

# Efficient XQuery Evaluation of Grouping Conditions with Duplicate Removals

Norman May and Guido Moerkotte

University of Mannheim

B6, 29

68131 Mannheim, Germany

`norman|moer@db.informatik.uni-mannheim.de`

**Abstract.** Currently, grouping in XQuery must be expressed implicitly with nested FLWOR expressions. With XQuery 1.1, an explicit **group by** clause will be part of this query language. As users integrate this new construct into their applications, it becomes important to have efficient evaluation techniques available to process even complex grouping conditions. Among them, the removal of distinct values or distinct nodes in the partitions defined by the **group by** clause is not well-supported yet. The evaluation technique proposed in this paper is able to handle duplicate removal in the partitions efficiently. Experiments show the superiority of our solution compared to state-of-the-art query processing.

## 1 Motivation

XML gains importance as a storage format for business or scientific data. As more and more data is stored in this format, analytical query processing, i.e. XOLAP, becomes an important requirement. XQuery is the query language standardized for this purpose. While XQuery already includes a rich set of features, it lacks functions to support analytical query processing efficiently. Most importantly, grouping must be formulated implicitly with nested queries. While this challenge has already been addressed by techniques that unnest nested queries, end users and database implementors have identified the need for an explicit grouping construct that allows for efficient processing.

Consequently, a value-based grouping construct is part of the core requirements for XQuery 1.1, the next version of XQuery. Since the work on this version has just started, we will use the proposal for a **group by** construct by Beyer et. al. [1]. An efficient XQuery execution engine should include a powerful implementation of the grouping operator. With minor extensions of the relational grouping operators, it is possible to support several cases of grouping. We focus on a case that is neither well-supported for XQuery nor for SQL: We investigate efficient evaluation techniques for **group by** where duplicates are removed on different attributes of tuples that are in the same partition.

## 1.1 Motivating Example

Consider the following example query on the XMark document instance depicted in Fig. 1(a). It counts for every open auction the total number of bidders, the number of distinct bidders, the maximum increase, and the number of different increases (We abbreviate some element or attribute names as follows: personref – pr, @person – @p, increase – i).

```

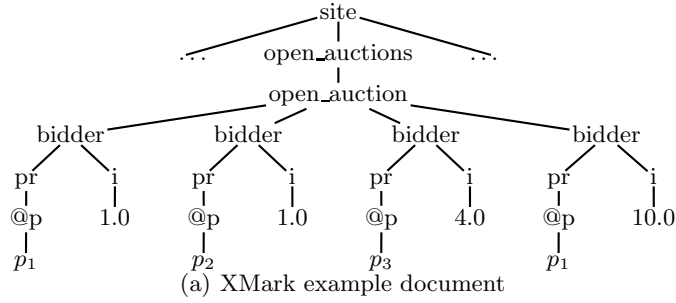
for $auction in $doc/site/open_auctions/open_auction,
    $bidder in $auction/bidder
let $person := $bidder/personref/@person,
    $increase := $bidder/increase
group by $auction into $a using fn:deep-equal
    nest $person into $p,
        $increase into $i
    let $pd := distinct-values($p),
        $pi := distinct-values($i)
return
    <status>
      { $a/seller }
      <bid-count> { count($p) } </bid-count>
      <distinct-bidders> { count($pd) } </distinct-bidders>
      <max-increase> { max($i) } </max-increase>
      <distinct-steps> { count($id) } </distinct-steps>
    </status>

```

In the example query, the keyword **group by** is directly followed by the *grouping expression* (a reference to variable \$auction), the keyword **into**, and the *grouping variable* (\$a). The function mentioned after the keyword **using** is used to partition the input tuples into groups. The *nesting expression* after the keyword **nest** is applied to every tuple that is assigned to some group. The result of this computation is appended to the current group referenced by the *nesting variable*. In the example query, we use the optional **let** clause to remove duplicates from the sequences bound to the nesting variables. The result of the **group by** is a sequence of tuples that contains bindings for every grouping variable and every nesting variable. Notice that sequence order is not meaningful in the context of this operator because there is no immediate relationship between input tuples and output tuples any more. We denote with *aggregation variable* all variables in the scope of the **return** clause that are the argument of an aggregate function, i.e. \$p, \$i, \$pd, \$pi in our example query.

Fig. 1(b) shows the tuples that serve as input to the **group by**. We trace how the group defined by a single open auction,  $a_1$ , is processed. In Fig. 1(c), we have identified groups – in this example, there is only a single group. To compute the result of the aggregate functions in the **return** clause of this query, we need to remove duplicate values from every sequence bound to the nesting variables. We have highlighted them in the two sequences.

Most systems try to avoid copying and, hence, would filter duplicates by discarding complete input tuples. In general, this only works when duplicates are removed from at most one aggregation variable. In our example, however,



\$auction	\$person	\$increase
$a_1$	$p_1$	1.0
$a_1$	$p_2$	1.0
$a_1$	$p_3$	4.0
$a_1$	$p_1$	10.0

(b) Input of **group by**

\$a	\$p	\$i
$a_1$	< $p_1$ ,	< <b>1.0</b> ,
	$p_2$ ,	<b>1.0</b> ,
	$p_3$ ,	4.0,
	$p_1$ >	10.0 >

(c) Groups and duplicates detected

\$a	\$p	\$pd	\$i	\$id
$a_1$	< $p_1$ ,	< $p_1$ ,	< 1.0,	< 1.0,
	$p_2$ ,	$p_2$ ,	1.0,	4.0,
	$p_3$ ,	$p_3$ >	4.0,	10.0 >
	$p_1$ >		10.0 >	

(d) Duplicates removed

**Fig. 1.** A complex grouping query

the second and the fourth tuple contain duplicates. However, we have to keep the second tuple because bidder  $p_2$  is a unique value. We also have to keep the fourth tuple because it contains a unique value for the increase. Clearly, every aggregation variable with duplicate removal needs to be processed separately. The result after duplicate removal in the **let** clause is shown in Fig. 1(d). It is the input to the aggregate functions in the **return** clause.

## 1.2 State-of-the-Art Processing

As we are not aware of any publically available system that supports the **group by** operator in XQuery, we have looked at similar queries in SQL.<sup>1</sup> We could distill two basic strategies to process this type of queries.

**Sort-Based Strategy:** Replicate the input for every aggregation variable that requires duplicate elimination. Sort the sequences to aggregate and use a sort-based implementation to evaluate the aggregate function with and without duplicate removal in one pass over the data.

<sup>1</sup> In SQL, the keyword **distinct** may occur only inside one single aggregate function. Nevertheless, actual database systems support the occurrence of this keyword in several aggregate functions over distinct attributes.

**Hash-Based Strategy:** Perform one scan and compute the aggregate function for all attributes without duplicate removal, and perform another scan for every aggregation variable with duplicate removal. This alternative may also use hash-based implementations for grouping and duplicate removal.

In our experiments we show that these two strategies do not scale well with an increasing number of duplicate removals. For every expression that demands a duplicate removal, either strategy introduces a new scan over the input data. Consequently, query performance suffers as the number of such expressions grows.

### 1.3 Our Contributions

The contribution of our work is to avoid the repeated scan of the input data mentioned above. We propose to process a single group at a time. In many cases, the whole group will fit into main memory and, thus, expensive I/O is avoided. Moreover, we can handle groups of arbitrary size – even larger than available main memory. In our evaluation strategy, available main-memory is a tunable parameter. Thus, we can trade increased memory consumption for faster processing in main memory. Unlike other proposals for grouping of XML data, our processing strategy fits well into current database architectures. In particular, we need to extend the query execution engine only with two new algebraic operators. We have implemented all three alternative processing strategies in Natix, our native XML database system [7]. Our experiments show that our approach performs favourably compared to the state of the art.

*Outline of the Paper.* Next, in Sec. 2 we discuss related work. The core of this paper is Sec. 3, in which we discuss the three alternative strategies to process grouping with duplicate removal in the nesting expression. In experiments presented in Sec. 4, we compare the performance of these plan alternatives. In Sec. 5, we summarize the results of this paper.

## 2 Related Work

The recently published W3C recommendation of XQuery does not contain an explicit **group by** construct [2]. Consequently, grouping must be expressed with nested queries, and optimizers need to detect that grouping is formulated implicitly. Proposals to detect *implicit grouping* by unnesting nested query blocks in XQuery include [13, 15, 16].

As motivated in [1], it can be quite cumbersome to formulate grouping queries correctly. Moreover, if the query optimizer cannot detect implicit grouping in a nested query, evaluating the nested query usually results in poor performance. Hence, both users and database implementors seem to agree that XQuery should include an explicit **group by** syntax and, thus, value-based grouping as we discuss it in this paper is a core requirement for the next version of XQuery [6]. Proposals for a syntax for grouping in XQuery have appeared in [1, 3, 12]. In this paper, we use the syntax and semantics presented in [1].

Recently, implementations for **group by** and the cube operator for analytical XML query processing were developed. The grouping operator proposed in [9] computes arbitrary aggregates over a single XML document by merging XML nodes belonging to the same group. In this proposal, retrieval of the XML data, grouping, and aggregation are tightly integrated into a single processing strategy.

Another extension to XQuery, a cube operator for XML, is presented in [17]. The paper investigates computational and semantic challenges associated with the aggregation of XML. Since implementations of the cube operator can benefit from efficient algorithms of the grouping operator, our work is also relevant when the arguments of aggregate functions are subject to duplicate removal.

Both proposals above do not explicitly address the problem of duplicate elimination. In fact, we are not aware of any proposal to handle duplicate elimination in grouping operations. The two processing strategies, which we use to benchmark our implementation, are derived from the explain utilities of two commercial database products.

Our solution is closely related to the XML result construction operators presented by Fiebig et al. [8]. Based on this framework, we process one group at a time. This allows us to optimize the processing of every aggregate individually and, thus, improve the performance of query evaluation.

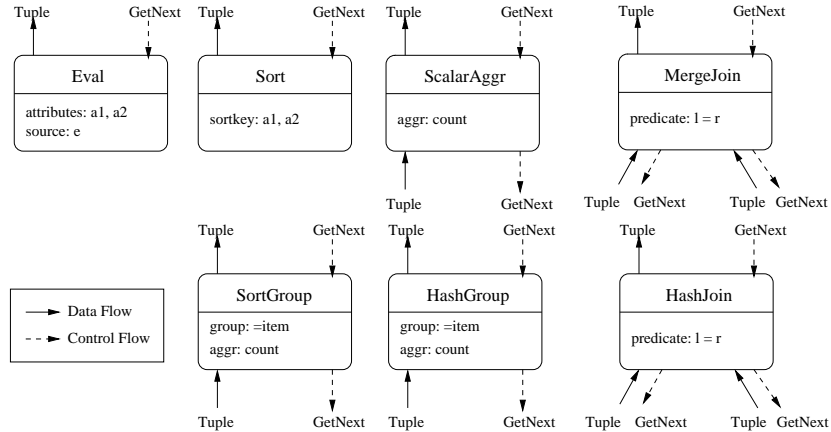
Standard implementation techniques for the grouping operator are discussed by Graefe [10]. This survey also discusses data flow and control flow beyond the standard iterator model, as we use them in this paper. We plan to extend our distinct processing strategy to the binary grouping operator [14] and to window-based aggregation. Of course, efficient implementations for the grouping operator need to be complemented with optimizations, as they were presented in [5, 11, 18]. These optimizations are still valid for our implementation.

### 3 The Grouping Algorithms

In this section, we develop the grouping algorithm that can handle duplicate elimination in arguments of aggregate functions efficiently. First, we introduce some notation necessary to understand the plan alternatives we present in this section. This notation is borrowed from [8]. Then, we discuss three alternatives to evaluate grouping with duplicate removal.

#### 3.1 Notation

The algebraic operators in the query execution engine of Natix are implemented as iterators [10]. They consume and produce sequences of tuples. Every tuple contains a set of attributes which are either bound to base types such as strings, numbers, or node references, or again contain ordered sequences of tuples. Thus, our operators conform to the basic iterator interface **open**, **getNext**, and **close**; details can be found in [7]. The operators we use in this paper are shown in Fig. 2. In the upper part of an operator, we give its name, whereas the lower part contains information about the subscript, e.g. the sort key of the **Sort** operator. They include:



**Fig. 2.** Operator Notation

**Eval** This operator is not an actual operator. In this paper, it denotes XML data retrieval needed as input to the grouping operators. It evaluates a complex expression,  $e$ , in its subscript and binds the attributes mentioned. Executing expression  $e$  might include index access or navigation over an XML document. Since efficient XML data retrieval is not the focus of this paper, we use this operator as an abbreviation.

**Sort** sorts its input according to the sort key mentioned in its subscript. Our implementation uses external sorting with replacement selection.

**ScalarAggr** returns a single tuple with a set of attributes, each of which is bound to the result of an aggregate function, e.g.  $fn:min$ ,  $fn:sum$ ,  $fn:count$ , or even SQL/XML functions such as `xmlagg`.

**MergeJoin** implements a 1:N sort-merge join. Of course, its arguments must be sorted on the attributes mentioned in the join predicate. In this paper, we do not need the more general N:M sort-merge join.

**HashJoin** implements a nested-loop join where blocks of the left producer are loaded into a main-memory hash table and matched with tuples of the right producer.

**SortGroup** groups the input assuming that the sequence of tuples of the producer is sorted by the grouping attributes.

**HashGroup** employs a main-memory hash table to perform the grouping operation.

All these operator implementations are well-known from the literature [10]. Notice that the two grouping operators can also be used to remove duplicates. When we want to remove duplicates, we denote this by `DupleElim`.

In this iterator-based implementation, control flows from a consumer of tuples to its producer, and data flows into the opposite direction. As depicted in Fig. 2, we denote the former by dashed arrows and the latter by solid arrows. As our solution involves control flow beyond the direct argument relationship, we explicitly present them in our plans.

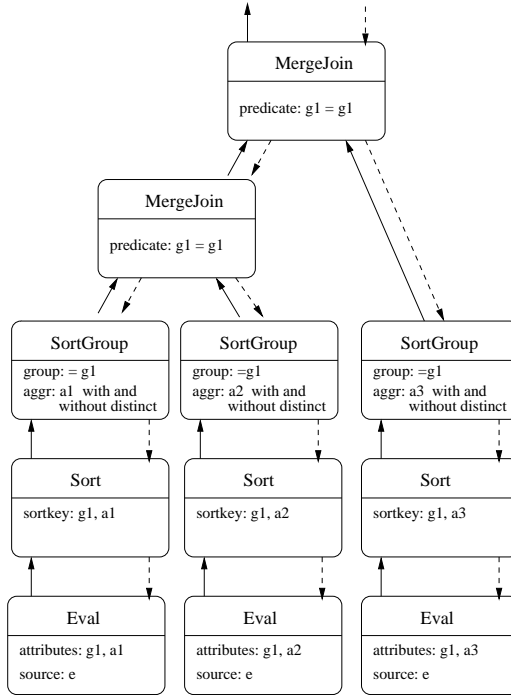


Fig. 3. Plan for sort-based strategy

### 3.2 Sort-Based Evaluation Strategy

The first evaluation strategy for grouping we have observed in commercial systems is shown in Fig. 3. It performs repeated scans of the input data for every attribute that contains a duplicate removal. Additional aggregate functions are piggy-backed on the branches in the plan. All grouping operators require their input to be sorted on the grouping attributes. In addition, the input must be sorted on the attributes mentioned in aggregate functions with duplicate removal. As a consequence, the sort-based grouping operator can compute aggregate functions that do not contain duplicate removals and all aggregates that contain duplicate removal on the same attribute in one scan. At the end, all partial plans are combined by a sort-merge join because this join implementation exploits the order available on the grouping attributes.

Evidently, this strategy performs the scan of the input data and, if needed, the sort operation repeatedly for every distinct attribute mentioned in an aggregate function with duplicate removal. It is our goal to share this repeated evaluation and, thereby, improve the performance of the plan.

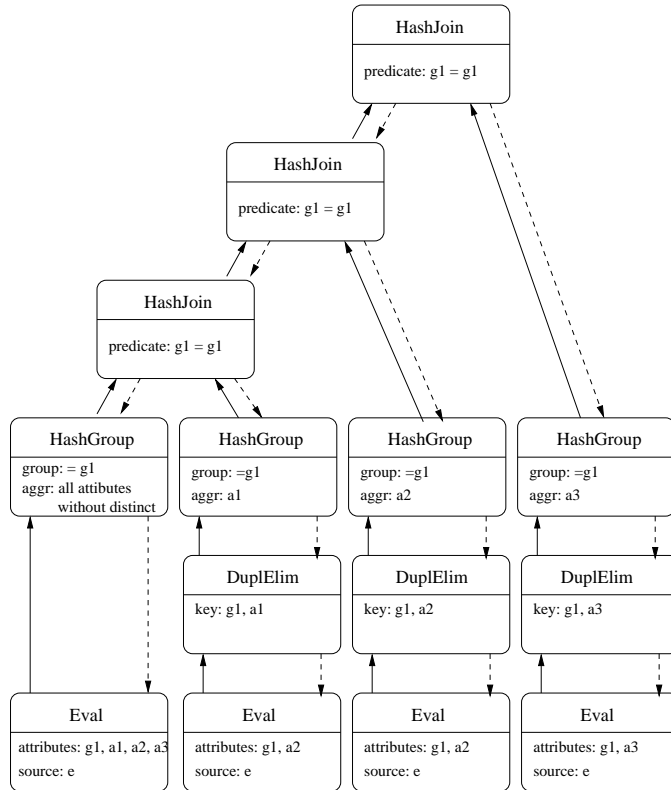


Fig. 4. Plan for hash-based strategy

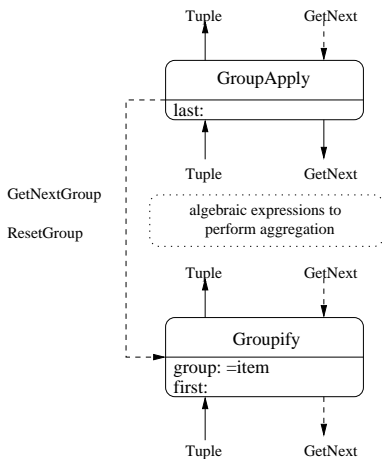
### 3.3 Hash-Based Evaluation Strategy

The second strategy we have found in commercial systems employs one partial plan to compute the aggregation function for all attributes where no duplicates are removed. For every aggregate function that contains a duplicate removal on some attribute, one partial plan is executed. All these plans are combined by joins to compute the final result. This strategy, depicted in Fig. 4, leaves the query optimizer the freedom to choose between sort-based and hash-based implementations for every partial plan. Thus, in this extreme case where we use sorting for every branch, we arrive at the sort-based strategy but with one additional partial plan.

For this reason, we investigate the case where all operations are performed by hash-based operators. The potential advantage of hash-based operators is that they avoid sorting. Notice, however, that this strategy shares the inefficiency of repeated scans (or evaluation) of the input.

### 3.4 Groupify and GroupApply

It is our goal to avoid the repeated evaluation of the argument expression of the **group by** operator. The key idea of our approach is to separate detecting group boundaries from processing groups. For every group, evaluating the complex grouping conditions follows three steps. These steps are governed by two new operators, **Groupify** and **GroupApply**, and involve non-standard control and data flow between them. This is shown in Fig. 5.



**Fig. 5.** Groupify and GroupApply

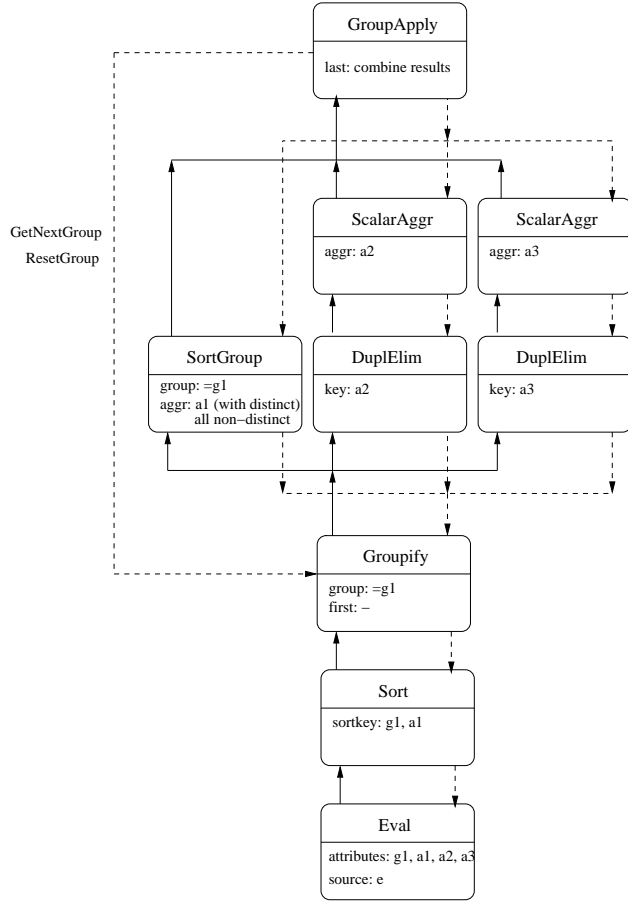
partial plans. For example, we can apply any of the strategies discussed in the previous two subsections. Another advantage is that processing of a group focuses on a fragment of the whole data and, thus, can benefit from the locality of data access.

Finally, we need the **GroupApply** operator to combine the results of the partial plans. After a group was detected and materialized, this operator evaluates all partial plans in a round-robin fashion. Since every partial plan is evaluated on the same sequence of tuples, the **GroupApply** operator signals the **Groupify** operator to scan the current group again from the beginning. Therefore, the **GroupApply** operator calls the function **Groupify::resetGroup**. The **GroupApply** operator combines the results of these plans into the overall result of the current group. When we have evaluated every partial plan, we can process the next group. Again, this requires communication between the **GroupApply** and **Groupify** operator. More precisely, the **GroupApply** operator uses the function **Groupify::getNextGroup** to notify that the evaluation of all argument plans is complete. Then, the **Groupify** operator discards the current group and processes the next one.

Fig. 5 shows the communication between the **Groupify** operator and the **GroupApply** operator as dashed arrows. The control flow bypasses the plans that perform the aggregation. The data flow extends the iterator model because

First, the **Groupify** operator detects group boundaries. Computing the aggregate functions for each group requires multiple passes over the tuples of this group. Therefore, the **Groupify** operator materializes the tuples of a group. Notice that a group might not completely fit into physical main memory. When we have finished processing a group, the operator discards the tuples of this group and continues with the next one. In our implementation, we expect the input to be sorted on the grouping attributes. But a hash-based implementation is also possible.

Second, aggregation and duplicate removal is done by plans that consume the tuples of a group. One advantage of our approach is that we can optimize these



**Fig. 6.** Plan based on Groupify and GroupApply

several algebraic operators consume the tuples produced by the Groupify operator. Notice that we neither have to modify any iterator-based operator nor the general query processing architecture of our system to support this data flow. All necessary extensions are restricted to the Groupify and GroupApply operator.

### 3.5 Evaluation Strategy-Based on Groupify and GroupApply

Now, we discuss how we use the operators introduced in the previous subsection to improve the query execution time for queries containing grouping and duplicate removal on different attributes of tuples that are in the same partition.

From the plan structure shown in Fig. 6, it is evident that we need to evaluate the argument expression only once. The result of this expression is sorted on the grouping attributes, and the Groupify operator exploits the order to detect group boundaries.

The leftmost partial plan computes the aggregate functions without duplicate removal, using a sort-based grouping operator. This partial plan can benefit from a minor sort on the attributes  $a_1$ , on which it needs to perform a duplicate removal. Notice that this plan also carries the grouping attributes needed in the final result. The remaining partial plans remove the duplicate values for a single attribute and compute the aggregate for this attribute. An advantage of this fine-grained approach is that every partial plan can be optimized by a cost-based optimizer. In particular, a sort-based or hash-based evaluation strategy can be chosen for every of these partial plans. The `GroupApply` operator combines the results of the partial plans to the final result tuple of the grouping operation.

## 4 Experiments

We have implemented all algorithms in Natix, our native XML database system [7]. Natix was compiled with GCC 4.1.2 and optimization level O3. All queries were executed on a Linux system with Kernel 2.6.18, an Intel Pentium 4 CPU 2.40GHz, 1 GB RAM, and IBM 18.3 GB Ultra 160 SCSI hard disk drive with 4 MB buffer. All queries were run with cold buffer cache of 8 MB size. We report the average execution time of three runs of every query.

### 4.1 Dataset and Queries

To investigate the performance of the three execution strategies presented in Sec. 3, we use a synthetic dataset. This setup allows us to carefully investigate the impact of different parameters both of the query and the data.

Every execution strategy retrieves the input to the **group by** clause from a materialized XML view that contains exactly the tuples needed to evaluate the query. This is reasonable because we want to isolate the effect of the different implementations for grouping. We have examined three XML views  $X_1$ ,  $X_2$ , and  $X_3$ . Their cardinality and raw data size is summarized in Fig. 7.

XML view	cardinality	raw size
$X_1$	$2^{16} = 65k$	1.3 MB
$X_2$	$2^{20} = 1M$	21 MB
$X_3$	$2^{23} = 8M$	168 MB

**Fig. 7.** Dataset

In Fig. 8, we show the query pattern we used to benchmark the query performance of the different evaluation strategies. The first **for** clause retrieves the grouping attribute and the remaining **for** clauses the  $k$  attributes to aggregate. Choosing different tag names `tag-g` and `tag-i` ( $i = 1 \dots k$ ) in these clauses allows us to modify the number of groups or the number of distinct values for the attributes to aggregate. Thus, we can control the number of groups in the result and the cost and effect of the duplicate removal in the nesting expressions. In the **group by** clause, we have  $k$  nesting variables. Every nesting variable stores one sequence of values to aggregate in this group. For each such sequence, the **let** clause after that computes the sequence of values with duplicates removed. In the **return** clause, we apply the aggregate function `fn:sum` to the nesting sequences computed this way.

```

for $g in $doc//tag_g,
    $a1 in $g//tag_a1,
    ...
    $ak in $g//tag_ak
where P
group by $g into $gg using eq
    nest $a1 into $a1,
    ...
    $ak into $an,
let $a1d := distinct-values($a1),
    ...
    $and := distinct-values($ak)
return
<result>
  { $g }
  <a1> { sum($a1) } </a1>
  <a1d> { sum($a1d) } </a1d>
  ...
  <ak> { sum($ak) } </ak>
  <akd> { sum($akd) } </akd>
</result>

```

**Fig. 8.** Query Pattern

functions on five distinct attributes with duplicate removal. We consider both only four (large) groups and 1024 (smaller) groups. Notice the logarithmic scale of both axes. Clearly, the sort-based approach scales worst among the three alternatives. From Fig. 9(a) it is evident that the hash-based plan can exploit that only four groups exist. Both grouping and the subsequent joins need to manage only few result tuples which leads to an overall performance advantage compared to the plan using **Groupify** and **GroupApply**. In Fig. 9(b), on the other hand, these two operators scale best.

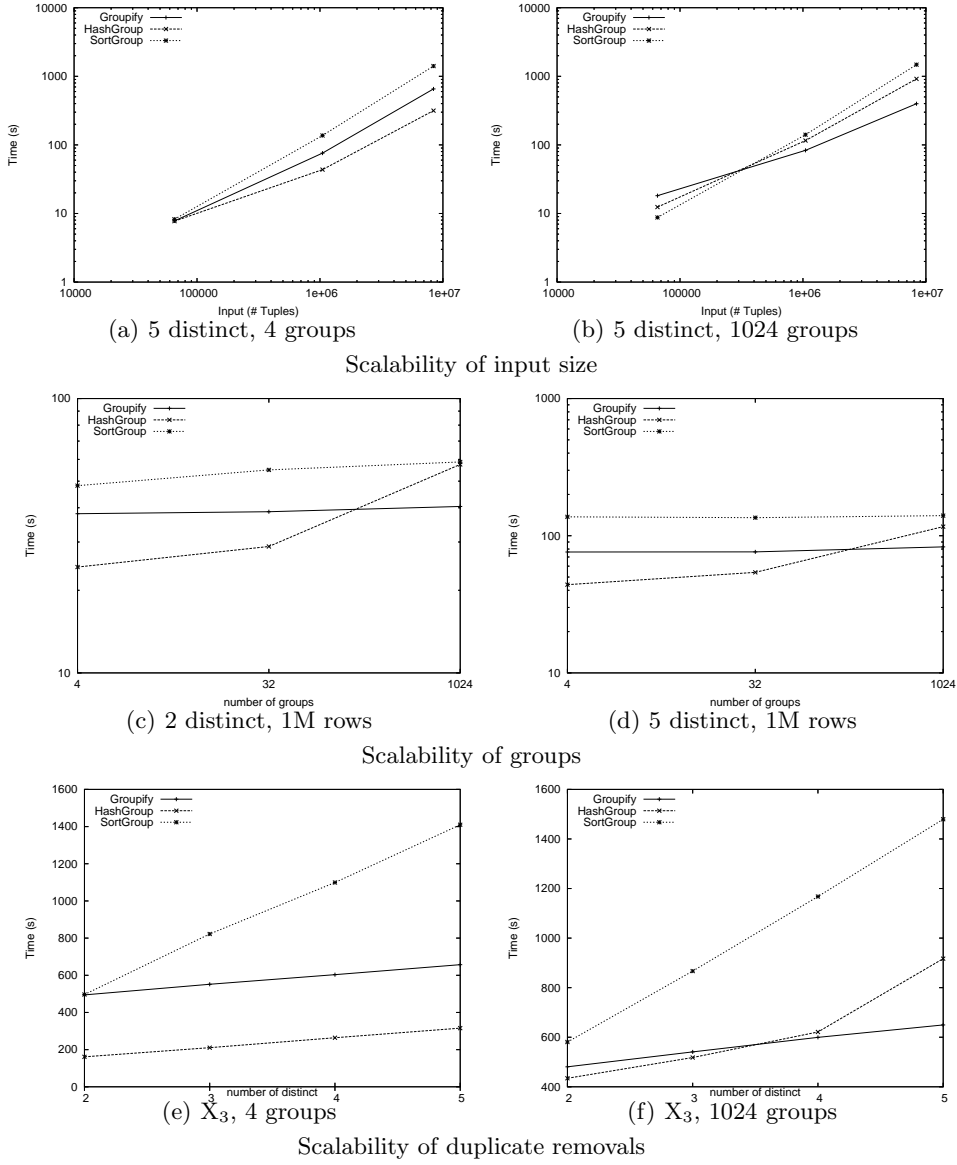
*Scalability of Groups.* In Figs. 9(c) and 9(d), we compare the effect of the number of groups to process. The more groups we have in the input data, the smaller they become. In both plots, we report the elapsed execution times on the XML view  $X_2$  with two or five attributes with duplicate removal. The experiments clearly show that both the sort-based strategy and **Groupify/GroupApply** are almost insensitive with respect to the number of groups. The cost of these plans is dominated by the cost for sorting, and this cost component does not change much with the size of the groups. After that, detecting group boundaries is very cheap. Independent of the number of groups, the strategy based on **GroupApply** and **GroupApply** outperforms the sort-based strategy. The performance of the hash-based plan, on the other hand, is very sensitive to the number of distinct groups. Consequently, the plan quality of this strategy strongly depends on good cardinality estimates for the number of groups. Unfortunately, estimating the number of groups is an inherently difficult task [4]. **Groupify** and **GroupApply** become faster than the hash-based method when there are more than approx. 50

Every tuple consists of 5 integer attributes: one with  $s$  unique values, the others with 4, 32, 1024, or 65536 distinct values. Every attribute corresponds to one tag name or attribute selected from the document. For space reasons, we can only report the most interesting results.

## 4.2 Experimental Results

Fig. 9 summarizes the results of our experiments. We investigate how each algorithm scales with respect to the input size, the number of distinct groups, and the number of nesting expressions with duplicate removal.

*Scalability of Input Size.* Figs. 9(a) and 9(b) show how the three algorithms behave with increasing input size when the query contains aggregate



**Fig. 9.** Experimental Results

groups. In further experiments, we observed similar results for larger and smaller input sizes.

*Scalability of Duplicate Removals.* In our final experiment, shown in Figs. 9(e) and 9(f), we scale the number of attributes on which we remove duplicates. The plots show the query execution times on XML view  $X_3$  for four and 1024 groups.

	Sort-based	Hash-based	Groupify/GroupApply
Scans of input	$\max( a_d , 1)$	$1 +  a_d $	1
Temp+Scan	$\max( a_d , 1)$	$ a_d $	can be tuned
Main Memory	Sort buffers	Hash buffers	can be tuned
Combining result		Join(s)	Copy values

where  $a_d$  = attributes that occur with **distinct**

**Fig. 10.** Summary of the plan alternatives

Clearly, the sort-based strategy performs worst among the three alternatives. For every new attribute with duplicate removal, it has to scan and sort the whole input once more. As both operations demand I/O operations, the performance suffers.

Both the hash-based plan and **Groupify/GroupApply** scale better than the sort-based plan. Again, the hash-based algorithm is the fastest for few groups. However, for a larger number of groups, **Groupify/GroupApply** outperform the other alternatives.

*Number of Duplicates to Remove.* In our experiments, the query execution times did not change significantly when we increased the number of duplicates to be removed before aggregating them. Hence, we do not present any experimental results that show the effect of duplicate elimination.

## 5 Conclusion

We have investigated three different strategies to evaluate grouping when duplicates are removed in several nesting expressions. Based on the algorithms underlying each strategy, we can derive the I/O operations needed for each strategy (see Fig. 10). Both the sort-based and the hash-based alternative scan the base data once for every aggregation variable which requires a duplicate elimination. The combination of **Groupify** and **GroupApply**, on the other hand, scans the base data only once. Consequently, **Groupify/GroupApply** scales better than the other two strategies with an increasing number of duplicate removals on different aggregation variables. Since this strategy keeps a single group in main memory while this group is processed by several partial plans, it avoids expensive I/O operations. If the group is too large to fit in main memory, parts of the group can be spooled to disk. Hence, I/O operations can be traded for main-memory usage. Overall, this leads to more local data access patterns.

Our novel evaluation strategy only requires two new algebraic operators in a query engine and, thus, it fits well into the standard architecture of database systems. Finally, we remark that neither alternative can be done in a fully pipelined fashion. But the group-wise processing of **Groupify/GroupApply** returns first results faster. The sort-based strategy demands several more sort operations and thus, is slower. The hash-based method must process all groups before the first

result tuple is returned. Based on these observations, we plan to develop a cost model for this processing strategy.

Clearly, evaluation techniques discussed in this paper are not restricted to grouping in XQuery. It may also be useful for analytical SQL queries in a data warehouse environment. Currently, however, the SQL standard allows `DISTINCT` to be applied only to a single aggregation expression.

*Acknowledgements* We would like to thank Simone Seeger for her comments on the manuscript.

## References

1. K. Beyer, D. Chamberlin, L. Colby, F. Özcan, H. Pirahesh, and Yu Xu. Extending XQuery for analytics. In *Proc. of the ACM SIGMOD*, 2005.
2. S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. *XQuery 1.0: An XML Query Language*. W3C, 2007.
3. V. Borkar and M. Carey. Extending XQuery for grouping, duplicate elimination, and outer joins. In *XML 2004*, 2004.
4. M. Charikar, S. Chaudhuri, R. Motwani, and V. Narasayya. Towards estimation error guarantees for distinct values. In *Proc. of the ACM PODS*, 2000.
5. S. Chaudhuri and K. Shim. Including group-by in query optimization. In *Proc. VLDB*, 1994.
6. D. Engovatov. *XML Query 1.1 Requirements*. W3C Working Draft, 2007.
7. T. Fiebig, S. Helmer, C-C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, and T. Westmann. Anatomy of a native XML base management system. *VLDB Journal*, 11(4), 2002.
8. T. Fiebig and G. Moerkotte. Algebraic XML construction and its optimization in Natix. *WWW Journal*, 4(3), 2001.
9. C. Gokhale, N. Gupta, P. Kumar, L. Lakshmanan, R. Ng, and B. A. Prakash. Complex group-by queries for XML. In *Proc. ICDE*, 2007.
10. G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2), 1993.
11. A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. In *Proc. VLDB*, 1995.
12. M. Kay. Positional grouping in XQuery. In *<XIME-P/>*, 2006.
13. N. May, S. Helmer, and G. Moerkotte. Strategies for query unnesting in XML databases. *ACM TODS*, 31(3), 2006.
14. N. May and G. Moerkotte. Main memory implementations for binary grouping. In *XSym*, 2005.
15. S. Pappas, S. Al-Khalifa, H.V. Jagadish, L. Lakshmanan, A. Nierman, D. Srivastava, and Y. Wu. Grouping in XML. In *EDBT workshops*, 2002.
16. C. Re, J. Siméon, and M. F. Fernández. A complete and efficient algebraic compiler for XQuery. In *Proc. ICDE*, 2006.
17. N. Wiwatwattana, H.V. Jagadish, L. Lakshmanan, and D. Srivastava. X<sup>3</sup>: A cube operator for XML OLAP. In *Proc. ICDE*, 2007.
18. W. P. Yan and P.-Å. Larson. Performing group-by before join. In *Proc. ICDE*, 1994.