations

6.2.1 Query performance

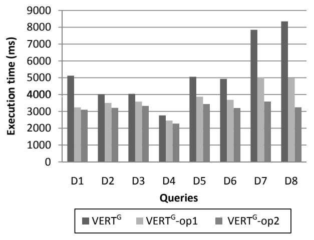
We process 8 queries in each document to compare the query performance between original $VERT^G$ algorithm and the two optimizations (named as $VERT^G$-op1 and $VERT^G$-op2). Queries X1-X8 are issued to Xmark document, N1-N8 to NASA document and D1-D8 to DBLP document. The experimental results on execution time are shown in Fig. 16.



(a) XMark data set



(b) NASA data set



(c) DBLP data set

Figure 16: Query performance comparison for $VERT^G$, $VERT^G$-op1 and $VERT^G$-op2

We can see that for all the queries $VERT^G$-op2 outperforms $VERT^G$-op1, and $VERT^G$-op1 outperforms $VERT^G$ without optimization. This validates the analysis in Section 5, which is $VERT^G$-op1 uses object/property table and can further simplify the query to improve the performance, and $VERT^G$-op2 combine object/property tables to object tables, so that the table accesses and the tuple searches are reduced and the performance is further improved.

6.2.2 Scalability as grouping levels increase

It is natural that the user issues a query with nested grouping. In this section we measure the time trend of our algorithm $VERT^G$ and its optimizations when the grouping levels increase. For each document, we select one type of query with predicates fixed and grouping levels varied. The result on the scalability is shown in Fig. 17.

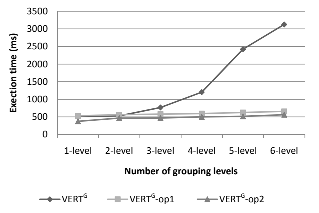
From the result we can see that running time for $VERT^G$ increases as the number of grouping levels increases. The reason is, if we group a set of objects by a new property, we have to include that property for pattern matching, which is time consuming. However, if we adopt $VERT^G$ with either $VERT^G$-op1 or $VERT^G$-op2, we only match the relevant objects, instead of each property node. As a result, the execution time increases slowly when more grouping levels are involved. $VERT^G$-op2 is better than $VERT^G$-op1 because we access less table indices in $VERT^G$-op2.

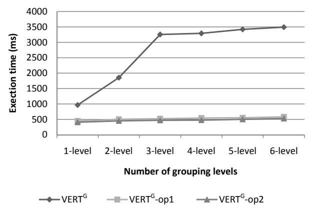
6.3 Comparison with other approaches

In this section, we compare our approach with other approaches including relational approaches, XQuery engine, and N-GB. We use $VERT^G$-op2 in our approach for the comparison.

6.3.1 Comparison with relational approaches

As mentioned in Section 1, relational approaches shred XML into relational tables and translate XML queries into SQL to query the database, and they support grouping in


(a) XMark data with SX


(b) NASA data with SN

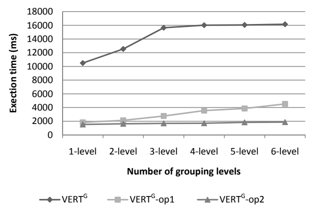
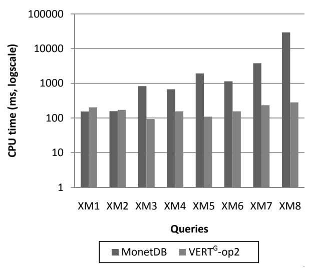

(c) DBLP data with SD

Figure 17: Scalability for $VERT^G$, $VERT^G$-op1 and $VERT^G$-op2

queries. In [14] they conducted experiments to show the bad performance using shredding method proposed in [25]. We tried another shredding method [17] in our experiments. We issued a query with 1-level grouping to an XMark document of 11MB. The relational database system takes around 10 minutes to return the answer, whereas in our approach such query only needs several seconds. When more query nodes are introduced, the processing time of the relational approach increases exponentially.

6.3.2 Comparison with XQuery

Besides, we take MonetDB [1], which is a well known efficient memory-based XQuery engine, for comparison. We used two data sets, XMark (11MB) and NASA (23MB), and conducted experiments on 8 queries in each data set (XM1-XM8 and NM1-NM8). All the queries contain grouping operation, and the group-by properties may not necessarily be the children or descendants of the object to be grouped. E.g. in NM8 for NASA data we group journals by subject, which is the ancestor node of "journal" in document. The experimental results are shown in Fig. 18 (Y-axis is in logarithmic scale).


(a) XMark data set

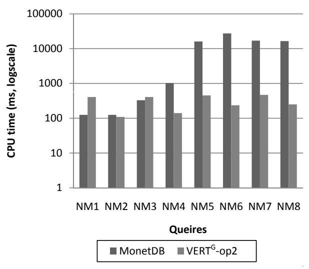
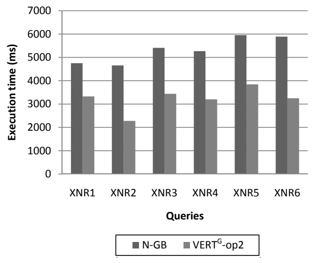

(b) NASA data set

Figure 18: CPU time comparison between *MonetDB* and $VERT^G$-op2

For both data sets, we can see that for 1-level grouping with one property, MonetDB performs well. However, when the number of group-by properties and the number of grouping levels increases, since XQuery needs to express such queries using nesting with multiple document retrievals and joins, the performance of MonetDB is affected. In XMark data set, the CPU time for MonetDB increases fast on XM3-XM8. In NASA data set, though the CPU time on NM3-NM4 is still relatively low, when we increase the number of grouping levels in NM5-NM8, the efficiency of MonetDB is significantly affected. Our approach, $VERT^G$-op2, outperforms MonetDB for those queries with multi-level groupings.

6.3.3 Comparison with N-GB

We also compare our work with a recently proposed algorithm N-GB ([14]) to process queries with grouping and aggregation. We take two data sets, XMark (111MB) and DBLP (91MB) for the comparison. For XMark data, we perform two sets of queries. The first set contains queries in which group-by properties appear in any positional relationship with the object to be grouped. E.g. we group journals by either its child property "year" or its ancestor property "subject". For this set of queries, our optimization can reduce the complexity during query processing, but we still need pattern matching to get query node occurrences in document. The second set of queries have group-by properties, output nodes and aggregate properties under the same object. In this case, we do not need to perform pattern matching, and the efficiency will be enhanced. For each query set, we have 6 queries with grouping levels varying among 1, 2 and 3. Fig. 19 shows the experimental results for XMark data.


(a) Group-by properties in random position

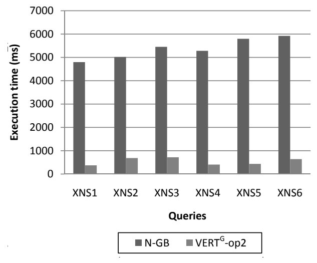

(b) Group-by properties in the same object as outputs

Figure 19: Execution time comparison between *N-GB* and $VERT^G$-op2 for XMark data

From the figure above we can see $VERT^G$-op2 always outperforms N-GB. For the first set of queries (Fig. 19(a)), $VERT^G$-op2 saves 30%-51% running time, and for the second query set (Fig. 19(b), this saving becomes 86%-93%.

We also used the real-world data DBLP to compare our approach and N-GB. We used 9 queries for DBLP data, which are DN1-DN9. Since N-GB assumes the answer tree can fit in memory, we allocated 1GB memory for JVM during experiments. The results are shown in Fig. 20. We can see from the figure, $VERT^G$-op2 outperforms N-GB for all kinds of queries. This result shows that our approach is efficient not only for complex documents (e.g. XMark), but also for flat documents (e.g. DBLP).

7. CONCLUSION

In this paper we analyzed the drawbacks of different existing approaches to process XML queries with grouping and aggregation, and proposed a novel algorithm, $VERT^G$, which can perform grouping operation and compute aggregate functions in XML queries with complex predicate. The main technique of $VERT^G$ is to introduce table index during XML query processing. After processing XML queries

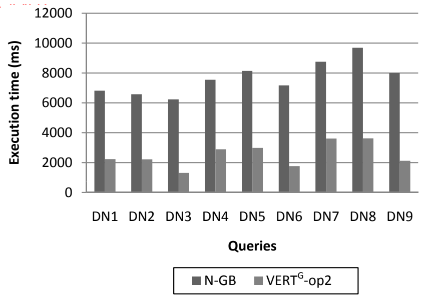


Figure 20: Execution time comparison between *N-GB* and $VERT^G$-op2 for DBLP data

over documents natively using any pattern matching algorithm, e.g. *VERT*, $VERT^G$ extracts actual values for relevant nodes with table indices and performs grouping and aggregation. Furthermore, we proposed two semantic optimizations to table index, which can significantly enhance the query processing performance. We conducted experiments to compare our approach with a relational approach, a well known XQuery engine and a recently proposed algorithm, to show the advantages of our approach.

In future work, we plan to investigate real-life queries and further optimize the table index so that the relationship between relevant objects can be efficiently discovered using index, instead of searching the document. Also we will extend our approach to handle queries with ID references, and queries across multiple XML documents.

8. REFERENCES

[1] MonetDB. http://monetdb.cwi.nl/.

[2] A. Berglund, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Simeon. XML path language XPath 2.0. W3C Working Draft, 2007.

[3] K. S. Beyer, D. D. Chamberlin, L. S. Colby, F. Özcan, H. Pirahesh, and Y. Xu. Extending XQuery for analytics. In *SIGMOD Conference*, pages 503–514, 2005.

[4] K. S. Beyer, R. Cochrane, L. S. Colby, F. Ozcan, and H. Pirahesh. XQuery for analytics: Challenges and requirements. In *XIME-P*, pages 3–8, 2004.

[5] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML query. W3C Working Draft, 2003.

[6] V. Borkar and M. Carey. Extending XQuery for grouping, duplicate elimination, and outer joins. In *XML Conference and Expo.*, 2004.

[7] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *SIGMOD*, pages 310–321, 2002.

[8] P. Buneman, S. B. Davidson, W. Fan, C. S. Hara, and W. C. Tan. Keys for XML. In *WWW*, pages 201–210, 2001.

[9] S. Ceri, S. Comai, E. Damiani, P. Fraternali, S. Paraboschi, and L. Tanca. XML-GL: a graphical language for querying and restructuring XML documents. In *WWW*, pages 1171–1187, 1999.

[10] A. Deutsch, Y. Papakonstantinou, and Y. Xu. The next logical framework for XQuery. In *VLDB*, pages 168–179, 2004.

[11] D. Engovatov. XML query (XQuery) 1.1 requirements. W3C Working Draft, 2007.

[12] L. Fegaras, D. Levine, S. Bose, and V. Chaluvadi. Query processing of streamed XML data. In *CIKM*, pages 126–133, 2002.

[13] T. Fiebig and G. Moerkotte. Algebraic XML construction and its optimization in natix. *World Wide Web*, 4(3):167–187, 2001.

[14] C. Gokhale, N. G. 0003, P. Kumar, L. V. S. Lakshmanan, R. T. Ng, and B. A. Prakash. Complex group-by queries for XML. In *ICDE*, pages 646–655, 2007.

[15] G. Gou and R. Chirkova. Efficiently querying large XML data repositories: A survey. *IEEE Trans. Knowl. Data Eng.*, 19(10):1381–1403, 2007.

[16] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-total. In *ICDE*, pages 152–159, 1996.

[17] T. Grust, M. van Keulen, and J. Teubner. Accelerating XPath evaluation in any RDBMS. *ACM Trans. Database Syst.*, 29:91–131, 2004.

[18] M. H. Kay. Positional grouping in XQuery. In *XIME-P*, 2006.

[19] W. Kim. On optimizing an sql-like nested query. *ACM Trans. Database Syst.*, 7(3):443–469, 1982.

[20] N. May, S. Helmer, and G. Moerkotte. Strategies for query unnesting in XML databases. *ACM Trans. Database Syst.*, 31(3):968–1013, 2006.

[21] N. May and G. Moerkotte. Efficient XQuery evaluation of grouping conditions with duplicate removals. In *XSym*, pages 62–76, 2007.

[22] W. Ni and T. W. Ling. GLASS: A graphical query language for semi-structured data. In *DASFAA*, pages 363–370, 2003.

[23] S. Paparizos, S. Al-Khalifa, H. V. Jagadish, L. V. S. Lakshmanan, A. Nierman, D. Srivastava, and Y. Wu. Grouping in XML. In *EDBT Workshops*, pages 128–147, 2002.

[24] C. Re, J. Siméon, and M. F. Fernández. A complete and efficient algebraic compiler for XQuery. In *ICDE*, page 14, 2006.

[25] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *VLDB*, pages 302–314, 1999.

[26] H. Wu, T. W. Ling, and B. Chen. VERT: A semantic approach for content search and content extraction in XML query processing. In *ER*, pages 534–549, 2007.

[27] XMark. An XML benchmark project. http://www.xml-benchmark.org, 2001.

[28] M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura. XRel: a path-based approach to storage and retrieval of XML documents using relational databases. *ACM Trans. Internet Techn.*, 1(1):110–141, 2001.

[29] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On supporting containment queries in relational database management systems. In *SIGMOD Conference*, pages 425–436, 2001.