Reihe Informatik
007 / 2005

Main Memory Implementations for Binary Grouping

Norman May Guido Moerkotte

University of Mannheim
norman|moer@pi3.informatik.uni-mannheim.de

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.

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 [19]. 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 Section 5). 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 [21, 5] 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. Section 4 gives some performance results. Section 5 discusses how to extend our algorithms to bags or sequences and its implications. In Section **??** we outline how the binary grouping operator can be integrated into query optimizers. We summarize algebraic equivalences which ensure that query optimizers can find efficient query evaluation plans containing the binary grouping operator. Before concluding this paper, Section 6 reviews related work.

2 Preliminaries

Before we can describe the implementations of the binary grouping operator, we need to introduce some notation and definitions. Most of the material is well known.

Based on the properties we define shortly, we can choose the most efficient implementation of the binary grouping operator.

2.1 The Algebra

We will only present the operators needed for our exposition. For an extensive treatment of our algebra we refer to [5]. Our framework is extendible to sequences as required in XQuery (cf. [19] 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`. Figure 1 presents an example of the evaluation of the map operator and the binary grouping operator. We will reuse this example in the remainder of the paper. By *id* we denote the identity function.

R_1

A_1
1
2
3

R_2

A_2	B
1	2
1	3
2	4
2	5

$(R_1)\Gamma_{a;A_1=A_2;id}(R_2) \equiv \chi_{a:id(\sigma_{A_1=A_2}(R_2))}(R_1)$

A_1	a
1	$\langle[1,2],[1,3]\rangle$
2	$\langle[2,4],[2,5]\rangle$
3	$\langle[3,\_]\rangle$

$(R_1)\Gamma_{a;A_1=A_2;count}(R_2) \equiv \chi_{a:count(\sigma_{A_1=A_2}(R_2))}(R_1)$

A_1	a
1	2
2	2
3	0

Fig. 1. Example for map operator and binary grouping operator

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* [4]. 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.

The definitions are given in terms of sets, but extensions to bags and sequences are valid. Only the definition of disjoint set union and the empty set need to be adjusted to the bulk type as follows:

bulk type	$\emptyset$	$\dot\cup$
set	empty set	disjoint set union
bag	empty bag	bag union
sequence	empty sequence	append sequence

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

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

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

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* $f : X \to \mathcal{N}$ *is called* **reversible** *if for* β *there exists a function* $\beta^{-1} : \mathcal{N}', \mathcal{N}' \to \mathcal{N}'$ *with*

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

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

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.

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. 2. Example of the reversible aggregate function `avg`

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 [4] 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 efficiency, 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 2(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. 2(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 2(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 2(c), we have $\beta([0, 0], [9, 2]) = [9, 2]$ and $\gamma([9, 2]) = 4.5$.

3 Algorithms

Before we present the algorithms for the binary grouping operator, let us enumerate desirable characteristics of these implementations. First, these implementations must be correct. For each algorithm, we will clearly state assumptions that must be met to apply this algorithm. Second, these implementations should be flexible enough to support duplicate-preserving or order-preserving variations. Hence, they should support different bulk types for both the grouping input and the aggregation input, particularly the ones mentioned in Section 2. In Section 5 we describe how our implementations must be accommodated to these requirements.

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 [6], 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 [10] 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 3 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 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. The algorithms at the bottom of the table perform in linear time compared to quadratic time in the general case of nested evaluation. In general algorithms that require sorted input demand constant space while hash-based algorithms use linear space in the size of the grouping input.

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.

Assumptions There are no assumptions on the structure of the nested query.

Implementation 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 [12]. The general idea is to memoize the result of the nested query

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

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

Fig. 3. Assumptions and complexity for the implementations of the binary grouping operator

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. 3). 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

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

Implementation We call the naive nested-loops-based implementation proposed in [2, 10] NLBINGROUP. The pseudo code for this algorithm is shown in Figure 4(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 3.

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 $\neq$-Predicate

EQBinGroup

```
OPEN
 1  open e1
    ▷ detect groups
 2  while T ← next e1
 3        do G ← HT.LOOKUP(T)
 4           if G does not exist
 5              then G ← HT.INSERT(T)
                       ▷ initialize group
 6                     ζα(G)
 7  close e1
    ▷ match aggregation input to groups
 8  open e2
 9  while T ← next e2
10        do for each group G
                       in the HT
11              do if p(G, T)
12                    then G ← α(G, T)
13  close e2
14  htIter ← HT.ITERATOR

NEXT
    ▷ next group in the hash table
1   if G ← htIter.NEXT
2      then return γ(G)
3      else return _

CLOSE
1   HT.CLEANUP
```

(a) NLBINGROUP

```
OPEN
 1  open e1
 2  ζα(GT)            ▷ initialize group tuple
 3  while T ← next e1
 4        do G ← HT.LOOKUP(T)
 5           if G does not exist
 6              then G ← HT.INSERT(T)
                       ▷ initialize group
 7                     ζα(G)
 8  close e1

 9  open e2
10  while T ← next e2
11        do G ← HT.LOOKUP(T)
12           if G exists
13              then G ← α(G, T)
14           if predicate is ≠
15              then GT ← α(GT, T)
16  close e2
17  htIter ← HT.ITERATOR

NEXT
1   if G ← htIter.NEXT
2      then if predicate is ≠
3              then G ← β⁻¹(G, GT)
4           return γ(G)
5      else return _

CLOSE
1   HT.CLEANUP
```

(b) EQBINGROUP

Fig. 4. Pseudo code of NLBINGROUP and EQBINGROUP

Assumptions For the predicate, we assume a conjunction of symmetric predicates (e.g. $=$ or $\neq$). All clauses need to be of the same type. That is, all clauses either can be equality or inequality. In Section 5 we explain how to relax this constraint. If the predicate is not an equivalence relation, the aggregate function must be decomposable and reversible.

Implementation We generalize the $\neq$-Table defined in [4] 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 4(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. 3).

Figure 5 shows how EQBINGROUP implements the idea of the $\neq$-table introduced in Section 2. Figure 5(b) shows the content of the hash table after function OPEN. For each

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. 5. Example of the evaluation of EQBinGroup

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. 5(c)).

3.5 Implementation of Binary Grouping with $\leq$-Predicate

Assumptions 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.

However, the predicate may only contain exactly one clause. Similar to the EQBinGroup operator, we can relax this constraint. We will come back to this point in Section 5.

In [4] an implementation of θ-Tables based on an array containing the groups is proposed. The authors already mentioned the idea of using a balanced search tree instead of an array. However, they missed to give the precise conditions for applicability we listed above.

In this paper we investigate a hash table and a balanced binary search tree to implement the $\leq$-table proposed in [4]. The advantage of this approach compared 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.

HashBinGroup

Implementation This algorithm, outlined in Figure 6(a), extends the NLBinGroup 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. 3.

TreeBinGroup

Implementation In an alternative implementation shown in Figure 6(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

```
OPEN
 1  open e1
 2  for T ← next e1
 3        do G ← HT.LOOKUP(T)
 4           if G does not exist
 5             then G ← HT.INSERT(T)
                      ▷ initialize group
 6                   ζα(G)
 7  close e1
 8  SORT groups by matching
    predicate of e1
 9  open e2
10  for T ← next e2
11        do G ← minimal group in
                      ≤-Table ≥ T
12           G ← α(G, T)
13  close e2
14  htIter ← ≤-TABLE.ITERATOR

NEXT
    ▷ next group in the ≤-table
 1  if G ← htIter.NEXT
 2     then G ← β(G, successor(G))
 3          return γ(G)
 4     else return _

CLOSE
 1  HT.CLEANUP
 2  ≤-TABLE.CLEANUP
```

(a) HASHBINGROUP

```
OPEN
 1  open e1
 2  for T ← next e1
 3        do if _ == RB-TREE.LOOKUP(T)
 4             then G ← RB-TREE.INSERT(T)
                       ▷ initialize group G
 5                    ζα(G)
 6  close e1

 7  open e2
 8  while T ← next e2
 9        do G ← minimal group in
                      RB-TREE ≥ T
10           G ← α(G, T)
11  close e2
12  G ← RB-TREE.MAXIMUM

NEXT
 1  if G ≠ RB-TREE.MINIMUM
 2     then G ← β(G, RB-TREE.SUCC(G))
 3          G′ ← γ(G)
 4          G ← RB-TREE.PRED(G)
 5          return G′
 6     else return _

CLOSE
 1  RB-TREE.CLEANUP
```

(b) TREEBINGROUP

Fig. 6. Pseudo code of HASHBINGROUP and TREEBINGROUP

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. 3.

Comparison of the Implementations Figure 7 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.

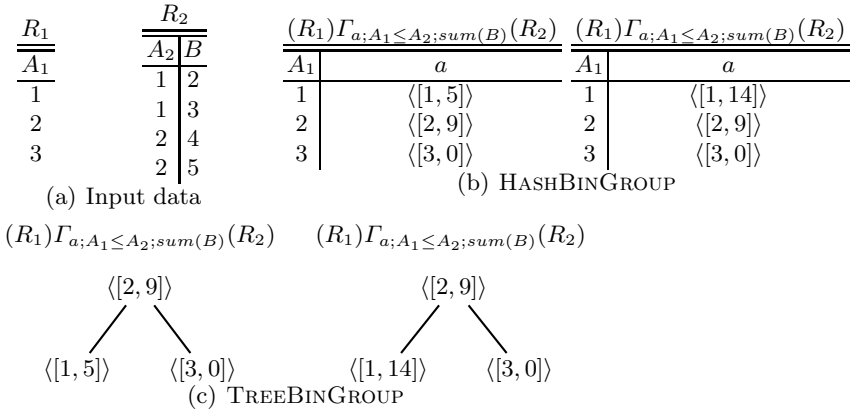


Fig. 7. Example for the evaluation of HashBinGroup and TreeBinGroup

```
Open
1  open e1
2  open e2

Next
1  if G ← next group in e1
2     then while T ← next e2
3                do if p(G,T)
4                      then G ← α(G,T)
5          close e2
6          open e2
7          return γ(G)
8     else return _

Close
1  close e1
2  close e2
```

(a) SortBinGroup

```
Open
1  open e1
2  open e2
3  ζα(GT)      ▷ initialize group tuple

Next
1  if G ← next group in e1
2     then copy group attributes of
                G into GT
3          while (T ← next e2) ∧
                p(GT, T)
4               do GT ← α(GT,T)
           ▷ keep aggregated result in GT
5          G ← γ(GT)
6          return G
7     else return _

Close
1  close e1
2  close e2
```

(b) LTSortBinGroup

Fig. 8. Pseudo code of SortBinGroup and LTSortBinGroup

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

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

Implementation Figure 8(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

Assumptions 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.

Implementation 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 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 8(b) is formulated in terms of $<$ or $\leq$ as predicates.

4 Experiments

To compare the different implementations of the binary grouping operator, we vary the parameters of the expression $e_1 \Gamma_{g;A_1\theta A_2;f} e_2$, i.e. function f, predicate θ and both input expressions e_1 and e_2.

From the universe of possible predicates, we investigated the predicates $<$ and $\neq$ to measure the possible improvements by using more efficient algorithms.

Since different aggregate functions will not fundamentally change the results of our experiments, we restricted ourselves to the aggregate function `sum`. This function is decomposable and reversible, which is a prerequisite for some of the efficient algorithms.

We implemented all algorithms in a prototype run-time system using GCC C++ version 3.3.4. For hash tables we used the STL hash_map. All queries were executed five times on an Intel Pentium M with 1.4 GHz and 512MB RAM running Linux with 2.6.8 Kernel. The average time of all executions was plotted. Query execution was aborted when the query demanded more than 10 minutes of elapsed time.

4.1 Data Set

The cardinality of the input sequences e_1 and e_2 ranged between 128 and 8388608. The largest data set contained 63MB of data. The input for a query was loaded into main memory before executing the queries. The time consumed for this was subtracted from all measured times. Each input tuple consumed approx. 30 bytes of main memory – including overhead by the memory manager of the operating system. Hence, the largest dataset consumed all physical memory for the input data. In each query both the grouping input e_1 and the aggregation input e_2 were of equal size. We used different types of distributions for the input to investigate their impact on the performance of the different algorithms. The input data consisted of random numbers.

uniform distribution which might contain duplicates

normal distribution for input of size x, we used $\mu = x/2$ and $\sigma = x/4$, and the values in the input were constrained to the range $[1, x]$

Zipf distribution with parameter z in the range [0.2, 2.0] in steps of 0.2. Again the range of values is constrained to the range $[1, x]$

sorted input in ascending order without any duplicates

4.2 Measurements

Since hash-based algorithms do not require sorted input on the grouping attributes, we start with discussing these algorithms. In our comparisons of the sort-based algorithms we use the hash-based algorithms to benchmark the sort-based algorithms again.

The left column of most figures in this section contains the elapsed time, and the right column contains the CPU time. Each row in a figure shows the execution times for one distribution of the aggregation input.

Hash-Based Algorithms In the first experiment we investigate the performance of the algorithm EQBinGroup compared to the naive evaluation using Nested and NLBinGroup. Figure 9 summarizes these results. All experiments used uniformly distributed grouping input.

The measurements consistently reveal the following characteristics: NLBinGroupNested is surprisingly slow – up to 10 times slower than nested evaluation using Nested. This is due to high overhead of the iterator implementation of the hash table we used which demanded almost 1/3 of CPU cycles. In addition, twice the number of L1 cache misses (read and write) were counted. When we loop over all groups in the hash table and execute the nested query for each group, the execution time decreases almost to the level of Nested. We think this is counter intuitive because traversing the main-memory hash table should be substantially faster than executing a possibly complex query expression.

As expected, the execution time of EQBinGroup is less by orders of magnitude. The drastic improvement is caused by the linear runtime developed in the complexity formula: both inputs need to be evaluated only one time.

Because NLBinGroup and EGBinGroup materialize the groups in main memory, their advantage in performance melts when they run out of physical main memory. Swapping memory pages consumes additional CPU time, and elapsed time increases steeper for more than 4 million groups.

Figure 10 compares the execution times with predicate $>$ of the algorithms Nested, NLBinGroup, HashBinGroup, and TreeBinGroup. Again, these algorithms were executed with uniformly distributed grouping input of different size. Both the grouping input and the aggregation input contained the same number of tuples.

The nested-loop-based algorithms perform orders of magnitude slower than both HashBinGroup and TreeBinGroup. For the same reasons as before, Nested performs better than NLBinGroup.

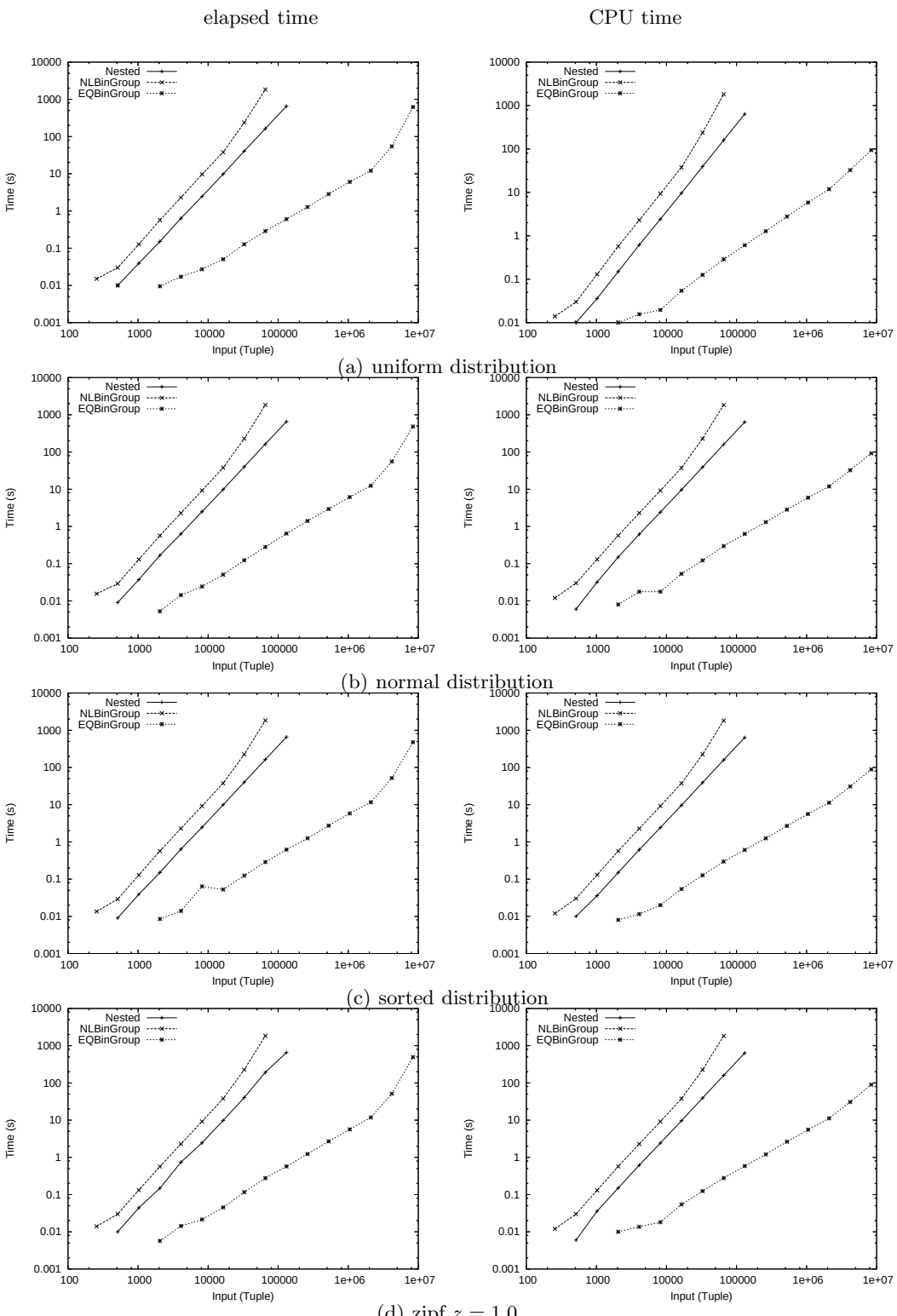
Both HashBinGroup and TreeBinGroup perform faster than the naive algorithms by orders of magnitude. Since they materialize the groups in the hash table, their performance degrades when main memory gets scarce — in our experiments this happens for input sizes larger than 4 million items. While the CPU time degrades only slightly, elapsed time is affected more because of swapping memory in and out.

In our experiments HashBinGroup executes faster than TreeBinGroup. Profiling revealed that insertion into the hash table and sorting together turns out to be cheaper than insertion into the red-black-tree. On the other hand, binary search during lookup is more expensive with the hash-based implementation. But in total, an advantage for HashBinGroup remains.

As already mentioned in Section 3, this advantage depends on the length of the collision lists, the quality of the hash function, and the data distribution.

To investigate the sensitivity of skewed input, we executed both algorithms with skewed aggregation input. Figure 11 gives the results of these experiments, showing that the narrower the distribution gets, the shorter the algorithms take for execution. The plots show that TreeBinGroup performs more robust for this range of different distributions. Only for larger input or for skewed data HashBinGroup performs better.

This observation suggests that lookup in the hash table in the presence of many duplicates is cheaper. Thus, preaggregation might be beneficial to shrink the input to the binary grouping operator.



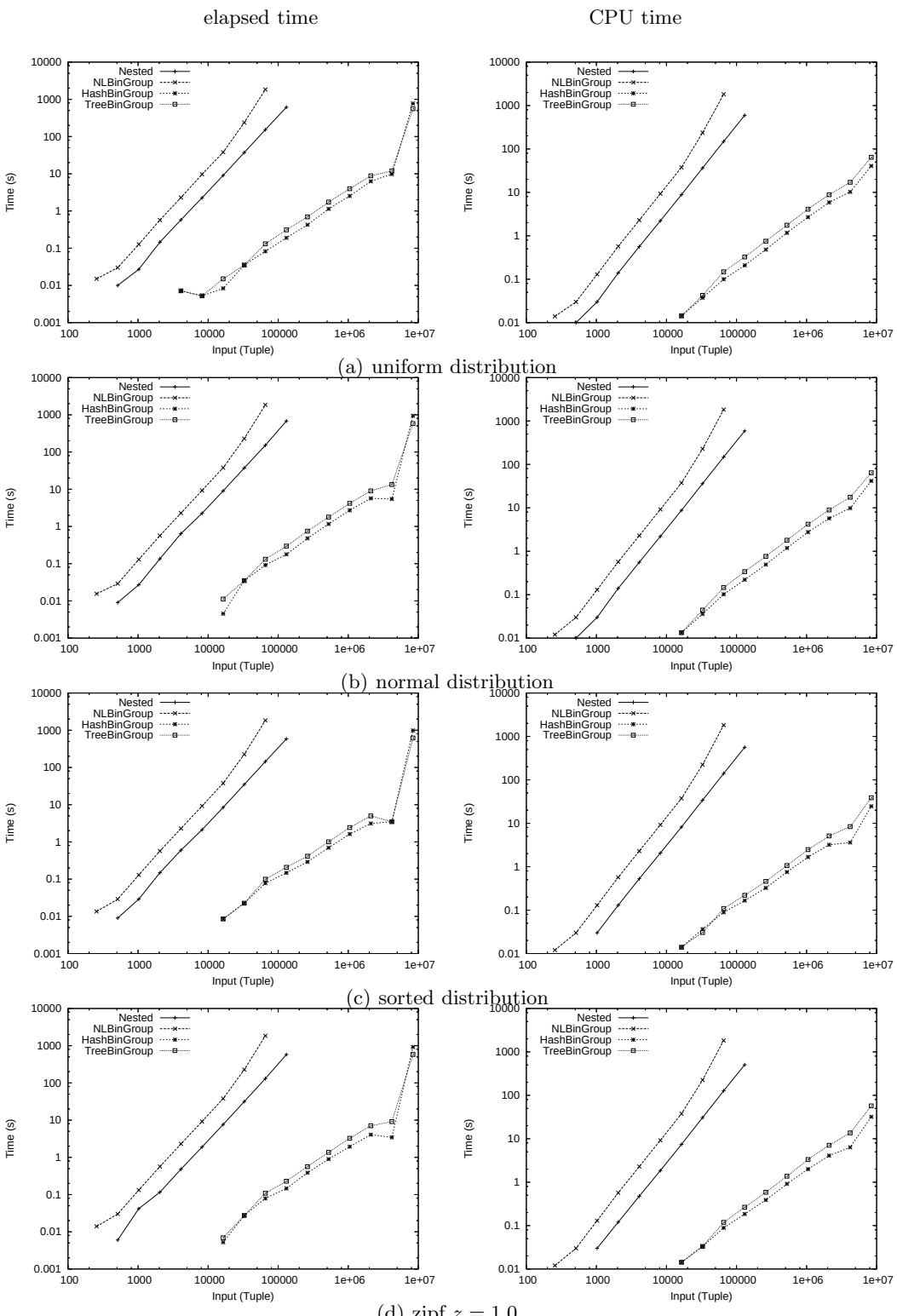
(a) uniform distribution

(b) normal distribution

(c) sorted distribution

(d) zipf $z = 1.0$

Fig. 9. Predicate $\neq$, group input uniform



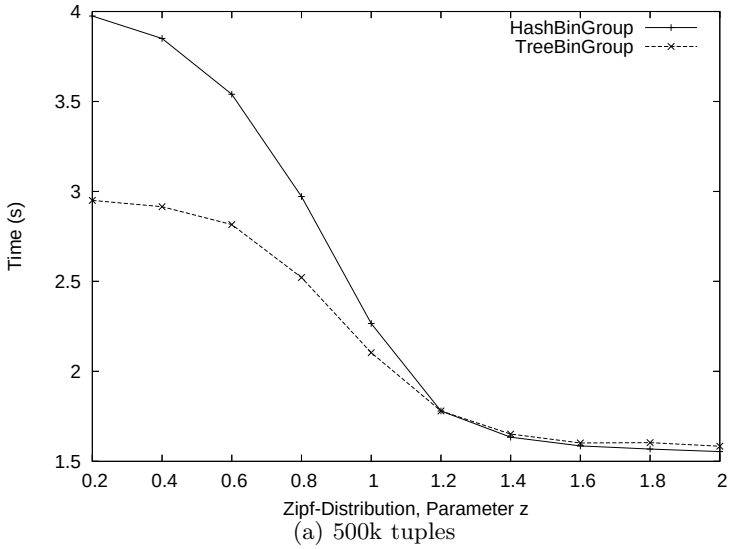
(a) uniform distribution

(b) normal distribution

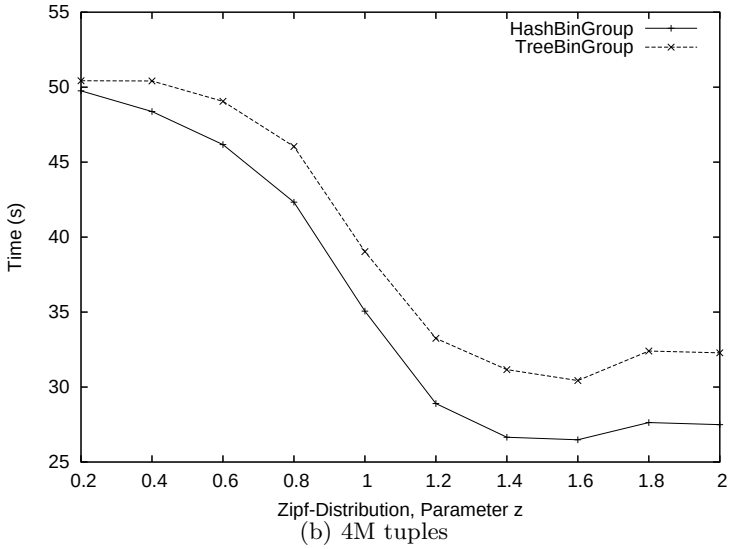
(c) sorted distribution

(d) zipf $z = 1.0$

Fig. 10. Predicate $>$, group input uniform



(a) 500k tuples



(b) 4M tuples

Fig. 11. Sensitivity to Zipf distribution

Sort-Based Algorithms For predicate $\neq$ and sorted grouping input, we compare NESTED, NESTEDSORT, NLBINGROUP, and EQBINGROUP with SORTBINGROUP. The results from the experiments are summarized in Figure 12.

Similar to uniformly distributed grouping input nested-loop-based algorithms – including SORTBINGROUP and NESTEDSORT – perform poorly. The sort-based algorithms exploit the sortedness of the grouping input, making it superior to the hash-based alternative NLBINGROUP. However, the weakness of NLBINGROUP is not as prominent here as in the previous algorithms. In this scenario EQBINGROUP is still the most efficient algorithm because of its linear run time. SORTBINGROUP is only a choice for arbitrary predicates.

For sorted grouping input, sorted aggregation input and predicate $>$, we can employ all algorithms but EQBINGROUP. Figure 13 shows again that nested-loop-based algorithms perform poorly compared to the improved algorithms.

Among those, only LTSORTBINGROUP exhibits the sorted input, and thus performs fastest. HASHBINGROUP and TREEBINGROUP are about one order of magnitude slower than LTSORTBINGROUP. This comes at no surprise when we compare the runtime complexities we have deduced from the algorithms in Section 3. LTSORTBINGROUP is more efficient in recognizing the groups. In addition, matching the aggregate input to these groups is more efficient by exploiting the sortedness on both inputs. However, in our experimental setup LTSORTBINGROUP also suffers from swapping memory pages.

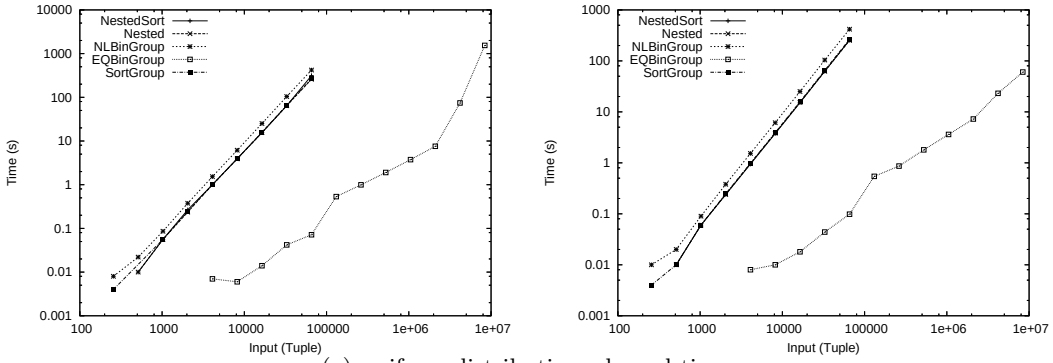
Validation of the Complexity Formulas To verify our complexity formulas we employed regression analysis. We based our analysis on the measured CPU time because the time complexity we want to verify is a model for the computational effort of an algorithm. In our analysis we restricted ourselves to sorted input on both inputs because all algorithms are applicable in this setting.

We used linear regression on the observed CPU times for the algorithms with linear time complexity. For HASHBINGROUP and TREEBINGROUP we also applied linear regression on the logarithmic values for the size and time values. For quadratic complexity functions we utilized curve fitting for quadratic polynomials. In the regression models, the size x is the independent variable, and the observed execution time is the dependent variable. We simplified the regression models to use only one independent variable $x = |e_1| = |e_2|$ which conforms with our experimental setup.

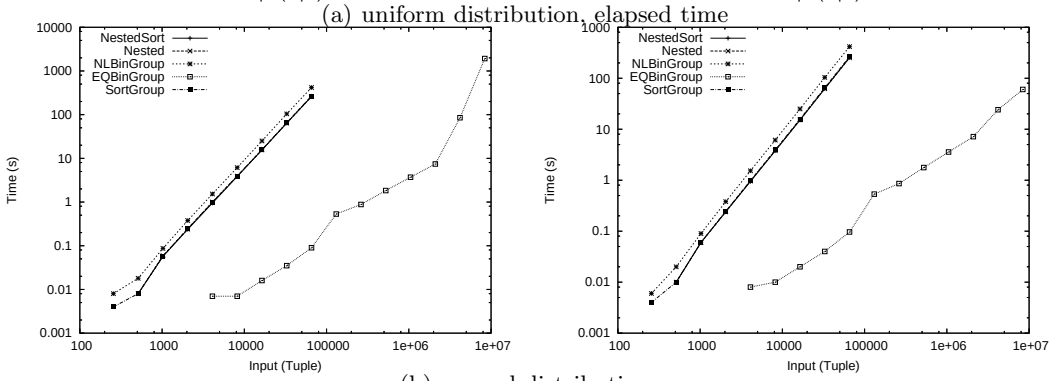
The table below summarizes our results. For each algorithm we repeat the theoretical time complexity as deduced in Section 3. The third column contains the results from the regression analysis. With R^2 we refer to the *coefficient of determination*, with $-1 \leq R^2 \leq 1$. Values of $|R^2|$ close to 1 mean a close fit of the function computed during regression analysis and the measured data.

Algorithm	Complexity	Regression	R^2
NESTED	$\frac{l}{f}\|e_1\|\|e_2\|$	$5.27 \cdot 10^{-8} \cdot x^2 -$ $7,03 \cdot 10^{-5} \cdot x + 0.13$	0.99999871
NLBINGROUP	$\frac{l}{f}\|e_1\|\|e_2\| + (l + \frac{1}{f})\|e_1\|$	$9,28 \cdot 10^{-8} \cdot x^2 -$ $9.30 \cdot 10^{-5} x + 0.10$	0.99999961
HASHBINGROUP	$(l + \frac{1}{f})\|e_1\| +$ $O((\frac{\|e_1\|}{f} + \|e_2\|) \lg \frac{\|e_1\|}{f})$	$x^{1.10} \cdot 10^{6.34}$	0.98061024
TREEBINGROUP	$\frac{\|e_1\|}{f} + O((\|e_1\| + \|e_2\|) \lg \frac{\|e_1\|}{f})$	$x^{1.10} \cdot 10^{6.34}$	0.98046729
EQBINGROUP	$l(\|e_1\| + \|e_2\|) + \frac{\|e_1\|}{f}$	$4.81 \cdot 10^{-6} \cdot x - 1.35$	0.94482793
NESTEDSORT	$\frac{1}{f}\|e_1\|\|e_2\|$	$5.39 \cdot 10^{-8} \cdot x^2 -$ $0.001 \cdot x + 0.21$	0.99999682
SORTBINGROUP	$\frac{1}{f}\|e_1\|\|e_2\|$	$5.36 \cdot 10^{-8} \cdot x^2 -$ $7.28 \cdot 10^{-5} \cdot x + 0.13$	0.99999879
LTSORTBINGROUP	$\|e_1\| + \|e_2\|$	$9.97 \cdot 10^{-7} \cdot x - 0.72$	0.80008

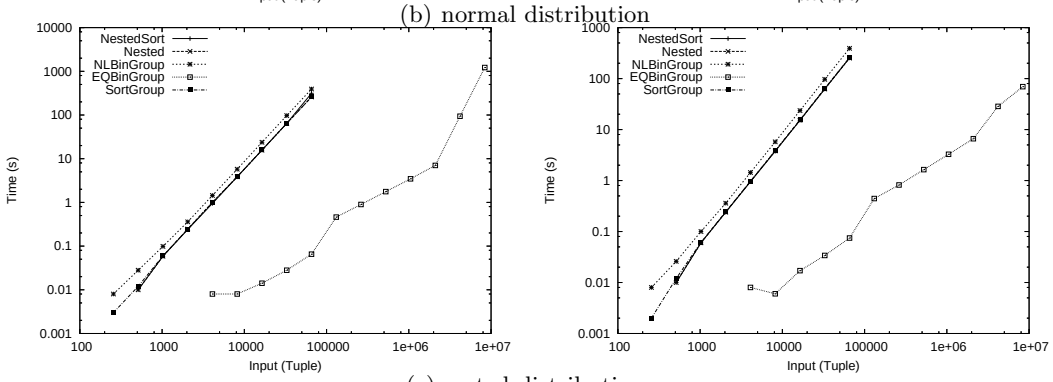
elapsed time CPU time



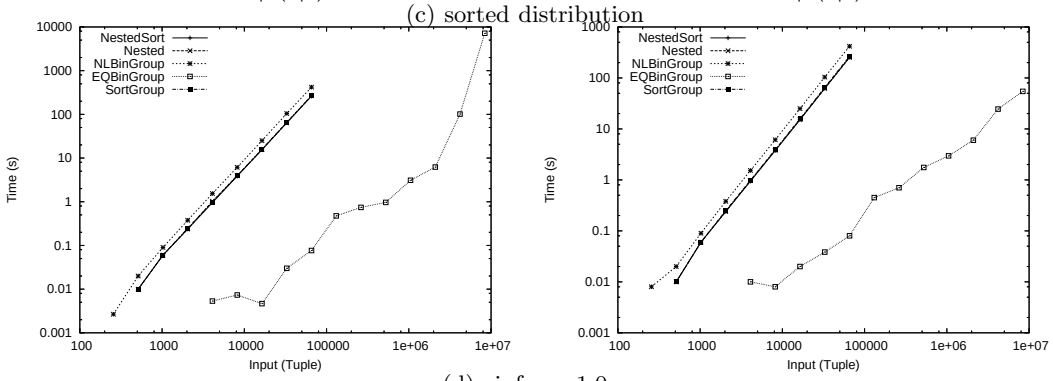
(a) uniform distribution, elapsed time



(b) normal distribution



(c) sorted distribution



(d) zipf $z = 1.0$

Fig. 12. Predicate $\neq$, group input sorted

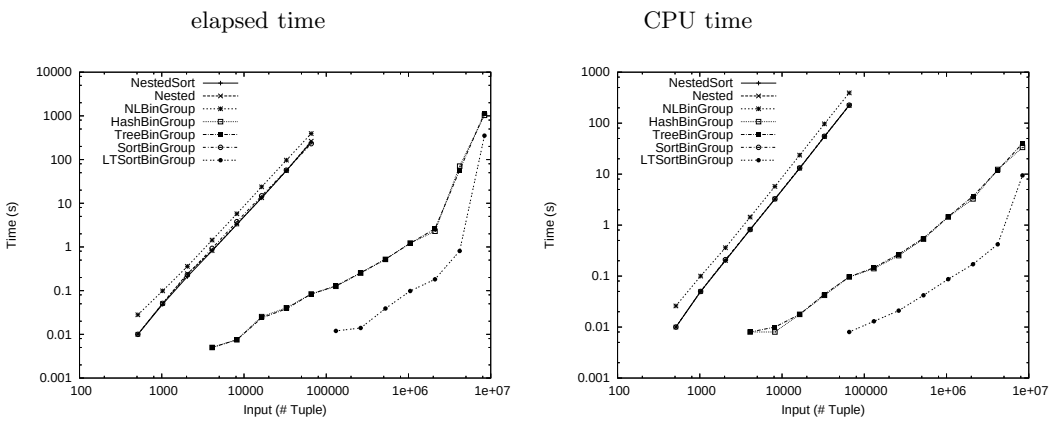


Fig. 13. Predicate $>$, group input and aggregation input sorted

As the main result we observe that all regression formulas yield a coefficient of determination close to 1. The lower coefficient of determination for LTSortBinGroup can be explained by the last data point which is caused by the overhead of page swapping. Thus, regression analysis strongly supports the validity of our time complexity formulas. However, the regression formulas for HashBinGroup and TreeBinGroup need further explaination.

As mentioned above, each measured time was transformed into double logarithmic space with basis 10, i.e. for a time t measured for size x we get a tuple $(\log(x), \log(t))$. Let $f(x) = a{\cdot}x{+}b$ be the linear function computed in the linear regression. Then the transformation back gives $10^{a\cdot\log(x)+b} = x^a \cdot 10^b$. These results are given in the table above. Since the dominant term in the complexity formulas of HashBinGroup and TreeBinGroup is $x \log x$. we need to validate that our findings are a good approximation for this complexity formula. Hence, we compute upper and lower bounds for $x \log x$: We know that $\frac{x}{x+1} \leq \log(x+1) \leq x$ (cf. [6]). For the lower bound, we use $\frac{x}{x+1} \leq \log(x+1)$, thus $x < x \log x$ for $x > 1$. For the upper bound we use $\log^b x = o(x^a)$ for some constant $a > 0$ – we choose $b = 1$. Thus, we can deduce $x \log x = o(x^{1+a})$. A change into basis 2 for the logarithm only changes the constants involved. Summarizing, our results validate the complexity formulas deduced in Section 3.

5 Extensions

In this section we show how our implementations can be extended to support other bulk types. Also, we explain how the restrictive conditions on the predicates for the algorithms EQBinGroup, HashBinGroup, TreeBinGroup, and LTSortBinGroup can be relaxed.

5.1 Supporting Different Bulk Types

The implementations and the experiments are based on totally ordered sets as bulk types. In this section, we review how other bulk types can be supported.

The hash tables used in the binary grouping operators remove duplicate values on groups. For bags and sequences, it is necessary to keep duplicate values, which can be done by keeping a list of duplicates for each group or appending duplicates to the collision list of the bucket in the hash table. While the first alternative results in faster lookups, the data structures need to be modified to handle these lists of duplicates. In the second alternative f in the complexity formulas becomes 1.

In both cases the insert and lookup functions of the hash tables can be modified to accommodate this change. Function insert must ignore duplicate groups. A series of calls to function lookup must successively return the matching groups. E.g. an iterator can be applied

```
OPEN
 1  open e1
 2  ζα(GT)
 3  while T ← next e1
 4        do
 5           G ← HT.INSERT(T)
 6           ζα(G)
 7  close e1

 8  open e2
 9  while T ← next e2
10        do while G ← HT.LOOKUP(T)
11                 do
12                    G ← α(G, T)
13           if predicate is ≠
14              then GT ← α(GT, T)
15  close e2
16  htIter ← HT.ITERATOR
```

Fig. 14. EQBINGROUP with bag or sequence semantics

R	
A	B
1	a
1	b
2	b

S	
C	D
1	b
1	c
2	b
2	c

Fig. 15. Example data

for this purpose. The algorithms we presented must be adjusted such that aggregation is done on each member of the group. Fig. 14 shows how, e.g. EQBINGROUP needs to be changed. In line 5 all groups are inserted regardles of duplicates and in line 10 all those need to be retrieved from the hash table. The functions NEXT and CLOSE remain unchanged.

Order-preserving implementations can be supported by recording the insertion order of groups into the hash table or the search tree. This can either be achieved by keeping a linked list with pointers to the inserted tuples or by directly linking the groups in the hash table. When the aggregated result is returned, this linked structure is traversed.

5.2 Multiple Conjunctive Clauses

We have already mentioned that the algorithm for EQBINGROUP assumes that all clauses in the conjunctive predicate are of the same type. To see the necessity for this restriction, consider the two tables R and S in Figure 15.

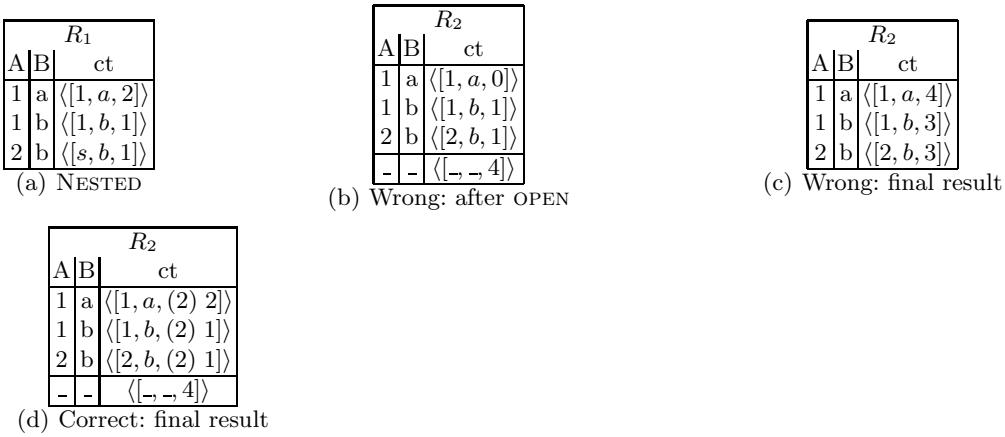
Let us first investigate the effect of conjunctive predicates on EQBINGROUP. Consider the nested expression

$$R_1 := \chi_{ct:count(\sigma_{A=C \wedge B \neq D}(S))}(R)$$

The result of this expression is given in Figure 16(a). According to the definition of the binary grouping operator the equivalent expression using this operator is

$$R_2 := (R)\Gamma_{ct;A=C \wedge B \neq D;count}(S)$$

To see why we cannot apply the algorithm EQBINGROUP blindly we show in Figure 16(b) the intermediate result after matching all tuples but before computing the final result for



R_1

A	B	ct
1	a	$\langle[1, a, 2]\rangle$
1	b	$\langle[1, b, 1]\rangle$
2	b	$\langle[s, b, 1]\rangle$

(a) NESTED

R_2

A	B	ct
1	a	$\langle[1, a, 0]\rangle$
1	b	$\langle[1, b, 1]\rangle$
2	b	$\langle[2, b, 1]\rangle$
-	-	$\langle[\_, \_, 4]\rangle$

(b) Wrong: after OPEN

R_2

A	B	ct
1	a	$\langle[1, a, 4]\rangle$
1	b	$\langle[1, b, 3]\rangle$
2	b	$\langle[2, b, 3]\rangle$

(c) Wrong: final result

R_2

A	B	ct
1	a	$\langle[1, a, (2)\ 2]\rangle$
1	b	$\langle[1, b, (2)\ 1]\rangle$
2	b	$\langle[2, b, (2)\ 1]\rangle$
-	-	$\langle[\_, \_, 4]\rangle$

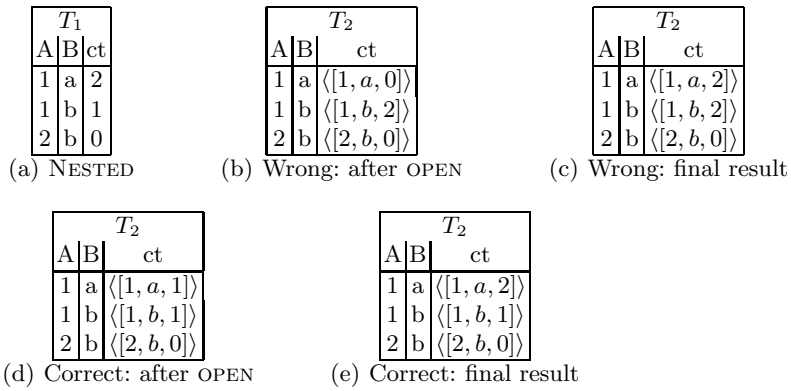
(d) Correct: final result

Fig. 16. Conjunctive symmetric predicates

each group using function β^{-1}. The last row in the table is the auxiliary tuple that aggregates all tuples. As you can see in Figure 16(c) this leads to the wrong final result because we cannot subtract the aggregated result of a subgroup (e.g. with attribute value of $A = 1$) from the aggregated result over all groups.

We propose the following solution: When any clause in the conjunctive predicate is an equivalence relation, we match groups only using the attributes involved in all conjuncts containing equivalence relations. Only for matching individual subgroups, we use the complete predicate. In this sense the clauses that are not an equivalence relation form a *residual predicate*. In this case, we need not combine results of individual groups with the auxiliary aggregation tuple. Note that this solution works for arbitrary residual predicates.

Resuming with the example, Figure 16(d) shows the content of the hash table after matching all tuples. In the column *ct* the numbers in the parenthesis are the number of tuples that matched the subgroup (e.g. 2 for $A = 1$ and 2 for $A = 2$). The last value in the last column is the final result of aggregation. It is computed by matching within the subgroup using the residual predicate. We need not apply function β^{-1} because matching the groups was done with an equality predicate.



T_1

A	B	ct
1	a	2
1	b	1
2	b	0

(a) NESTED

T_2

A	B	ct
1	a	$\langle[1, a, 0]\rangle$
1	b	$\langle[1, b, 2]\rangle$
2	b	$\langle[2, b, 0]\rangle$

(b) Wrong: after OPEN

T_2

A	B	ct
1	a	$\langle[1, a, 2]\rangle$
1	b	$\langle[1, b, 2]\rangle$
2	b	$\langle[2, b, 0]\rangle$

(c) Wrong: final result

T_2

A	B	ct
1	a	$\langle[1, a, 1]\rangle$
1	b	$\langle[1, b, 1]\rangle$
2	b	$\langle[2, b, 0]\rangle$

(d) Correct: after OPEN

T_2

A	B	ct
1	a	$\langle[1, a, 2]\rangle$
1	b	$\langle[1, b, 1]\rangle$
2	b	$\langle[2, b, 0]\rangle$

(e) Correct: final result

Fig. 17. Conjunction of antisymmetric predicates

A similar problem and solution must be used for antisymmetric predicates. For TREEBINGROUP, HASHBINGROUP, and LTSORTBINGROUP, we restricted the predicate only to one clause. To see why multiple conjunctive clauses must be handled carefully, we use the

data in Figure 15 again. Consider the following example expression:

$$T_1 := \chi_{ct:count(\sigma_{A<C \wedge B<D}(S))}(R)$$

The correct result for this expression is shown in Figure 17(a). Using the binary grouping operator, we would expect that either TreeBinGroup, HashBinGroup, or LTSortGroup should produce the correct result.

$$T_2 := (R)\Gamma_{ct;A<C \wedge B<D;count}(S)$$

Restricting ourselves to the evaluation of HashBinGroup, Figure 17(b) shows the intermediate result after matching all tuples, and Figure 17(c) contains the final result. Naive application of either algorithm stumbles over the same obstacle that we have identified for EQBinGroup: Both algorithms will not find the closest match that satisfies the predicate. In the example the third tuple of S does not match with the first tuple of R, but with the second.

The solution is the same as for conjunctive predicates containing clauses with equivalence relations: For matching groups we may only use one pair of attributes. The remaining attributes must be applied as residual predicates in the same way as it is done in a B-Tree index. Figure 17(d) and 17(e) trace the steps of the algorithms to the correct solution. In the example $A < C$ is used to find the closest group. Only within the group the residual predicates are applied. This modification results in the closest possible match and to the correct result.

Note that these extensions can be incorporated into the hash tables discussed before. Finding the correct hash bucket is done with the same hash function. But we employ a compare function for identifying groups and a compare function for matching individual tuples. The latter extends the first by the residual predicate. In addition to that, the hash table then needs to support duplicates.

6 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, 21, 4]. Only [4] 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 [23] 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 [13].

Our work also relates to work comparing sort-based and hash-based implementations of algebraic operators [8, 10, 11, 14, 15, 20]. However, they concentrate on implementations of equijoins. Non-Equality joins have been studied first in [9]. A generic framework for Non-Equality joins is described in [7] where an appropriate index structure is employed for partioning and matching. Using e.g. a B+-tree for partitioning and probing is similar to the idea of the algorithm TreeBinGamma.

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, 7, 10, 16, 17] for such proposals.

Our implementations are orthogonal to the optimization techniques for queries containing grouping [22, 3, 24]. When pushing grouping before a join is not possible, our algorithms can be used to improve query execution.

Full support of binary grouping in query optimizers requires algebraic equivalences that hold for this operator. [2, 5] present algebraic equivalences in an unordered context. Which of these equivalences hold on sequences is not yet investigated. Binary grouping can also benefit from preagregation as proposed by Larson [18].

7 Conclusion and Future Work

Binary grouping is a powerful operator to evaluate analytic queries [2] or to unnest nested queries [5, 19]. 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.

We summarize algebraic equivalences that can be incorporated into query optimizers for deriving efficient query evaluation plans. Possible extensions of our algorithms to bags or sequences have been presented. 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.

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. Chaudhuri and K. Shim. Including group-by in query optimization. *Proc. VLDB*, pages 354–366, 1994.
4. S. Cluet and G. Moerkotte. Efficient evaluation of aggregates on bulk types. *Proc. of 5-th DBPL*, 1995.
5. S. Cluet and G. Moerkotte. Nested queries in object bases. Technical Report RWTH-95-06, GemoReport64, RWTH Aachen/INRIA, 1995.
6. T. Corman, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.
7. J. V. d. Bercken, M. Schneider, and B. Seeger. Plug&join: An easy-to-use generic algorithm for efficiently processing equi and non-equi joins. In *EDBT '00*, pages 495–509, 2000.
8. 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.
9. D. J. DeWitt, J. F. Naughton, and D. A. Schneider. An evaluation of non-equijoin algorithms. In *Proc. VLDB*, pages 443–452, 1991.
10. G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.
11. G. Graefe. Sort-merge-join: An idea whose time has(h) passed? In *Proc. ICDE*, pages 406–417, 1994.
12. G. Graefe. Executing nested queries. In *BTW*, pages 58–77, 2003.
13. G. Graefe, R. Bunker, and S. Cooper. Hash joins and hash teams in Microsoft SQL server. In *Proc. VLDB*, pages 86–97, 1998.
14. G. Graefe, A. Linville, and L. D. Shapiro. Sort vs. hash revisited. *IEEE TKDE*, 6(6):934–944, December 1994.
15. 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.
16. S. Helmer, T. Neumann, and G. Moerkotte. Early grouping gets the skew. Technical Report TR-02-009, University of Mannheim, 2002.
17. S. Helmer, T. Neumann, and G. Moerkotte. A robust scheme for multilevel extendible hashing. *Proc. 18th ISCIS*, pages 218–225, 2003.
18. P.-Å. Larson. Data reduction by partial preaggregation. In *Proc. ICDE*, pages 706–715. IEEE Computer Society, 2002.

19. N. May, S. Helmer, and G. Moerkotte. Nested queries and quantifiers in an ordered context. *Proc. ICDE*, pages 239–250, 2004.
20. D. E. Simmen, E. J. Shekita, and T. Malkemus. Fundamental techniques for order optimization. *SIGMOD Record*, 25(2):57–67, 1996.
21. 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.
22. X. Wang and M. Cherniack. Avoiding sorting and grouping in processing queries. *Proc. VLDB*, pages 826–837, 2003.
23. T. Westmann and G. Moerkotte. Variations on grouping and aggregation. Technical report, University of Mannheim, 1999.
24. W. P. Yan and P.-Å. Larson. Performing group-by before join. *Proc. ICDE*, pages 89–100, 1994.