Main Memory Implementations for Binary Grouping

Norman May and Guido Moerkotte

University of Mannheim
B6, 29
68131 Mannheim, Germany
{norman,moer}@pi3.informatik.uni-mannheim.de

Abstract. An increasing number of applications depend on efficient storage and analysis features for XML data. Hence, query optimization and efficient evaluation techniques for the emerging XQuery standard become more and more important. Many XQuery queries require nested expressions. Unnesting them often introduces binary grouping.
We introduce several algorithms implementing binary grouping and analyze their time and space complexity. Experiments demonstrate their performance.

1 Motivation

Optimization and efficient evaluation of queries over XML data becomes more and more important because an increasing number of applications work with XML data. In XQuery – the emerging standard query language for XML – queries including restructuring or aggregation often require nested queries. For example, the following query returns for each of the fifty richest persons of the world the number of countries with smaller gross domestic product (GDP) than the person's total capital.

```
for $p in document("richest-fifty.xml")//person
return
 <result>
  <person> { $p/name } </person>
  <count-richer> {
    count(for $c in document("countries.xml")//country
       where  $p/capital gt $c/gdp
       return $c) }
  </count-richer>
 </result>
```

This query combines data of two different documents and performs grouping and aggregation over the XML data. Note that each country can contribute to the count of multiple persons, and that a non-equality predicate is used to relate items from both documents.

S. Bressan et al. (Eds.): XSym 2005, LNCS 3671, pp. 162–176, 2005.

Direct nested evaluation of this query is highly inefficient because for each person the nested FLWR expression is evaluated, demanding a scan of the countries document. Fortunately, the query can be unnested introducing binary grouping [17]. Moreover, optimizers can then apply algebraic equivalences to further improve performance. However, efficient implementations for binary grouping are not available yet. If they were, the optimizer could choose among them, ensuring an efficient query evaluation. We fill this gap and present several main-memory algorithms for implementing binary grouping. Further, we analyze their time and space complexity. The different algorithms will require different conditions to hold. Enumerating them then enables the query optimizer to select the most efficient implementation of binary grouping for a given situation. Experiments demonstrate that performance can be improved by orders of magnitude. Due to space constraints, we restrict ourselves to the formulation of algorithms working on sets of tuples. However, an extension to bags or sequences is not difficult (see [18]). Let us stress that binary grouping is useful not only in the context of XQuery. It has also been successfully applied to unnest nested OQL-queries [4, 20] and to evaluate complex OLAP queries [2].

The paper is structured as follows. Section 2 presents the definition of binary grouping and surveys properties of predicates and aggregate functions. They form the basis for the selection of an efficient implementation for the binary grouping operator. The main contribution of this paper – Section 3 – introduces several algorithms for binary grouping and analyzes their time and space complexity. Exemplary performance results are given in Section 4. More detailed experimental data is presented in [18]. Before concluding this paper, Section 5 reviews related work.

2 Preliminaries

2.1 The Algebra

We will only present the operators needed for our exposition. For an extensive treatment of our algebra we refer to [4]. Our framework is extendible to sequences as required in XQuery (cf. [17] for this algebra and related work).

The algebra works on sets of unordered tuples. Each tuple contains a set of variable bindings representing the attributes of the tuple. Single tuples are constructed by using the standard $[\cdot]$ brackets. The concatenation of tuples and functions is denoted by $\circ$. The set of attributes defined for an expression e is defined as $\mathcal{A}(e)$. The set of free variables of an expression e is defined as $\mathcal{F}(e)$.

For an expression e_1 possibly containing free variables and a tuple t, $e_1(t)$ denotes the result of evaluating e_1 where bindings of free variables are taken from variable bindings provided by t – this requires $\mathcal{F}(e_1) \subseteq \mathcal{A}(t)$. Note that this can also be used for function application. We denote NULL values by $\_$.

The semantics of the binary grouping operator is defined by the map operator (χ) and the selection (σ). If their input is the empty set ($\emptyset$), their output is also empty.

Let us briefly recall **selection** with predicate p defined as $\sigma_p(e) := \{x | x \in e, p(x)\}$ and **map** defined as $\chi_{a:e_2}(e_1) := \{y \circ [a : e_2(y)] | y \in e_1\}$. The latter extends a given input tuple $y \in e_1$ by a new attribute a whose value is computed by evaluating $e_2(y)$.

Definition 1. *We define the* **binary grouping** *operator as:*

$$e_1 \Gamma_{g;A_1\theta A_2;f} e_2 := \chi_{g:f(\sigma_{A_1\theta A_2}(e_2))}(e_1)$$

In this definition we call e_1 **grouping input** *and* e_2 **aggregation input**.

Note that the result of the binary grouping operator is empty if and only if the grouping input evaluates to an empty set. When the aggregation input is empty we assume that $f(\emptyset)$ is well-defined, and $f(\emptyset)$ is returned as the result. In many cases f will be an aggregation function such as `sum`. We refer to [18] for examples of applying these operators.

2.2 Properties of Predicates

To find the most efficient implementation for binary grouping, we take a closer look at the properties of predicates. Therefore, we distinguish, for example, *symmetric, irreflexive* predicates ($\neq$) from *antisymmetric, transitive* predicates ($<, \leq, >, \geq$).

2.3 Properties of Aggregate Functions

Aggregate functions can be *decomposable* and *reversible* [3]. These properties help us to find the most efficient implementation for binary grouping. To make the paper self-contained, we recall the definitions of these properties.

Let $\mathcal{N}$ be the codomain of a *scalar aggregate function* $f : X \rightarrow \mathcal{N}$ over some set X of tuples. In the definitions below, we will make use of (sub-) sets X, Y, and Z, with $X = Y \mathbin{\dot\cup} Z$ and $Y \cap Z = \emptyset$.

Definition 2. *We say* $f : X \rightarrow \mathcal{N}$ *is* **decomposable** *if there exist functions*

$$\begin{aligned} \alpha &: X \rightarrow \mathcal{N}' \\ \beta &: \mathcal{N}', \mathcal{N}' \rightarrow \mathcal{N}' \\ \gamma &: \mathcal{N}' \rightarrow \mathcal{N} \end{aligned}$$

with $f(X) = \gamma(\beta(\alpha(Y), \alpha(Z)))$

Decomposable aggregate functions allow us to aggregate on subsets of the whole data and combine the results of these computations to the aggregate over the whole data. Obviously, the common aggregate functions are decomposable.

Definition 3. *A decomposable scalar function* $f : X \rightarrow \mathcal{N}$ *is called* **reversible** *if for* β *there exists a function* $\beta^{-1} : \mathcal{N}', \mathcal{N}' \rightarrow \mathcal{N}'$ *with*

$$f(Z) = \gamma(\beta^{-1}(\alpha(X), \alpha(Y)))$$

for all X*,* Y*, and* Z *with* $X = Y \mathbin{\dot\cup} Z$ *and* $Y \cap Z = \emptyset$.

α

A_1	s	c
1	5	2
2	9	2
3	0	0
-	14	4

(a) after matching

β^{-1}

A_2	s	c
1	9	2
2	5	2
3	14	4

γ

A_2	a
1	4.5
2	2.5
3	3.5

(b) $\neq$-table

β

A_2	a	
1	14	4
2	9	2
3	0	0

γ

A_2	a
1	3.5
2	4.5
3	-

(c) $\leq$-table

Fig. 1. Example of the reversible aggregate function `avg`

Reversible scalar aggregates allow us to compute the value of an aggregate function over some subset by computing the aggregate function over some superset. Using this result, we can use the inverse function β^{-1} to compute the desired value for the subset. As examples `sum`, `count`, and `avg` are reversible, `min` and `max` are not.

For function `avg`, we define $\alpha(X) = [s : sum(X), c : |X|]$ computing the sum and cardinality of each group, $\beta([s : s_1, c : c_1], [s : s_2, c : c_2]) = [s : s_1 + s_2, c : c_1 + c_2]$, $\beta^{-1}([s : s_1, c : c_1], [s : s_2, c : c_2]) = [s : s_1 - s_2, c : c_1 - c_2]$ combining the sums and counts of two groups, and $\gamma([s : s_1, c : c_1]) = [a : s_1/c_1]$ yielding the average for each group.

The θ-table proposed in [3] exploits the properties of decomposable and reversible aggregate functions. Conceptually, the θ-table is an array with an entry for each group that stores data collected during aggregation. First, partial aggregation for some subset of the matching data is done. Then the results of the first step are combined to the final result for each group. The first step avoids duplicate work and is the source of improved efficency, while the second step benefits from the properties of the predicate and the aggregation function.

To make this more concrete, let us assume that after matching the grouping input and aggregation input the θ-table contains the data shown in Figure 1(a). In case of the $\neq$-table, aggregation is done with data matched with $=$ instead of $\neq$. In addition, the values for sum and count over the whole data set are collected in an auxiliary entry shown in the last row of the table (c.f. Fig. 1(b)). This auxiliary entry is used to obtain the sum and count values of each group using function β^{-1}. The final result is computed using function γ. For the first row in Figure 1(b) we have $\beta^{-1}([14, 4], [5, 2]) = [14 - 5, 4 - 2] = [9, 2]$ and $\gamma([9, 2]) = 4.5$.

With a $\leq$-table aggregation is only done on the closest matching group. The final result of each group is computed in a walk backwards through the table, incrementally combining the aggregated values of each group using function β. Applying function γ to each group yields the final result for each group. For the second row in Figure 1(c), we have $\beta([0, 0], [9, 2]) = [9, 2]$ and $\gamma([9, 2]) = 4.5$.

3 Algorithms

3.1 Notation

The following notation will be used in the complexity formulas to describe the time and space complexity of the various algorithms:

$$\begin{aligned} f &:= \textit{duplication factor} \\ g &:= \text{storage space per group} \\ \alpha &:= \textit{load factor} \text{ of the hash table} \\ l &:= \Theta(1+\alpha) \\ n &:= max(|e_1|, |e_2|) \end{aligned}$$

The *duplication factor* as defined in [1] is the ratio of the number of tuples before duplicate elimination to the number of tuples after duplicate elimination. Note that α, the load factor of the hash table, changes while values are inserted into the hash table. We will ignore this fact and use the load factor as an upper bound after all values have been inserted into the hash table. Therefore, all complexity formulas will represent upper bounds. For brevity reasons, we denote $l = \Theta(1+\alpha)$ as the time for a lookup in the hash table [5], and n as the maximum cardinality of both inputs.

In the exposition of each alternative algorithm we will follow the same basic structure: First, we state the assumptions on the predicate and the aggregate function as introduced in Section 2. Then, we present the algorithm in pseudo code and deduce the time and space complexity from the code. Finally, we explain implementation details. All operators are implemented as iterators [9] consisting of an **open** function for initialization, a **next** function which returns one result tuple of the operator for each call, and a **close** function that does some deinitialization. The implementations in our experiments are set-based. The pseudo code uses the following notations:

$p(x,y)$ – returns the result of evaluating the predicate $A_1\theta A_2$, where $A_1 \in \mathcal{A}(e_1)$, $A_2 \in \mathcal{A}(e_2)$, and θ a comparison as described in Section 2
T – a tuple of either input
G – a tuple representing a group
GT – an auxiliary grouping tuple
$\zeta_\alpha(G)$ – initializes a tuple G appropriately for α,
$\alpha(G,T)$ – returns the result of evaluating function α on a group G with tuple T from the aggregation input
$\beta(G_1,G_2)$, $\beta^{-1}(G_1,G_2)$ – return the result of evaluating β and β^{-1} on groups G_1 and G_2
$\gamma(G)$ – returns the result of γ on a group G

Figure 2 summarizes the algorithms we present in this paper. The left part of the table contains the algorithms with their time and space complexity derived from their code. The right part of the table surveys the assumptions for each algorithm. Thus, this table can be used as a guide to the most efficient implementation. The assumptions are related to the inputs e_1 and e_2, the predicate $A_1\theta A_2$, and the function f as used in Definition 1.

The last column indicates the ratio of improvement in execution time over the direct nested evaluation of the nested query. For simplicity, we restrict ourselves only to sorted input for both the grouping and aggregation input for an input size that all algorithms were capable to evaluate. We use the algorithm NestedSort

Algorithm			Assumptions				
Name	Time	Space	e_1	e_2	$A_1 \theta A_2$	f	Δ
NESTED	$\frac{l}{f}\lvert e_1\rvert\lvert e_2\rvert$	$\frac{g}{f}\lvert e_1\rvert$	-	-	-	-	0.95-1.2
NLBINGROUP	$\frac{l}{f}\lvert e_1\rvert\lvert e_2\rvert + (l + \frac{1}{f})\lvert e_1\rvert$	$\frac{g}{f}\lvert e_1\rvert$	-	-	-	-	0.65-0.75
HASHBINGROUP	$(l + \frac{1}{f})\lvert e_1\rvert +$ $O((\frac{\lvert e_1\rvert}{f} + \lvert e_2\rvert)\ lg \frac{\lvert e_1\rvert}{f})$	$\frac{(1+g)\lvert e_1\rvert}{f}$	-	-	¬SY, T	D	1300
TREEBINGROUP	$\frac{\lvert e_1\rvert}{f} + O((\lvert e_1\rvert + \lvert e_2\rvert) \lg \frac{\lvert e_1\rvert}{f})$	$\frac{g}{f}\lvert e_1\rvert$	-	-	¬SY, T	D	1300
EQBINGROUP	$l(\lvert e_1\rvert + \lvert e_2\rvert) + \frac{\lvert e_1\rvert}{f}$	$\frac{g}{f}\lvert e_1\rvert$	-	-	¬R, SY, ¬T	RE	1850
NESTEDSORT	$\frac{1}{f}\lvert e_1\rvert\lvert e_2\rvert$	$O(1)$	S	-	-	-	1.0
SORTBINGROUP	$\frac{1}{f}\lvert e_1\rvert\lvert e_2\rvert$	$O(1)$	S	-	-	-	1.1-1.2
LTSORTBINGROUP	$\lvert e_1\rvert + \lvert e_2\rvert$	$O(1)$	S	S	¬SY, T	-	2100

S	sorted	**R**	reflexive	**D**	decomposable
		SY	symmetric	**RE**	reversible
		T	transitive		

Fig. 2. Assumptions and complexity for the implementations of the binary grouping operator

as the basis defining it as $\Delta = 1.0$. For some algorithms ranges for Δ are given because they are applicable for different types of predicates. Values of $\Delta > 1.0$ indicate an improvement by a factor Δ. Obviously, algorithms with more assumptions evaluate up to three orders of magnitude faster than the nested-loops-based algorithms with fewer assumptions.

3.2 Direct Evaluation of Nested Query

Nested evaluation is most generally applicable and the basis of comparison for implementations of the binary grouping operator.

In general, nested queries are implemented by calling the nested query for each tuple given to the map operator. However, more efficient techniques were proposed to evaluate nested queries [11]. The general idea is to memoize the result of the nested query for each binding of the nested query's free variables. When the same combination of free variables is encountered, the result of the previous computation is returned. In general, a hash table would be employed for memoizing which demands linear space in the size of the grouping input. For sorted grouping input, only the last result needs to be stored resulting in constant space.

We have implemented both strategies, and we will refer to these strategies by NESTED and NESTEDSORT. Because of its simplicity we omit the pseudo code for the nested strategies and restrict ourselves to the analysis of the complexity (cf. Fig. 2). Both strategies expose quadratic time complexity because the nested query must be executed for each value combination of free variables generated by the outer query. In absence of duplicates, this is also true when memoization is used.

3.3 Nested-Loop-Implementation of Binary Grouping

NLBinGroup. There are no assumptions on the predicate, the aggregate function, or the sortedness of any input.

```
OPEN
open $e_1$
▷ detect groups
while $T \leftarrow$ next $e_1$
      do $G \leftarrow$ HT.LOOKUP($T$)
         if $G$ does not exist
            then $G \leftarrow$ HT.INSERT($T$)
                 ▷ initialize group
                 $\zeta_\alpha(G)$
close $e_1$
▷ match aggregation input to groups
open $e_2$
while $T \leftarrow$ next $e_2$
      do for each group $G$
                 in the HT
            do if $p(G, T)$
                  then $G \leftarrow \alpha(G, T)$
close $e_2$
htIter $\leftarrow$ HT.ITERATOR

NEXT
▷ next group in the hash table
if $G \leftarrow$ htIter.NEXT
   then return $\gamma(G)$
   else return _

CLOSE
HT.CLEANUP
```

(a) NLBinGroup

```
OPEN
open $e_1$
$\zeta_\alpha(GT)$            ▷ initialize group tuple
while $T \leftarrow$ next $e_1$
      do $G \leftarrow$ HT.LOOKUP($T$)
         if $G$ does not exist
            then $G \leftarrow$ HT.INSERT($T$)
                 ▷ initialize group
                 $\zeta_\alpha(G)$
close $e_1$

open $e_2$
while $T \leftarrow$ next $e_2$
      do $G \leftarrow$ HT.LOOKUP($T$)
         if $G$ exists
            then $G \leftarrow \alpha(G, T)$
         if predicate is $\neq$
            then $GT \leftarrow \alpha(GT, T)$
close $e_2$
htIter $\leftarrow$ HT.ITERATOR

NEXT
if $G \leftarrow$ htIter.NEXT
   then if predicate is $\neq$
           then $G \leftarrow \beta^{-1}(G, GT)$
        return $\gamma(G)$
   else return _

CLOSE
HT.CLEANUP
```

(b) EQBinGroup

Fig. 3. Pseudo code of NLBinGroup and EQBinGroup

We call the naive nested-loops-based implementation proposed in [2, 9] NLBinGroup. The pseudo code for this algorithm is shown in Figure 3(a). Most work is done in function OPEN. First, the grouping input is scanned, and all groups are detected and stored in a hash table ($l|e_1|$ time). Most of the following algorithms will follow this pattern. Next, the aggregation input is scanned once for each group in the hash table. The tuples from the aggregation input are matched with the tuple of the current group using the predicate. This matching phase is similar to a nested-loop join and requires $O(\frac{l}{f}|e_1||e_2|)$ time. When a match is found, function α is used for aggregation. After this matching phase a traversal through all groups in the hash table is done to execute function γ to finalize the groups ($\frac{|e_1|}{f}$ time). The complete complexity formulas can be found in Figure 2.

From the complexity equations we see that this algorithm introduces some overhead compared to the hash-based case of Nested because several passes through the hash table are needed. Hence, the time complexity is slightly higher then the direct nested evaluation. The following sections discuss more efficient algorithms for restricted cases.

3.4 Implementation of Binary Grouping with = or ≠-Predicate

EQBinGroup. If the predicate is not an equivalence relation, the aggregate function must be decomposable and reversible.

We generalize the $\neq$-Table defined in [3] for predicate $\neq$. Instead of an array, we use a hash table to store an arbitrary number of groups. When collision lists do not degrade, the asymptotic runtime will not change, however. The algorithm in Figure 3(b) extends NLBinGroup.

In function Open detecting all groups requires $l|e_1|$ time. In line 11 we do matching with equality for both kinds of predicates. But in line 15 all tuples are aggregated in a separate tuple GT using function α if the predicate is $\neq$. Alltogether matching requires $l|e_2|$ time.

When we return the result in a final sweep through the hash table ($\frac{|e_1|}{f}$ time) we have to apply the reverse function β^{-1} when the predicate is $\neq$ (cf. line 3 in Next). For that, we use the auxiliary grouping tuple GT and the group G matched with = and compute the aggregation result for $\neq$. For scalar aggregate functions, this computation can be done in constant time and space. For both types of predicates, groups are finalized using function γ.

Compared to the directly nested evaluation and hash-based grouping, the time complexity can be improved to linear time and linear space complexity (cf. Fig. 2).

Figure 4 shows how EQBinGroup implements the idea of the $\neq$-table introduced in Section 2. Figure 4(b) shows the content of the hash table after function open. For each detected group, the tuple for attribute a stores the value of attribute A_1, the `sum`, and the `count` of all matching tuples of the group. The additional tuple GT is added at the bottom of the table. Note that the group with value 3 did not find any match, but a properly initialized tuple for it exists in the hash table. Applying function β^{-1} to each group and GT and then function γ as described in Section 2 produces the final result (cf. Fig. 4(c)).

3.5 Implementation of Binary Grouping with ≤-Predicate

These algorithms are applicable if the predicate is antisymmetric, transitive, and the aggregate function is decomposable; no assumptions are made on the sortedness of any inputs.

In this paper we investigate a hash table and a balanced binary search tree to implement the $\leq$-table proposed in [3]. The advantage of this approach compared

R_1

A_1
1
2
3

R_2

A_2	B
1	2
1	3
2	4
2	5

(a) Input data

$(R_1)\Gamma_{a;A_1\neq A_2;avg(B)}(R_2)$

A_1	a
1	$\langle[1,5,2]\rangle$
2	$\langle[2,9,2]\rangle$
3	$\langle[3,0,0]\rangle$
GT	$\langle[\_,14,4]\rangle$

(b) After **open**

$(R_1)\Gamma_{a;A_1\neq A_2;avg(B)}(R_2)$

A_1	a
1	$\langle[1,4.5]\rangle$
2	$\langle[2,2.5]\rangle$
3	$\langle[3,3.5]\rangle$
GT	$\langle[\_,14,4]\rangle$

(c) Final result

Fig. 4. Example of the evaluation of EQBinGroup

```
Open
open e1
for T ← next e1
    do G ← HTl.Lookup(T)
        if G does not exist
            then G ← HT.Insert(T)
                  ▷ initialize group
                  ζ_α(G)
close e1
sort groups by matching
predicate of e1
open e2
for T ← next e2
    do G ← minimal group in
              ≤-Table ≥ T
        G ← α(G, T)
close e2
htIter ← ≤-Table.Iterator

Next
   ▷ next group in the ≤-table
if G ← htIter.Next
   then G ← β(G, successor(G))
        return γ(G)
   else return _

Close
HT.Cleanup
≤-Table.CleanUp
```

(a) HashBinGroup

```
Open
open e1
for T ← next e1
    do if _ == RB-Tree.Lookup(T)
          then G ← RB-Tree.Insert(T)
                ▷ initialize group G
                ζ_α(G)
close e1

open e2
while T ← next e2
    do G ← minimal group in
              RB-Tree ≥ T
       G ← α(G, T)
close e2
G ← RB-Tree.Maximum

Next
if G ≠ RB-Tree.Minimum
   then G ← β(G, RB-Tree.Succ(G))
        G' ← γ(G)
        G ← RB-Tree.Pred(G)
        return G'
   else return _

Close
RB-Tree.CleanUp
```

(b) TreeBinGroup

Fig. 5. Pseudo code of HashBinGroup and TreeBinGroup

to using an array is that no upper bound for the number of groups needs to be known. Since the assumptions are the same for both alternatives, we will only discuss implementation details.

R_1

A_1
1
2
3

R_2

A_2	B
1	2
1	3
2	4
2	5

(a) Input data

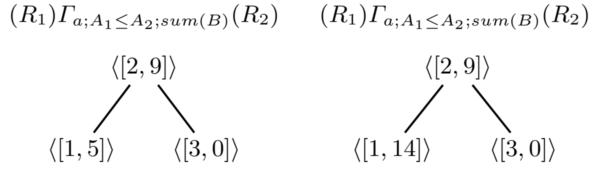
$(R_1)\Gamma_{a;A_1 \leq A_2;sum(B)}(R_2)$

A_1	a
1	$\langle[1,5]\rangle$
2	$\langle[2,9]\rangle$
3	$\langle[3,0]\rangle$

$(R_1)\Gamma_{a;A_1 \leq A_2;sum(B)}(R_2)$

A_1	a
1	$\langle[1,14]\rangle$
2	$\langle[2,9]\rangle$
3	$\langle[3,0]\rangle$

(b) HashBinGroup



(c) TreeBinGroup

Fig. 6. Example for the evaluation of HashBinGroup and TreeBinGroup

HashBinGroup. This algorithm, outlined in Figure 5(a), extends the NL-BinGroup operator. It is formulated in terms of predicate $<$.

First, all groups are identified using a hash table ($l|e_1|$ time). Before matching the tuples from the aggregation input, these groups are sorted according to the predicate ($O(\frac{|e_1|}{f} \lg \frac{|e_1|}{f})$ time). This can be done in a separate array in which the items in the hash table are referenced. In the matching phase binary search is employed to find the closest group that still matches with the predicate ($O(|e_2| \lg \frac{|e_1|}{f})$ time). Aggregation is done using function α. To compute the final result, one walk backwards through the array visits each group ($\frac{|e_1|}{f}$ time). First, the aggregated values of distinct groups are combined using function β. Then, function γ computes the final result of the group. One must be careful not to destroy the aggregated result of the previous group when applying function γ. The overall complexity can be found in Fig. 2.

TreeBinGroup. In an alternative implementation shown in Figure 5(b), we use a balanced search tree (e.g. a Red-Black-Tree) to identify all groups ($O(|e_1| \lg \frac{|e_1|}{f})$ time). The search tree structure implies the inclusion of groups. Thus, no sorting is needed after this step. Matching of tuples is done by a lookup in the search tree ($O(|e_2| \lg \frac{|e_1|}{f})$ time). When a group cannot be found, matching and aggregation is done on the last node in the tree that was visited. As with the previous algorithm, a backward traversal through the tree is done to aggregate the final result for each group using function γ ($\frac{|e_1|}{f}$ time). The resulting complexity is summarized in Fig. 2.

Comparison of the Implementations. Figure 6 resumes with the example in Section 2 to trace the evaluation of HashBinGroup and TreeBinGroup showing the state after **open**. Note that the groups must be sorted to find the closest matching group for aggregation with function α. This is achieved either by sorting the groups in the hash table or implicitly during insertion into the binary search tree. Each tuple stores the value of the grouping attribute and the aggregated result for the group. The result of the final walk backwards through the $\leq$-table computes the final result using function β and γ.

When we compare the complexity formulas we observe that sorting is dominant in HashBinGroup, and insertion is dominant in TreeBinGroup. Note that in both cases, we can remove duplicates during insertion. The hash-based implementation removes duplicates before sorting. In contrast, lookup of all items in e_1 in the balanced search tree demands $O(|e_1| \lg \frac{|e_1|}{f})$ time. This gives the hash-based implementation a potential advantage. On the other hand, the hash-based implementation does not degrade nicely when the collision lists on the hash table are not bounded by a constant any more. This can lead to linear search time in the collision lists ($l \in O(|e_1|)$, where l is the size of the collision list). Thus, the hash-based implementation depends on a good hash function.

3.6 Implementations of Binary Grouping on Sorted Input

When the grouping input or the aggregation input is sorted, we can improve the algorithm NLBinGroup.

SortBinGroup. First, we assume that only the grouping input is sorted.

Figure 7(a) presents the pseudo code for this algorithm. With sorted grouping input, groups can be detected efficiently because only subsequent tuples need to be compared (line 1 in Next). This can be done in constant space.

Matching the tuples of the aggregation input can be done with an algorithm similar to a 1:N sort-merge join, i.e. a sort-merge join algorithm that assumes that no duplicates occur on the left input. In the general case of an arbitrary predicate, the aggregation input needs to be scanned once for each group. This is done in $O(\frac{1}{f}|e_1||e_2|)$ time. It is also the reason for having no assumptions on the sortedness of the aggregation input.

Since the algorithm iterates through each group and matches all tuples from the aggregation input, groups does not have to be combined. Thus, the aggregation function need not be decomposable.

LTSortBinGroup. In addition to the assumptions of the previous algorithm, we now assume a antisymmetric and transitive predicate (e.g. $<$, or $\geq$). Both inputs need to be sorted. The direction of sorting depends on the predicate used. For example, for predicates $<$ and $\leq$ both inputs need to be sorted in descending order, for $>$ and $\geq$ in ascending order. No restrictions apply to the aggregation function.

These assumptions allow us to scan both inputs only once resulting in a time complexity of $|e_1| + |e_2|$. Each group resumes aggregation on the aggregated result of the previous group. For aggregation, we always use function α. The

```
OPEN
1  open $e_1$
2  open $e_2$

NEXT
1  if $G \leftarrow$ next group in $e_1$
2     then while $T \leftarrow$ next $e_2$
3              do if $p(G,T)$
4                   then $G \leftarrow \alpha(G,T)$
5           close $e_2$
6           open $e_2$
7           return $\gamma(G)$
8     else  return _

CLOSE
1  close $e_1$
2  close $e_2$
```

(a) SortBinGroup

```
OPEN
1  open $e_1$
2  open $e_2$
3  $\zeta_\alpha(GT)$      ▷ initialize group tuple

NEXT
1  if $G \leftarrow$ next group in $e_1$
2     then copy group attributes of
                 $G$ into $GT$
3          while ($T \leftarrow$ next $e_2$) $\wedge$
                 $p(GT, T)$
4             do $GT \leftarrow \alpha(GT,T)$
           ▷ keep aggregated result in $GT$
5          $G \leftarrow \gamma(GT)$
6          return $G$
7     else return _

CLOSE
1  close $e_1$
2  close $e_2$
```

(b) LTSortBinGroup

Fig. 7. Pseudo code of SortBinGroup and LTSortBinGroup

result of finalizing a group using function γ is stored in a separate tuple, so that the current value of aggregation is not destroyed (cf. line 5 in Next). The algorithm stated in Figure 7(b) is formulated in terms of $<$ or $\leq$ as predicates.

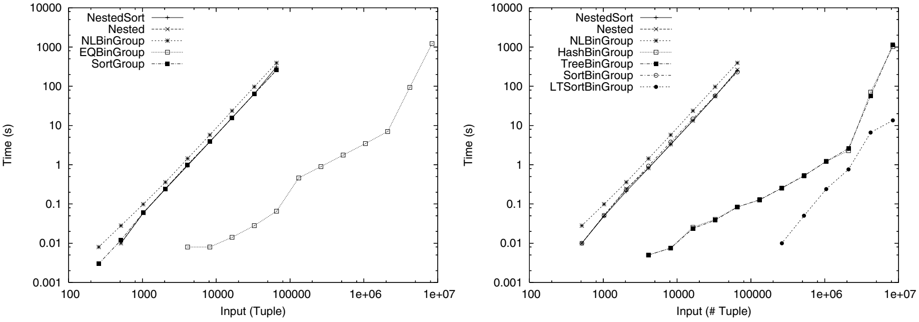
4 Experiments

We have implemented all algorithms in a prototype run-time system using GCC C++ version 3.3.4. All queries were executed on an Intel Pentium M with 1.4 GHz and 512MB RAM running Linux with 2.6.8 Kernel. In several experiments the performance of each algorithm was evaluated for different distributions. For space reasons we can only present a tiny fraction of the experimental data and refer to [18] for the details of the benchmark and the complete set of experimental results.

The cardinality of the input sequences e_1 and e_2 ranged between 128 and 8388608. The grouping input e_1 and the aggregation input e_2 were of equal size. The largest data set contained 63MB of data. The input for a query was loaded into main memory before executing the queries.

Figure 8 summarizes the most interesting results of our experiments. It shows the elapsed time for sorted input for both the grouping input and aggregation input for the predicate $\neq$ and $>$.

Figure 8(a) clearly shows that EQBinGroup is the most efficient algorithm for predicate $\neq$. It performs orders of magnitude faster than the nested-loops-based algorithms Nested, NestedSort, NLBinGroup and SortBinGroup



(a) Predicate $\neq$

(b) Predicate $>$

Fig. 8. Group input and aggregation input sorted

which are hard to distinguish in their query performance. However, since EQBinGroup loads all detected groups into a main memory data structure, its performance suffers when memory gets scarce. In our experiments, this happens for more than 2 million groups.

Figure 8(b) presents the experimental results for predicate $>$. The most efficient algorithm is LTSortBinGroup which is suited best for sorted input. When the input is not sorted, both HashBinGroup and TreeBinGroup are efficient algorithms with similar performance. When they run out of memory, both reveal the same weakness as EQBinGroup.

Among the nested-loop-based algorithms NLBinGroup is slowest. The inefficiency was caused by the iterator used for traversing the hash table. Only SortBinGroup exposes slightly improved efficiency compared to direct nested evaluation using memoization.

Summarizing, our experiments confirm the theoretical results from Section 3. We refer to [18] for more experimental results and a more detailed analysis.

5 Related Work

To the best of our knowledge, this paper is the first to investigate *efficient* implementations for binary grouping. Only one implementation corresponding to the NLBinGroup was presented so far [2].

However, previous work justifies the importance of binary grouping. Slightly different definitions of it can be found in [2, 3, 20]. Only [3] describes possible implementations. These papers enumerate use cases for binary grouping. In this paper we propose efficient implementations of binary grouping and evaluate their efficiency.

In addition, implementation techniques known for other operators apply for the binary grouping operator as well. The idea of merging the functionality of different algebraic operators to gain efficiency is well known. In [21] query patterns for OLAP queries are identified. One of these patterns – a sequence of grouping and equi-join – is similar to the implementation of the binary grouping operator. Sharing hash tables among algebraic operators was proposed in [12].

Our work also relates to work comparing sort-based and hash-based implementations of algebraic operators [7, 9, 10, 13, 14, 19]. However, they concentrate on implementations of equijoins. Non-Equality joins have been studied first in [8].

We presented main-memory implementations of the binary grouping operator. Implementation techniques that materialize data that does not fit into main memory can be applied to the binary grouping operator. We refer to [1, 6, 9, 15, 16] for such proposals.

6 Conclusion and Future Work

Binary grouping is a powerful operator to evaluate analytic queries [2] or to unnest nested queries [4, 17]. We have introduced, analyzed, and experimentally evaluated main memory implementations for binary grouping. Further, we have identified the conditions under which each algorithm is applicable.

The results show that query processing time can be improved by orders of magnitude, compared to nested evaluation of the query. Hence, binary grouping is a valuable building block for database systems that support grouping and aggregation efficiently.

For space reasons we refer to [18] for extensions of our algorithms to data models working on bags or sequences.

Acknowledgements

We would like to thank Carl-Christian Kanne and Simone Seeger for their comments on the manuscript.

References

1. D. Bitton and D. J. DeWitt. Duplicate record elimination in large data files. *ACM TODS*, 8(2):255–265, June 1983.
2. D. Chatziantoniou, M. Akinde, T. Johnson, and S. Kim. The MD-Join: An Operator for Complex OLAP. In *Proc. ICDE*, pages 524–533, 2001.
3. S. Cluet and G. Moerkotte. Efficient evaluation of aggregates on bulk types. *Proc. of 5-th DBPL*, 1995.
4. S. Cluet and G. Moerkotte. Nested queries in object bases. Technical Report RWTH-95-06, GemoReport64, RWTH Aachen/INRIA, 1995.
5. T. Corman, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, 2nd edition, 2001.
6. Jochen Van den Bercken, Martin Schneider, and Bernhard Seeger. Plug&join: An easy-to-use generic algorithm for efficiently processing equi and non-equi joins. In *EDBT '00*, pages 495–509, 2000.

7. D. J. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *Proc. of the ACM SIGMOD*, pages 1–8, June 1984.
8. D. J. DeWitt, J. F. Naughton, and D. A. Schneider. An evaluation of non-equijoin algorithms. In *Proc. VLDB*, pages 443–452, 1991.
9. G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.
10. G. Graefe. Sort-merge-join: An idea whose time has(h) passed? In *Proc. ICDE*, pages 406–417, 1994.
11. G. Graefe. Executing nested queries. In *BTW*, pages 58–77, 2003.
12. G. Graefe, R. Bunker, and S. Cooper. Hash joins and hash teams in Microsoft SQL server. In *Proc. VLDB*, pages 86–97, 1998.
13. G. Graefe, A. Linville, and L. D. Shapiro. Sort vs. hash revisited. *IEEE TKDE*, 6(6):934–944, December 1994.
14. L. M. Haas, M. J. Carey, M. Livny, and A. Shukla. Seeking the truth about *ad hoc* join costs. *VLDB Journal*, 6(3):241–256, May 1997.
15. S. Helmer, T. Neumann, and G. Moerkotte. Early grouping gets the skew. Technical Report TR-02-009, University of Mannheim, 2002.
16. S. Helmer, T. Neumann, and G. Moerkotte. A robust scheme for multilevel extendible hashing. *Proc. 18th ISCIS*, pages 218–225, 2003.
17. N. May, S. Helmer, and G. Moerkotte. Nested queries and quantifiers in an ordered context. *Proc. ICDE*, pages 239–250, 2004.
18. N. May, S. Helmer, and G. Moerkotte. Main memory implementations for binary grouping. Technical report, University of Mannheim, 2005. available at: `http://pi3.informatik.uni-mannheim.de/publikationen.html`.
19. D. E. Simmen, E. J. Shekita, and T. Malkemus. Fundamental techniques for order optimization. *SIGMOD Record*, 25(2):57–67, 1996.
20. H. J. Steenhagen, P. M. G. Apers, H. M. Blanken, and R. A. de By. From nested-loop to join queries in OODB. *Proc. VLDB*, pages 618–629, 1994.
21. T. Westmann and G. Moerkotte. Variations on grouping and aggregation. Technical report, University of Mannheim, 1999.