Paper 928

Complex Group-By Queries for XML

C. Gokhale[+], N. Gupta[+], P. Kumar[+], L.V.S. Lakshmanan[*], R. Ng[*], and B.A. Prakash[+]
[+] Indian Institute of Technology, Bombay
[*] University of British Columbia, Canada

1 Introduction

The popularity of XML as a data exchange standard has led to the emergence of powerful XML query languages like XQuery [21] and studies on XML query optimization. Of late, there is considerable interest in analytical processing of XML data (e.g.,[2, 3]). As pointed out by Borkar and Carey in [3], even for data integration, there is a compelling need for performing various group-by style aggregate operations. A core operator needed for analytics is the group-by operator, which is widely used in relational as well as OLAP database applications. XQuery requires group-by operations to be simulated using nesting [2].

Studies addressing the need for XML grouping fall into two broad categories: (1) Provide support for grouping at the logical or physical level [6] and recognize grouping operations from nested queries and rewrite them with grouping operations [4, 5, 9, 12]. (2) Extend XQuery FLWOR expressions with explicit constructs similar to the group-by, order-by and having clauses in SQL [3, 2]. However, direct algorithmic support for a group-by operator is not explored.

In this paper, we focus on efficient processing of a group-by operator for XML – with the additional goal of supporting a full spectrum of aggregation operations, including holistic ones such as `median()` [8] and complex nested aggregations, together with having clause, as well as moving window aggregation.

Consider the simple catalogue example in Figure 1. This can be part of an input XML database, or intermediate result of a query. The catalogue is heterogeneous: it contains information about books, music CDs, etc. Books are organized by `Subject`, e.g., `physics, chemistry`. For each book, there is information on its `Title`, `Author`, `Year`, `#Sold`, `Price`, (publisher) `Name`, etc. Books may have multiple authors. The data value at a leaf node is shown in italics. The node id of a node is also shown for future discussion.

Consider the following nested group-by query Q_1. While we could follow the syntax proposed by [2], syntax not being our main focus, we use a more concise form. We also omit the selection part of the query, and just focus on the aggregation part.

```
group //Book
by //Name return (
    //Name, avg(/Price), count(*)
    then by /Year return (
        /Year, median(/#Sold)
    ) )
```

The intent of the query is to group `Book` nodes first by (publisher) `Name`. For each group, we get the average `Price`, and the number of `Book` nodes in the group. Moreover, for each publisher group, the book nodes are further sub-grouped by `Year`. For each of these nested groups, the median `#Sold` per year is returned.

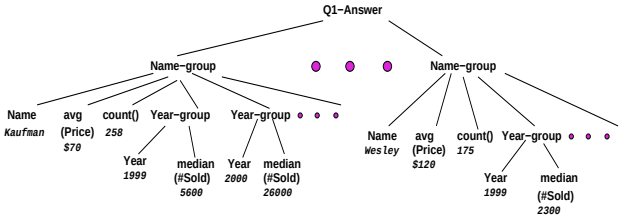


Figure 2. Partial Result of Q_1

Figure 2 shows the answer to query Q_1 for the (partial) data in Figure 1. The first group shown, for instance, is for `Name = Kaufman`. Among the 258 books in this group, the average price is $70. These books are further sub-grouped by `Year`. For each year that appears in the input data, the median number of copies sold is also returned (e.g., 5600 for 1999). We can enhance nested group-by query Q_1 with two features, as illustrated by query Q_2:

```
group //Book
by //Name
having count(*) ≥ 100 return (
    //Name, avg(/Price), count(*)
    then by /Year return (
        /Year(10,5), median(/#Sold)
    ) )
```

In Q_2, the having-clause for the outer block removes publishers with the total number of book nodes less than 100. Besides, we form moving windows over years – with each window having a width of 10 years and a step size of 5 years (e.g., [1990,2000], [1995,2005], etc.). While in

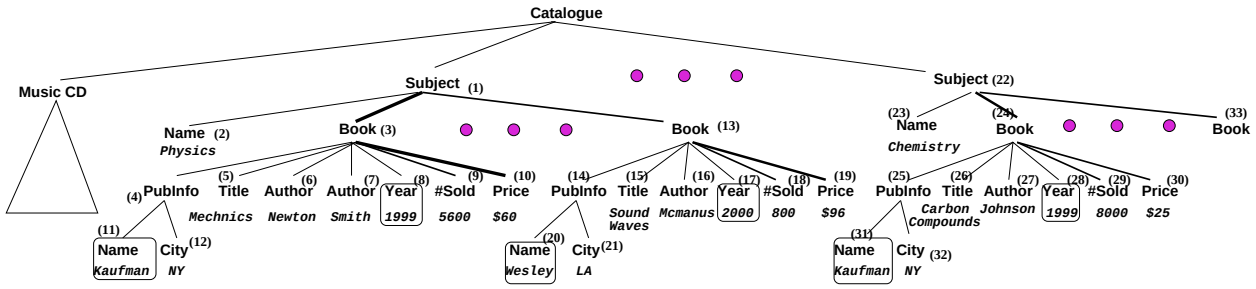


Figure 1. The Catalogue Example

the next section we will present a more comprehensive set of moving window options, it should be easy to appreciate the value of supporting nested group-bys with having clauses and moving windows for XML querying. In principle, all value aggregations required of XML can be obtained by shredding it to relations and using SQL (the "SQL approach"). We examine this issue empirically in Section 7 with an emphasis on queries involving grouping together with nesting. Indeed, owing to XML's inherent hierarchical nature, nested group-by (e.g., query Q_1) is a fundamental type of group-by that merits study. In our experiments, we observed an order of magnitude difference between the performance of the SQL approach (using Oracle) and ours. We make the following contributions.

- We propose a framework for expressing complex aggregation queries on XML data featuring nested group-by, having cluase, and moving windows (Section 3).
- We develop a disk-based algorithm for efficient evaluation of queries involving any subset of the above fearures (Section 5).
- We discuss the results of a comprehensive set of experiments comparing our approach with that of shredding XML into relations and using SQL, and with those of Galax [7] and Qizx [17], validating the efficiency of our algorithm and the effectiveness of the optimizations developed (Section 7).

Related work appears in the next section. Section 8 summarizes the paper and discusses future work.

2 Related Work

While for relational data, SQL provides explicit support for group-by, XQuery requires us to simulate it using nesting. It has been noted that this leads to expressions that are hard to read, write, and process efficiently [2, 3]. Beyer et al. [2] and Borkar and Carey [3] propose syntactic extensions to XQuery FLOWR expressions to provide explicit support for group-by. They also demonstrate how related analytics such as moving window aggregations and cube can also be expressed in the extended syntax. Beyer et al. report preliminary experimental results indicating better performance than simulating grouping via nesting. None of these papers discuss algorithms for directly computing group-bys (with possible nesting, having, and moving windows).

A second line of studies investigates how to support group-by at a logical or physical level [6], and detect group-bys from nested queries and rewrite them with explicit grouping operations [4, 5, 9, 12]. However, detecting grouping inherent in nested queries is challenging and such queries are hard to express and understand. In particular, the focus of [12] is on structural aggregation by node types as opposed to value aggregation. Studies by Fiebig and Moerkotte [6], Pedersen et al. [13], and Deutsch et al. [4] all consider using query optimization-style rewrite rules for various kinds of grouping. The transformed query plan would be based on nested loops.

There is an extensive body of work on efficient computation of group-by and cube queries for relational data (e.g., [8, 10]). These algorithms are not directly applicable to hierarchical data especially when group-by elements (β's) may involve combination of forward and backward axes and aggregations on values may be nested and may occur at multiple levels (e.g., Q_2). Of course, by shredding XML to relations, all such queries can be expressed in SQL. The performance impact of this approach compared with our direct approach is discussed in Section 7.

Finally, [15, 19] study XPath selectivity estimation to obtain statistical summaries and approximate answers for XPath expressions. They do not directly support exact computation of group-bys.

3 Class of Nested Group-bys

3.1 General Form of 1-level Nesting and Examples

The examples discussed so far are instances of the general form of a one-level nested group-by query below.

group α
where $Cons$
by $\beta_1^{out}(mw_1^{out}) \ldots \beta_k^{out}(mw_k^{out})$
having $AggCons^{out}$ **return (**
$\beta_1^{out}, \ldots, \beta_k^{out}, agg_1^{out}(\gamma_1^{out}), \ldots, agg_m^{out}(\gamma_m^{out})$
then by $\beta_1^{in}(mw_1^{in}), \ldots, \beta_p^{in}(mw_p^{in})$

Figure 3. An Example Illustrating Node Type Inversion

having $\mathrm{AggCons}^{\mathrm{in}}$ **return** (
$\beta_1^{\mathrm{in}}, \ldots, \beta_p^{\mathrm{in}}, \mathrm{agg}_1^{\mathrm{in}}(\gamma_1^{\mathrm{in}}), \ldots, \mathrm{agg}_q^{\mathrm{in}}(\gamma_q^{\mathrm{in}})$
))

Path Expressions: Here, α is an absolute XPath expression, while β's and γ's are relative to α. That is, for every node \$x that binds to α, those nodes that bind to the related β's and γ's are returned.

Note that the β's and γ's are not restricted to be descendants of α. Hierarchy "inversion" is supported by using the keywords (axes) **par** and **anc** to denote the parent and the ancestor node type of α. Consider the catalogue database of Figure 3, which is more complex than the one previously discussed. The database contains details of books classified by `Publisher`, `Location`, `Year`, and other pieces of information such as `Review`. Consider the following nested group-by query Q_3.

group //Book
by anc::Publisher/Name, Subject return (
anc::Publisher/Name, Subject,
count(distinct(anc::Location/Name)), count(*)
then by par::Year/Value **return** (
par::Year/Value, median(/Price)
))

The intent of the query is to group `Book` nodes first by (publisher) `Name` and `Subject`, and then to further sub-group by `Year`. For each outer group, it returns the number of locations the publisher is in and the number of books in the group. For each inner sub-group, it finds the median price of books. Observe that the β and γ node types are related to the α `Book` node by forward (child/descendant) or *backward* (parent/ancestor) relationships or a combination thereof (e.g., `par::Year/Value` and `count(distinct(anc::Location/Name))`).

Aggregation Operations: All of $\mathrm{agg}^{\mathrm{out}}$'s and $\mathrm{agg}_{\mathrm{in}}$'s are aggregation operations such as `min()`, `count()`, `avg()`, `median()`, etc. Some of the γ elements are multi-valued (e.g., `Author`, `Review`). Aggregations applied on such γ's can be nested. (Nested group-bys and nested aggregations are orthogonal concepts.) For example, if medianMax(/Review/Rating) was specified in the inner block of query Q_3, the query would first compute for *each* book, the highest or best rating. Then it would obtain the median among all the best ratings of the books in the group (for a given publisher, subject, and year). Simiarly, minCount(/Review) obtains the minimum number of reviews any book received in a group. As a last example of nested aggregation, `spread(Rating)`$=_{\mathrm{def}}$ `max(Rating)` $-$ `min(Rating)` combines `min()` and `max()`.

Aggregation Conditions: Cons, $\mathrm{AggCons}^{\mathrm{out}}$ and $\mathrm{AggCons}^{\mathrm{in}}$ are sets of conditions. Cons in the where clause are the usual node-level selection conditions. $\mathrm{AggCons}^{\mathrm{out}}$ and $\mathrm{AggCons}^{\mathrm{in}}$ are sets of aggregation conditions of the form $\mathrm{agg}_i(\gamma_i)\,\theta_i\,c_i$, where $\theta_i \in \{=, \neq, >, <, \geq, \mathrm{leq}\}$, and c_i is a constant. With the use of the having clause, iceberg queries can be easily expressed in the proposed framework.

Moving windows: Moving window queries have been studied extensively for relational databases [18]. We adopt a similar framework for XML data here. The mw^{out}'s and mw_{in}'s denote moving window specifications. If the corresponding β_i is non-numeric, mw_i is null. Otherwise, mw_i can be null or of the general form: $mw_i \equiv (width, step, winType, domType)$. The two most prevalent parameters of a moving window are the width (i.e., $width$) and step size (i.e., $step$) of the window. The parameter $winType$ can either be `cummulative` or `fixedWidth`. For a cummulative moving window, the first window is of size $width$; subsequent windows expand in size by $step$ at a time. For a fixed width moving window, the window moves by $step$ units at a time. There are two kinds of domains: `active` or `standard`. For an active domain, only the values that appear in the database are included in the formation of a moving window. For a standard domain, the entire interval $[min, max]$ is used, where min and max correspond to the minimum and maximum values that appear in the data. E.g., consider the following query Q_4 which modifies Q_3.

group //Book
by anc::Publisher/Name, Subject return (
anc::Publisher/Name, Subject, count(*)
then by par::Year/Value(2,5,fixedWidth,active) **return** (
par::Year/Value, avg(/#Sold)
))

Q_4 specifies an outer group involving each combination of publisher and subject. For the inner groups, it picks "samples" of only 2 (consecutive) years in every 5 years. As an example, let the set of years appearing in Book nodes that satisfy the where clause be $\{91, 93, 95, 96, 97, 00, \ldots, 06\}$. Then the sequence of moving windows that Q_4 obtains is: {91,93}, {00,01} and {05,06}. If Q_4 is specified with $domType$ being standard, then the set of year values used is $\{91, \ldots, 06\}$, and the corresponding sequence becomes: {91,92}, {96,97}, {01,02}, and {06}.

In the full paper, we also show how percentiles can be expressed and evaluated within the same framework.

While we have given the template for a one level nesting, it is trivial to generalize it to arbitrary levels. Further, to simplify exposition, we assume, unless otherwise specified, that a moving window specification is has $winType =$ fixedWidth and $domType =$ standard, and so omit these

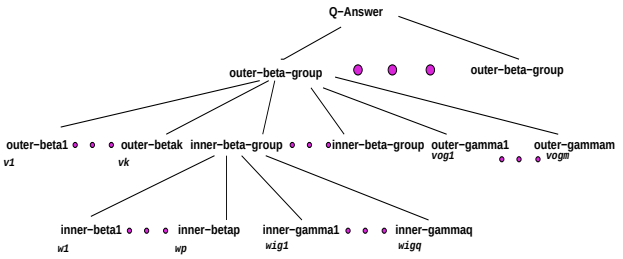


Figure 4. Answer tree of a 1-level nested group-by

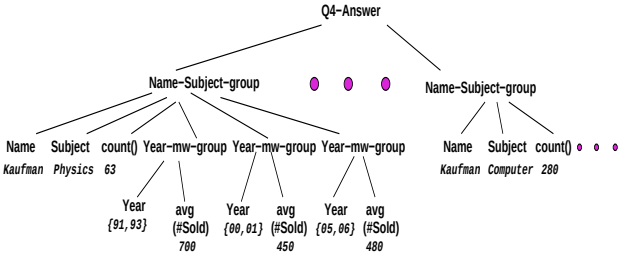


Figure 5. (Partial) Result of Q_4

components from the specification; Q_2 is an example of this.

Figure 4 depicts the form of answer tree for the query template given in this section. Figure 5 shows the result of Q_4 against the input data of Figure 1.

4 Overview of a Group-by Operator

We first consider a single block group-by. In [1], we propose a group-by operator and develop a main-memory based algorithm, called Merge-GB, for computing it. In this section, we give an overview of Merge-GB (which does not support nesting, having, or moving windows). It consists of three steps: (i) initialization, (ii) the merge* phase where the node merge operation is repeatedly applied, and (iii) the answer extraction step.

4.1 Algorithm MERGE-GB: Initialization

Given a group-by query identifying node types α, β's, and γ's, we prune nodes other than those types. The outcome of this step is the creation of a "canonical tree" T_{can}, containing only these nodes but following the input data tree structure. We use the following running example Q_7 throughout this section: ''`group //Book by //Year, return Year, median(#Sold), spread(Price), count(*)`'' We use the input tree shown in Figure 1. Figure 6 shows the canonical tree after initialization.

MERGE-GB computes group-bys by repeated merging of nodes of the same type. The α (e.g., `Book`) nodes are

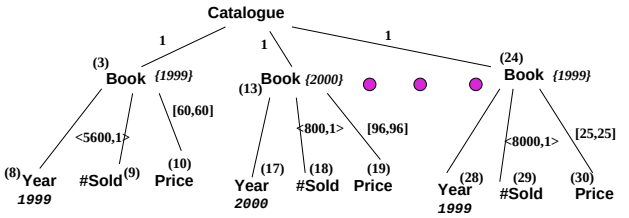


Figure 6. Canonical Tree for Q_7 after Initialization

merged based on equality of the associated β values, which serve as their *group-by label*. In addition, child nodes of α nodes that are of a given type are merged the same way (e.g., `Price, #Sold`, etc.).

Counter Initialization: Based on the aggregate functions in the query, an appropriate counter is associated with certain edge types. E.g., for `count(*)`, we associate a simple counter with each edge of type (`Catalogue`, `Book`) and initialize it to 1. All these (`Catalogue`, `Book`) edges are eventually merged, and the counter is updated to give the answer to the aggregation `count(*)` in Q_7. For `spread(Price)`, the edge type (`Book`, `Price`) is associated with a counter [`min`, `max`] containing the minimum/maximum price of books in a group. For the first edge, this is initialized to $[60, 60]$ (cf: Figures 1 and 6). When `Price` nodes are merged, this counter is updated appropriately (see Section 4.2). For `median(#Sold)`, since it's a holistic function, we need a frequency table as the counter, which keeps track of the frequency for each value. In Figure 6, the first book has a frequency table edge counter $\langle 5600, 1\rangle$, indicating that there is 1 book with 5600 copies sold.

4.2 The Merge* Phase of MERGE-GB

When nodes are merged, counters get updated. Counter update differs for nodes that were siblings in the original data tree compared to nodes that weren't. E.g., in Figure 6, all the `Book` nodes are siblings while all the `Price` nodes are non-siblings. For siblings, the counters can be "summed" together. For Q_7, Figure 7 shows the intermediate stage when all the sibling `Book` nodes are merged. Suppose there are 258 `Book` nodes with the group-by label 1999 and 317 with group-by label 2000. These respective sets of nodes are merged in Figure 7. The relevant child nodes of *all* the 1999 `Book` nodes in Figure 6 are now consolidated to have the same parent. The situation for `Year = 2000` is similar. The edge counters are updated to reflect the summation. The counter on the (`Catalogue`, `Book`) edge in Figure 7 yields `count(*)`. We use procedure `domergesiblings()` (not shown) for implementing this.

The next phase is to merge non-sibling nodes and

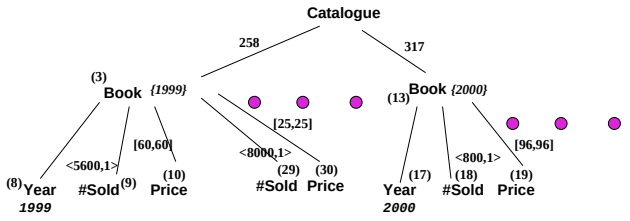


Figure 7. End of the Merge Siblings Pass for Q_7

update their counters, for which we use procedure `domergenonsiblings()` (not shown). For our example, all the `Price` sibling nodes in Figure 7 (which were non-siblings in Figure 6) are merged. For brevity, the resulting tree is suppressed. Suppose of the 258 1999-books in Figure 7, the minimum price is \$25 and the maximum price is \$130. Then the counter for the first (`Book`, `Price`) edge from left is updated to $[min\{60, 25, ..., 130, ...\}, max\{60, 25, ..., 130, ...\}] = [25, 130]$. Similarly, the frequency table of the first (`Book`, #Sold) edge is updated to say $\{\langle 5600, 5\rangle, \langle 8000, 3\rangle, \langle 200, 10\rangle, \ldots\}$. In effect, this says, the price of the 258 1999-books is in the range $[25, 130]$, there were 5 books which sold 5600 copies, 3 for 8000 copies, etc. Notice that we need frequency table for `#Sold` since median is required.

Both procedures `domergesiblings()` and `domergenonsiblings()` are invoked in Figure 8. The group-by summary tree after merging non-siblings contains the necessary information to construct the answer for Q_7. We omit the obvious detail on answer tree construction.

5 A Disk-based Algorithm for Nested Group-bys with Having Clauses

In this section, we develop a disk-based algorithm for processing nested group-by queries. Section 5.2 deals with having clause, and Section 6 with moving windows.

5.1 Dealing with Nesting

We assume the worst case, where there is no associated index for quickly selecting the required node types, and assume we have to scan the input data tree with nodes stored in pre-order fashion. As the input data is scanned, all nodes that are not of α, β's and γ's node types are ignored. The answer tree is constructed with aggregation computed on-the-fly as much as possible. For simplicity of presentation, we assume that the answer tree fits in main memory.

Proceesing β nodes: Whenever a β node is encountered, the algorithm checks to see if this is a new value. If so, the value is used to create the corresponding group-by label in the answer tree. Otherwise, appropriate updating may take place. For instance, for Q_1 and the tree shown in Figure 1, the first time `Name = Kaufman` is encountered, a new `Book` group node is created in the answer tree as a new child node of the root, with $\{$`Kaufman`$\}$ as the group-by label. This node, in turn, has 3 child nodes: a `Name` child with value `Kaufman` and child nodes for `avg(Price)` and `count(*)` with associated counters properly initialized.

As more input data are scanned, γ nodes `Price` are encountered. Let us defer the discussion on γ nodes. Instead, let us consider the processing of the inner β nodes `Year`. Exactly like how outer β nodes are processed, if a new `Year` value is encountered, a new `Year` group node is created. E.g., the first time `Year = 1999` is encountered, a new group node is created as a child node of the appropriate `Name` group node, with group-by label $\{$`Kaufman,1999`$\}$. This node in turn has two child nodes. The first child node is `Year` with value 1999. The second child node is `median(#Sold)` with the counter initialized approrpriately.

Q_1 discussed so far is simpler than the general case. E.g., consider Q_3. Ignore for now the node inversion part (i.e., `anc::Publisher`). Here there are two β elements – `Name` and `Subject`. As discussed above, there is checking to see if a new `Name` or a new `Subject` is encountered, which is implemented by hashing. Furthermore, associated with each β is a list of values. This list facilitates the creation of group nodes. For instance, when a new `Subject` value s is encountered, then there is a new group node created corresponding to the pair (n, s) for each `Name` value n in the list of `Name` values seen so far. The appropriate group-by label is created as well.

For node inversion, one complication is that the β node (or γ node) may be read before the α node. This is easily dealt with by using a dummy α node. That is, the nodes in the answer tree are created in exactly the same way, except there may be nodes with missing values to be filled in later when they are read. The set of nodes to be created in this manner is completely determined by the query, as discussed before.

Processing γ nodes and Updating counters: There are two cases for actions to be taken on reading a γ node. If the aggregate operation is holistic, then all the values of the γ node for the specific β combination have to be collected before the aggregation can be carried out. As these values are being read one by one, they are accumulated in a frequency table in main memory. However, our algorithm does not assume that all the frequency tables will fit in main memory simultaneously. Thus, these values are written out to a file, called a gamma file. For Q_1, `median(#Sold)` is a holistic aggregation, and each `#Sold` value encountered is written out to the gamma file with the associated α-id. As

```
Algorithm NGB-Disk
Input: XML tree-file, query
Output: answer tree
  (1) Open input file and initialize answer tree.
  (2) for each node encountered {
  (3)   if the node is not an α, β, or γ node, skip the node
  (4)   if it is an α node {
  (5)     update appropriate counter if count(*) is specified
  (6)     if node type inversion is involved, update the dummy α node }
  (7)   if it is a β node {
  (8)     if a new β value is encountered
  (9)       create a new set of group nodes with the group-by label
  (10)    otherwise, update appropriate counters if count(*) is specified }
  (11) if it is a γ node {
  (12)    if the aggregation is holistic,
  (13)      output the value and the α-node id to the gamma file
  (14)    otherwise {
  (15)      if the parent-id associated with the counter is the same
              as the parent-id of the current node, invoke domergesiblings()
  (16)      otherwise, invoke domergenonsiblings() } }
  (17) } /* end-for */
  (18) scan through the gamma file, using the α-node ids to form groups
  (19) use domergenonsiblings() to compute the aggregation for each group
  (20) put the computed values in the appropriate ndoes of the answer tree }
```

Figure 8. Algorithm NGB-Disk

shown in Figure 8, there is a separate gamma file pass after all the input has been read.

If the aggregation operation is not holistic, then the aggregation can be computed on-the-fly by updating the appropriate counters. The updating can be done by invoking either the domergesiblings() or domergenonsiblings() procedures discussed earlier. To decide between which procedure to use, the algorithm compares the current parent-id with the stored parent-id associated with the last update of the counter. If the two id's match, then the current γ node is a sibling of the last γ node, and domergesiblings() is invoked; else domergenonsiblings() is invoked.

To complete the discussion of processing Q_1, when the `Price` nodes are read, for `avg(Price)` two counters – sum and count – are maintained and updated as usual. At the end, the average value can be computed from the two. For `count(*)`, the first time when `Name = Kaufman` is encountered, the required set of nodes are created in the answer tree as discussed before. Furthermore, the counter associated with `count(*)` is initialized to 1. Next time when `Name = Kaufman` is encountered again, the counter is incremented. Finally, for `median(#Sold)`, a gamma file is used. Each γ value is associated with the α-id so that in the final pass when these values are re-read into main memory, the procedure domergenonsiblings() can be used to compute the median. For our example, a frequency table is used to aggregate the `#Sold` values, from which the median can be computed.

Recall that the proposed framework supports nested aggregation. Suppose that `medianMax(Review/Rating)` is specified in Q_3. As the `Review/Rating` nodes are read, domergesiblings() is used to compute the highest rating for that particular `Book` group. This highest rating is then written out to the gamma file and processed in the final pass to compute the median as discussed in the previous paragraph.

5.2 Dealing with a Having Clause

We first consider a having clause in an unnested group-by query and then generalize to nested queries.

Anti-monotonic early pruning: In an unnested group-by query, the obvious naive solution to process a having clause is to compute the aggregation in the clause and then to check if the aggregation result satisfies the constraint. However, for some constraints, it is possible to apply early pruning. As studied in [11], an anti-monotonic constraint is a constraint that will remain false once it is first violated. For instance, if the having clause includes the constraint: `max(Price)` $\leq$ `10`, then as soon as we have encountered a single item in that group with price > 10, then no item encountered later can reverse the violation of the constraint. Other examples include: `count(*)` $\leq$ `100`, `min(Price)` $\geq$ `100`, `sum(Price)` $\leq$ `1000`. The class of anti-monotone constraints has been extended by the notion of convertible constraints studied in [14]. Both anti-monotonic and convertible constraints allow early pruning of groups violating the constraint.

With Nesting: Let us first consider how early pruning can be incorporated into Figure 8. First, whenever a counter is updated in line (15) or (16), the constraint is checked if it is anti-monotonic or convertible. If the constraint is already violated, then the corresponding β group is flagged. Lines (10), (15) and (16) check if the group to be updated is a flagged β group. If so, no updating is required. E.g., suppose in Q_2 that the having clause is `count(*)` $\leq$ `100` instead. Then once a particular β group (i.e., `Name` in this example) is flagged, there is no need to update the counters corresponding to `count(*)`, and `avg(/Price)`. We use a hash table to map a β group to a corresponding node in the answer tree. Hereafter, we use $hash(\beta_v)$ to return the corresponding node in the answer tree for a particular β value β_v. Each β node has a flag that indicates whether the group has been flagged due to the violation of a having clause.

Similar to the skipping of outer γ's, all the processing within the inner query can be skipped once a β group has been flagged. For Q_2, once the outer having clause fails, the processing for the inner β (i.e., `Year`) and the inner γ (i.e., `#Sold`) can be skipped. Thus, to process a having clause, lines (7) and (11) in Figure 8 are modified with the condition that the nodes are not flagged.

So far the discussion focuses on the situation when anti-monotonic early pruning has flagged a β node. However, a similar kind of processing can be applied when there is an outer having clause. Recall that lines (18) and (20) deal with holistic aggregations. A condition is added to make sure that a holistic aggregation in *an inner block is not processed*

until the having clauses in all the outer blocks have been processed. To have the maximum benefit, it is not sufficient to have a single gamma file for all the holistic aggregations. In the best case, for each β group in a query block with a having clause, there should be a separate gamma file for each holistic aggregation. For Q_2, this corresponds to the situation when each publisher `Name` has a separate gamma file. (The `#Sold` values of all the years for a particular publisher shares the same gamma file.) In this way, if the β group is flagged because of failing the having clause, the entire gamma file need not be re-read. This leads to the following guarantee for minimizing I/O's.

Lemma 1 With the aforementioned setup, a γ value in an inner block that does not appear in the answer is not read after the value was written into the appropriate gamma file.

6 Dealing With Moving Windows

First, we consider the simpler case of no having clause in the query (but possibly with nested group-bys). We propose two evaluation strategies. Later we consider more genaral cases.

6.1 The Repeated-aggregation Strategy

A natural strategy for processing a moving window $mw \equiv (width, step, winType, domType)$ is to enumerate all the groups apriori, and then to aggregate for all these groups as if they were independent. E.g., first consider a standard domain moving window, i.e., $domType = standard$. Because the range is known without reading the data, all the groups that are specified by mw can be enumerated apriori. For these groups, the corresponding nodes are created in the answer tree even before the data are read.

E.g., let $mw_1 \equiv (5, 1, fixedWidth, standard)$ be specified for `Year` and let that range of values be [1991,2006]. Thus, all the groups can be enumerated apriori, e.g., 1991-1995, 1992-1996, and so on. With these groups created, the one extension to Figure 8 that is necessary is line (7). When a β node with a particular value β_v is read, there may be multiple groups that have to be engaged. For instance, for mw_1, if $\beta_v = 1993$, counters of the three groups 1991-1995, 1992-1996 and 1993-1997 should be updated. This is implemented by extending the hash index $hash(\beta_v)$ so as to direct the updating of all the appropriate group counters. This strategy is called *repeated-aggregation*. The case $winType = cumulative$ is handled similarly.

When $step > width$, some β_v values may not participate in the aggregation, and for them $hash(\beta_v)$ returns a null list of locations.

So far we have considered $domType = standard$. The situation for $domType = active$ is handled in like manner.

6.2 The Rolling-over Strategy

One potential drawback of repeated aggregation is that aggregation may need to be repeated many times. E.g., for the above mw_1 example, for a specific β_v value, say 1993, since this year value is engaged with the three groups 1991-1995, 1992-1996 and 1993-1997, all the 1993 values are essentially aggregated three separate times. In general, the larger the ratio $width/step$, the more often the aggregations are repeated. The rolling-over strategy avoids this potential inefficiency by making sure that each β value is aggregated at most once.

The strategy consists of 2 main steps, given a query Q with at least one moving window. (1) Run $Q_{\overline{mw}}$, which is formed by removing the moving window specification in Q. Essentially, $Q_{\overline{mw}}$ represents a degenerate moving window with $width = 1$ and $step = 1$. The outcome is an intermediate answer tree $T_{\overline{mw}}$. (2) Use $T_{\overline{mw}}$ to compute the moving window part of Q and to return the final answer tree. The specific computation depends on the nature of the aggregate function. First, consider a distributive function such as `sum`. Once the sum for a particular window is calculated (e.g., for 1991-1995), the sum for the next window (e.g., 1992-1996) is obtained by subtracting the sums for those years that left the window (e.g., 1991) and adding the sum for those years that entered the window (e.g., 1996). If the aggregate function is algebraic such as `avg`, by breaking it into corresponding distributive functions `sum` and `count`, we can use the same technique. If the aggregate is holistic like `median`, then the counter used is the frequency table. The frequency table for 1992-1996 is obtained from that of 1991-1995 by removing rows corresponding to 1991 and adding rows corresponding to 1996. Active domain and cumulative windows are handled similarly. For the rolling-over strategy, we have:

Lemma 2 For each value of β, the rolling-over strategy guarantees that aggregation is done at most once.

While the above lemma guarantees that aggregation is done at most once for each value of β, the rolling-over strategy may perform aggregation for values that are not needed in the answer, when $width < step$. It may incur unnecessary overhead in first executing $Q_{\overline{mw}}$. In contrast, when $width < step$, repeated aggregation does not perform unnecessary aggregation for values not required. In the next section, we will give empirical results quantifying the performance tradeoff between the two strategies under various circumstances.

Nested group-bys with a single moving window in the outer inner clause can be handled in a straightforward way. The foregoing discussion essentially says how to extend the algorithm in Figure 8.

6.3 Multiple Moving Windows

A natural question to ask is whether the two strategies work when there are multiple moving windows either in the same block or in a nested relationship. Multiple moving windows in the same block give rise to "hyper-rectangular" windows. Essentially, the attributes with moving window specifications are orthogonal to each other. For the repeated-aggregation strategy, the formation of moving window groups essentially performs a "cartesian product" on the moving window groups from each such attribute. The resultant answer tree may be big, but both repeated aggregation and rolling over work just as before.

Finally, consider the situation when there is a moving window in both the outer and the inner blocks. The processing for both the repeated-aggregation and the rolling-over strategies is, modulo the nesting involved, similar to the previous discussion on multiple moving windows.

6.4 Combined with Having Clauses

The discussion so far on moving windows assumes there is no having clause. For moving windows with (nesting and) having clauses, repeated aggregation works with no changes. For rolling-over, as long as there is no holistic aggregation, no change is needed. If a holistic aggregation is involved, we only need to delay the processing of the gamma files. In sum, the algorithm follows the same principle of not processing an inner block until the having clauses in all the outer blocks have been processed. For lack of space, we omit the details here.

7 Experimental Evaluation

7.1 Experimental Setup

We implemented Algorithm N-GB in Java. For comparision, we picked Galax [7] (the single major complete reference implementation of XQuery), and Qizx [17] (one of the most efficient XQuery engines available). We used the well known synthetic dataset XMark (50-500 MB) and real data sets DBLP (250 MB and 400 MB), and Protein [16] (13 MB), chosen for its high heterogeneity. Experiments were run on 2GHz CPU, 1GB RAM machine. All the runtimes are trimmed averages of 10 runs. We consider three broad classes of queries for which we ran several tests. For Galax and Qizx, we had to simulate grouping via nesting. For Oracle, we used the corresponding group-by features of SQL.

7.2 Simple Nested Group-bys

Here we consider simple group-by queries with nesting only. We analyze the performance on varying parameters such as levels of nesting etc. and also compare with competing XML systems. We also did some intial probing to see how the "SQL approach" (using Oracle) compares with N-GB.

7.2.1 Comparision with Oracle

For comparing with Oracle, we shredded the XML data and loaded the relational database. We used XMark as a basis for this comparison. For shredding, we followed the approach of [20]. For lack of space, we suppress the graphs, but while for single block group-bys the performance was comparable, even with 1 level of nesting, Oracle was 2-3 times worse than N-GB. This factor went up to 12 with 2 levels of nesting. Since nested group-by is fairly fundamental to XML, this motivates the need for direct efficient algorithms for this purpose.

7.2.2 Comparision with Galax

As a sanity check, we compared N-GB wit Galax. As expected, Galax performance was quite poor, taking more than 1000 sec in many cases. E.g., for a typical 1-level nested group-by on XMark 100 MB data set, Galax took 5 min – some 30 times more than N-GB. We do not compare with Galax further.

7.2.3 Comparision with Qizx

We considered various parameters of interest for testing against Qizx.

Size and Number of Groups : We measured the performance when we vary the number and size of the groups in the answer. We designed two types of queries, one producing small number of large groups (Query Q1 and Q3) and other producing large number of small groups (Query Q2 and Q4). Figure 9(a) shows how runtime varies for Qizx and N-GB for various datasets and sizes. Note the cutoff of 1000 secs and logscale on the Y-axis. Clearly, N-GB outperforms Qizx (sometimes by more than two orders of magnitude) when there are a large number of groups in the answer. But, for queries producing small answers, Qizx performs excellently – one of the reasons why we chose it for comparision. Another important observation is that, unlike Qizx, N-GB is very stable w.r.t. group number and size. Moreover, on the very heterogenous yet small Protein dataset, Qizx performs very poorly on *both* the queries. N-GB has consistent efficient performance.

Fully Vs. Partially Specified Paths : We tested two types of queries which differ only in the fact that the α-path is fully specified in one (Q5 and Q7) and partially specified in other (Q6 and Q8). Figure 9(b) shows the results for different datasets. Surprisingly, whether paths are fully or

partially specified affects the performance of Qizx quite dramatically (up to an order of magnitude difference). On the other hand, the performance of N-GB is stable.

Increasing levels of nesting : To study the effect of levels of nesting, we designed simple nested queries where we increased the number of nesting levels from 0 (flat query with no nesting) to 3. Due to lack of space, we show the results only for Xmark dataset of size 100MB (Figure 9(c)). Interestingly, we observe that N-GB is stable even in this case: the number of levels hardly affects its performance. Qizx performance rapidly degrades (by more than two orders of magnitude) from the flat query as the levels increase to 3.

Scalability : From the above graphs, we can draw conclusions about scalability of N-GB. For example, both Figure 9(a) & (b) show results for XMark dataset (50 MB to 200 MB). Qizx doesn't run for XMark, size 500 MB and more on a 1GB RAM machine (insufficient heapspace). (N-GB completed in 45-50 sec on XMark 500 MB.) For N-GB, we observed the parsing time of course increases linearly, but the rest of the computation and I/O grow sub-linearly. On the other hand, the scalability of Qizx is is sensitive to path expressions (fully/partially specified) and the number/size of groups.

7.3 Nested Queries with Having Clause

Our objective was to measure the benefits of early pruning on nested queries with having clause. We consider two types of 1-level nested queries with having and an anti-monotonic constraint in the outer block. One has a non-holistic aggregate in the inner block (Q9 and Q11) and the other has a holistic aggregate in the inner block (Q10 and Q12). Figure 9(d) shows the results for each query with and without early pruning. Since the main impact of early pruning is on computation and not parsing, in this graph, we show only the total aggregation time and gamma file I/O time. As expected, there are substantial savings (200-300%) for early pruning in all the cases. Moreover, note that the savings are more for queries in which the inner-aggregate is holistic. This is expected as holistic aggregate computation involves gamma-file I/O as well as more intensive computation and early pruning avoids aggregate computation for those inner groups whose outer group has been pruned.

7.4 Moving Windows (MW)

For group-by queries involving MW clauses, we wanted to measure the gain of rolling-over over repeated-aggregation as a function of the ratio of window width to step size. Since the gain is only w.r.t. the moving window aggregation time, this is what we show in the graph (Figure 9(e)). The figure shows the percentage gain in the computation time of rolling-over over repeated-aggregation for DBLP 250 MB dataset. The query used was a flat group-by query with MW clause and we varied the Width to Step ratio. Clearly for ratio < 1, repeated-aggregation is better whereas for ratios > 1, rolling-over is more efficient. Also the percentage gain increases as the ratio increases. We also measured the effect of early pruning on the above two strategies. We used 1-level nested group-by queries with moving window aggregate in the inner block and having clause with an anti-monotonic constraint in the outer block. Figure 9(f) shows the variation in computation time for the four self-explanatory exhaustive cases for DBLP dataset of sizes 250 MB and 400 MB. Note that the gains with early pruning are greater for the repeated-aggregation strategy as against the gains for rolling-over. As already discussed in Section 6, the reason is that repeated aggregation involves updating counters for multiple groups for each β value as against a single group in case of rolling-over. On the other hand, early pruning prunes away many groups - which explains the reduced gains for rolling-over over repeated-aggregation in this case.

8 Conclusions

Using a rich framework for expressing sophisticated aggregate queries on XML data with grouping, nesting, having, and moving window aggregations, we developed an efficient disk-based algorithm for computing all such queries. Using a comprehensive set of experiments, we showed that our algorithm has stability, scalability, and efficiency, and is often orders of magnitude faster than existing approaches. Furthermore, our algorithm naturally supports several optimizations which improve its efficiency even further. In ongoing work, we are exploring these ideas for fast computation of cube on XML data. The complete set of our experimental data, queries, and results are available from `http://www.cs.ubc.ca/~chai2006/xmlGBExperiments`.

References

[1] N. Bansal et al. Deep Processing of Group-bys for XML Analytics. submitted to a technical journal. July 2006.

[2] K. Beyer et al. "Extending XQuery for Analytics," SIGMOD 2005, pp. 503–514.

[3] V. Borkar and M. Carey. Extending XQuery for Grouping, Duplicate Elimination, and Outer Joins. XML Conference and Expo., Nov. 2004.

[4] A. Deutsch et al. "The NEXT framework for logical XQuery optimization," VLDB 2004, pp. 168–179.

[5] L. Fegaras et al. Query processing of streamed XML data. CIKM 2002: 126-133.

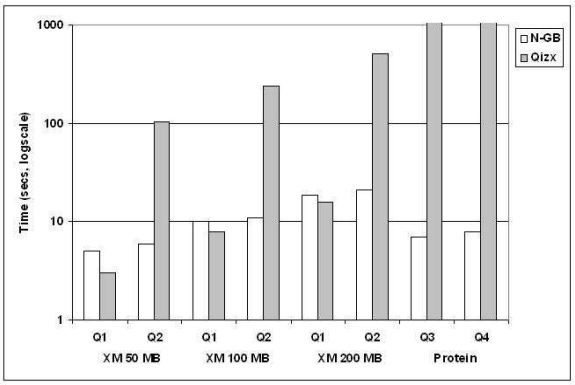
[6] T. Fiebig and G. Moerkotte. "Algebraic XML Construction and its Optimization in Natix," World Wide Web, 4(3), pp. 167–187, 2001.

[7] Galax. Galax XQuery engine. http://www.galaxquery.org.

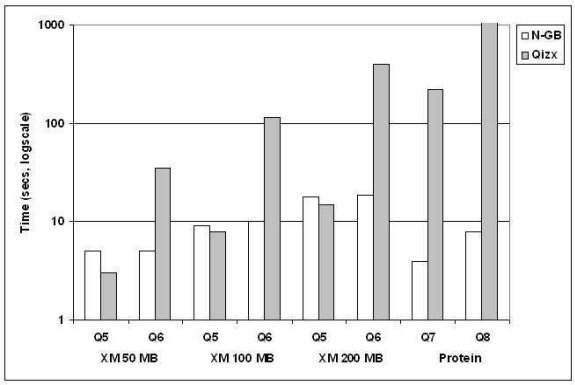
[8] J. Gray et al. Data Cube: A Relational Aggregation Operator Generalizing Group-by, Cross-Tab, and Sub Totals. Data Min. Knowl. Discov. 1(1): 29-53 (1997).

[9] N. May et al. Three Cases for Query Decorrelation in XQuery. 70-84.

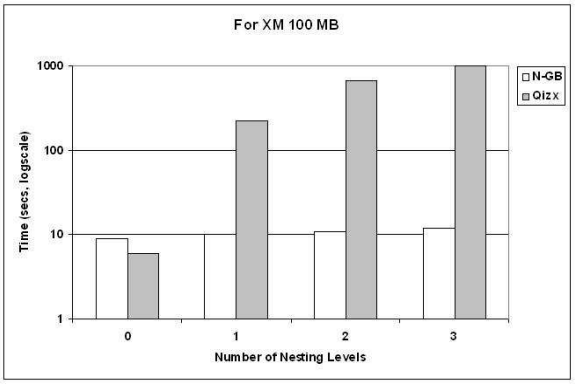
[10] A.O. Mendelzon et al. Data warehousing and OLAP: A research oriented bibliography. http://www.daniel-lemire.com/OLAP/index.html.



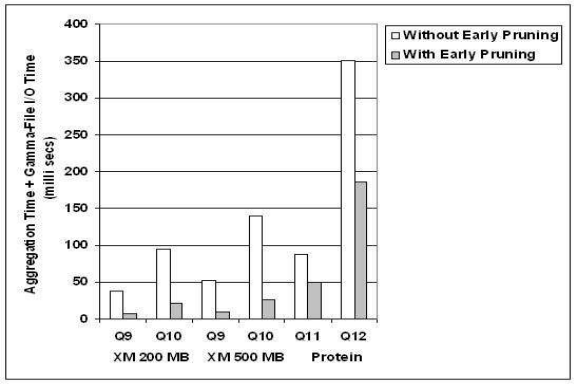
(a) Varying Size of Groups



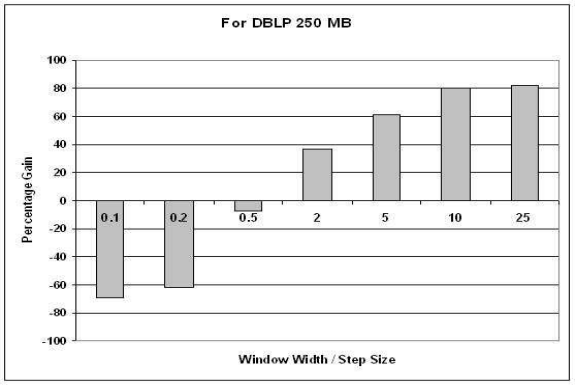
(b) Fully vs Partially Specified Paths



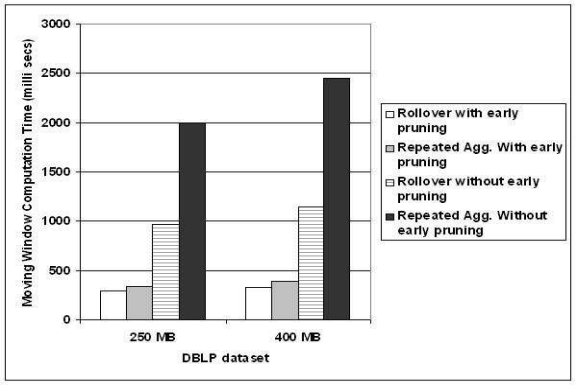
(c) Increasing levels of Nesting



(d) Early Pruning



(e) MW with varying ratios



(f) MW with having clause

Figure 9. Experimental Results.

[11] R. Ng et al. Exploratory Mining and Pruning Optimizations of Constrained Association Rules. SIGMOD 1998: 13-24.

[12] S. Paparizos et al., "Grouping in XML," EDBT 2002 Workshop, LNCS 2490, pp. 128–147.

[13] D. Pedersen et al. "Query Optimization for OLAP-XML Federations," ACM Workshop on Data Warehousing and OLAP 2002, pp. 57–64.

[14] J. Pei et al. Mining Frequent Item Sets with Convertible Constraints. ICDE 2001: 433-442.

[15] N. Polyzotis et al. "Approximate XML Query Answers," SIGMOD 2004, pp. 263-274.

[16] Georgetown Protein Information Resource. `http://pir.georgetown.edu/home.shtml`.

[17] Qizx/open. Qizx/open XQuery engine. `http://www.xfra.net/qizxopen`.

[18] R. Ramakrishnan et al. SRQL: Sorted Relational Query Language. SSDBM 1998: 84-95.

[19] M. Ramanath et al. "IMAX: The Big Picture of XML Dynamic Statistics," ICDE 2005.

[20] J. Shanmugasundaram et al. Relational Databases for Querying XML Documents: Limitations and Opportunities. VLDB 1999: 302-314.

[21] World Web Consortium (W3C) "XQuery 1.0: an XML Query Language," April 2005. http://www.w3.org/TR/xquery/ .