Reihe Informatik

11 / 1999

Variations on Grouping and Aggregation

Till Westmann Guido Moerkotte

Variations on Grouping and Aggregation

Till Westmann Guido Moerkotte

Lehrstuhl für Praktische Informatik III
Universität Mannheim
68131 Mannheim
Germany

westmann|moerkotte@informatik.uni-mannheim.de

May 31, 2002

Abstract

Grouping and aggregation are constantly gaining importance in the evaluation of queries. Hence, there is a need to improve the execution times of queries containing these operations. To achieve this goal we propose to extend the relational algebra by new operators that address different query patterns and that allow for more efficient implementation than those currently used. These new operators have to be specific enough to allow improved performance and general enough to be of general use. We present three patterns and the corresponding operators and show how these operators can be used to speed up query evaluation by a factor of two.

1 Introduction

One of the main advantages of declarative query languages such as SQL is their optimizability. The equivalence of calculus—the formal foundation of a declarative query language—and algebra—the language for specifying query execution plans—forms the basis for optimizability. Moreover, in order to be able to optimize a query, different query evaluation plans must exist for a given query with different associated costs. The set of query evaluation plans equivalent to the original query defines the search space a query optimizer must explore in order to find a low cost plan. There are two factors that imply that the search space contains more than a single plan: algebraic equivalences allow to construct different plans at the logical level and different implementations of algebraic operators allow for further alternatives at the physical level.

The tradition has been to provide different implementations for a single logical algebraic operator, leading to a one-many relationship between operators and their implementations. For example, the join operator comes with many implementation alternatives such as nested-loop join, sort-merge join and hash join [Gra93]. Our hypothesis is that further efficiency improvements can be gained, if special implementations for special cases of algebraic operators or combinations thereof are provided. We will identify three special cases, define according algebraic operators, and provide their efficient implementations. Every case will cover a special intent of the user stating the query. As we will see, this intent is reflected within the query by a certain pattern. Hence, we will refer to such special cases as *semantic query patterns*. The first two operators cover different semantic query patterns of grouping with aggregation. The last operator covers a semantic query pattern matching a combination of grouping with aggregation and join.

Of course, when identifying these semantic query patterns, the trade off between generality and efficiency has to be considered carefully. The more special a pattern is, the more likely an efficient implementation is but the less useful it might be. This is one of the reasons why we consider only cases that were identified as important by other people, too. All presented cases can be found in the TPC-D benchmark [TPC95]. As a positive side effect, we were able to use the TPC-D data to evaluate the performance of our implementations.

We concentrate on cases involving grouping and aggregation since the use of databases for Data Warehouse and OLAP applications increased drastically in recent years. Within these applications, grouping and aggregation play a major role. Witness, again, the TPC-D benchmark [TPC95]. Out of 17 queries 17 queries contain either grouping or aggregation or both. It is therefore very useful to improve the execution of these operations.

Grouping and aggregation have quite some history. Klug [Klu82] was the first who gave precise definitions for aggregate functions and extended relational algebra and calculus to support these. Ceri and Gottlob [CG85] generalized Klug's *aggregate formation* operator to allow translation of SQL to relational algebra. Dayal [Day87] gave an implementation of this generalized *GAgg* operator and described tactics to include aggregation into query optimization. All semantic query patterns we identify will involve the *GAgg* operator. Implementation techniques for *GAgg* can be found in Epstein's Memorandum [Eps79], in Dayal's paper [Day87], and in Graefe's survey [Gra93]. Shatdal and Naughton provide implementation techniques for parallel versions [SN95]. However, we concentrate on sequential implementations. Chatziantoniou and Ross describe an extension of SQL that allows for easier formulation of some queries containing grouping and aggregation. Further they introduce the relational operator Φ that facilitates the search for an efficient implementation of the queries considered [CR96]. Nevertheless, the described evaluation algorithm does lead–if applicable–to the ususal implementations in the cases addressed by our operators. Applying our special operators leads to twofold improvements in speed or memory utilization compared to those found in [Gra93].

The rest of the paper is organized as follows. Section 2 contains the preliminaries. It reviews the definition of the conventional *GAgg* operator and its implementation. Sections 3-5 are devoted the semantic query patterns. Each section introduces a pattern by means of an example, defines an according operator, gives its implementation, compares an original query plan with one using the new operator, and exemplifies performance gains. Section 6 concludes the paper.

2 Preliminaries

In SQL, aggregates can be formed within or without a grouping context. As an example of the latter consider the query

```
select  sum(Salary), max(Salary)
from    Employee
```

where *Employee* is a relation with schema

$$E = (\text{EmpID} : \text{int}, \text{Name} : \text{string}, \text{Salary} : \text{decimal}, \text{Dept} : \text{string}).$$

The query sums up the salary of all employees and retrieves their maximum salary. Note that a vector of scalar aggregate functions is applied to a set of tuples, i.e. a relation. A *scalar aggregate* returns a single number for a given input relation [Eps79]. Examples thereof are *count*, *sum*, *avg* etc. A vector of aggregate functions that returns a set (or tuple) of values for a given input relation is sometimes called *aggregate function* [Eps79]. Within the algebra, we denote the application of a vector of scalar aggregate functions *agg* by the operator γ_{agg} (pronounced *small gamma*) defined below.

As an example for aggregation in the context of grouping consider the query

```
select     sum(Salary), max(Salary), Dept
from       Employee
group by   Dept
```

Here, a vector of scalar aggregate functions is applied to every group of employees. Every group consists of a set of tuples. The groups themselves are specified by the *group by* clause. This combination of grouping with a successive application of a vector of scalar aggregate functions to each group is captured by the operator $\Gamma_{A;agg}$. This operator corresponds to *generalized aggregation* operator [Day87].

In the subsequent definitions of these two operators, we use Maier's notation [Mai83]: lowercase letters are used for relations and uppercase letters are used for attribute-sets and relational schemas, $r(R)$ signifies that r is a relation with schema R. Using these symbols the two operators are defined as follows:

Let $r(R)$ be a relation and $agg : \mathcal{P}(r) \rightarrow o$ be a vector of scalar aggregate functions returning a tuple. Then

$$\begin{aligned} \gamma_{agg}(r) &:= \{agg(r)\} \\ \Gamma_{A;agg}(r) &:= \{t(A) \circ a : t \in r \wedge a = agg(\{s \in r : s(A) = t(A)\})\}. \end{aligned}$$

Example Suppose e is a relation of employees with schema

$$E = (\text{EmpID} : \text{int}, \text{Name} : \text{string}, \text{Salary} : \text{decimal}, \text{Dept} : \text{string})$$

and its extent is

$$\begin{aligned} e \;= \{ \;\; & (1, \text{“Smith”}, 3000.00, \text{“production”}), (2, \text{“Miller”}, 2800.00, \text{“production”}), \\ & (3, \text{“Clark”}, 3300.00, \text{“sales”}), (4, \text{“Hill”}, 3500.00, \text{“sales”}), \\ & (5, \text{“Barth”}, 3100.00, \text{“sales”}) \;\; \}. \end{aligned}$$

If we define agg for all $q \in \mathcal{P}(e)$ as

$$agg(q) := \left(\sum_{t \in q} t.Salary \;,\; \max_{t \in q}(t.Salary) \right),$$

we can get the sum of all salaries and the maximum salary using the γ-operator:

$$\gamma_{agg}(e) \quad = \quad \{(15700.00, 3500.00)\}.$$

If we want the sum of all salaries and the maximum salary *per department* we use the Γ-operator:

$$\Gamma_{(Dept);agg} \quad = \quad \{(\text{“production”}, 5800.00, 3000.00), (\text{“sales”}, 9900.00, 3500.00)\}.$$

Today, there exist three basic methods to implement the Γ operator: one based on nested loops, a second based on sorting and a third based on hashing [Gra93]. As the nested loop algorithms show a much weaker performance than the other two types and as its specific properties are not needed in normal RDBMS, they are rarely used. According to Graefe [Gra93] sorting and hashing are of nearly equal performance so in a RDBMS there is usually one module that implements grouping, aggregation and duplicate removal using one of the latter two approaches. Note that the situation is very similar to the join operator. For a single algebraic operator, several implementations exist. Each implementation implements the full functionality of the operator– no more and no less. Subsequent sections will introduce algebraic operators for semantic query patterns that cover special cases of γ, Γ, and a combination of Γ and $\bowtie$.

3 The Max-Operator

This and the following two sections consist of five parts each. We start with a description of a semantic query pattern, continue with a definition the operator capturing the semantics of the pattern. Then, we sketch a possible implementation of the operator. The fourth part contains a comparison of two query evaluation plans. The first is a traditional plan and the second plan utilizes the newly introduced operator. The fifth part reports experimental results on the performance gains.

All performance experiments were carried out by integrating the newly proposed operators into our experimental database management system AODB. The queries were run against the TPC-D database with a scaling factor of 1 [TPC95]. AODB was running on a lightly loaded UltraSparc2 with 256 MByte main memory running under Solaris 2.6.

3.1 Semantic query pattern

Example Suppose we have a relation Employee with schema

Employee(EmpID, Name, Salary, DeptID).

Against this relation, we pose the following query:

Retrieve the employee with the highest salary.

Its translation into SQL yields

```
select Name
from   Employee
where  Salary = ( select max(Salary)
                  from   Employee)
```

Most commercial DBMSs (at least all we had access to) take the given SQL-statement quite literally. The result is generated in two steps. In the first step, the subquery is used to find the maximum value for *Salary*. In the second step all qualifying tuples from the relation *Employee* are selected. Both steps could either be done using an index or by scanning the relation *Employee*. In any case the system would have to look at the salaries and the corresponding tuples twice. Obviously this is neither desirable nor necessary, especially if the number of result tuples is small and easily fits into main memory. Furthermore, this will typically be the case. It is even more undesirable, if the relevant set of tuples is not a base relation but the result of a subquery (e.g. because *Employee* is a view). In this case, we definitely do not have an index to look up our tuples and we have to execute the whole subquery twice or materialize its result and scan it.

The general semantic query pattern selects

"all tuples exhibiting a maximum value for a given expression."

The following discussion only talks about maximization. Minimization can be treated analogously. Hence, we call this pattern the *global extremum pattern*.

3.2 Operator definition

This pattern leads directly to the following definition of the Max-operator, which is an adaption of the Max-operator proposed by Cluet and Moerkotte [CM93] to the relational context.

Let $r(R)$ be a relation, X an ordered set of atomic values and $exp : r \rightarrow X$ an expression that should be maximized. We define

$$\mathrm{Max}_{exp}(r) \quad := \quad \{t \in r : exp(t) = \max(exp(r))\}$$

In contrast to this definition the operator from [CM93] returns a nested result, which is unsuitable for our purpose since relations in the traditional relational model are flat. Further, Cluet and Moerkotte did not give an implementation.

We propose to use the Max-operator to evaluate queries showing the global extremum pattern.

3.3 Operator implementation

A simple, efficient, but naive implementation for this operator is

```
MAX(r,exp)         /* relation r, expression exp to be maximized */
M := emptyset
choose m from r
add m to M
foreach t in r
do
  if exp(t) > exp(m)
  then
    M := emptyset
    m := t
    add m to M
  else if exp(t) = exp(m)
    add t to M
done
return M
```

This implementation touches each tuple contained in r exactly once and is therefore more efficient than the usual "2-step-algorithm". The only problem with this implementation may be the size of M. There is no problem as long as there is sufficient memory to hold M. But this might not be the case, even if all result tuples fit into memory. To understand why, imagine the *Employee* relation taking the role of r. Assume that every employee has one of three salaries 20k, 200k, 2000k and that the *Employee* relation is accidentally sorted on increasing salaries. Further, most of the 10 million employees will exhibit the 20k salary, fewer the 200k salary, and almost none the 2000k salary. Then, the operator implementation would first fill M with all employees with 20k. As soon as it encounters the first employee with the 200k salary, it discards the current M and initializes it with the first employee earning 200k. The same procedure starts again. Hence, M is bound in turn to all employees with salary 20k, 200k, and 2000k. If one of these sets exceeds the memory capabilities, the algorithm is in trouble. This case should however be very unlikely but we better prepare for it.

One remedy could be to repeatedly write tuples to disk and delete them, if a new maximum is found. This is very undesirable as it would lead to writing I/O. There is, however, an easy way to dynamically handle memory shortage with the following slightly modified implementation:

```
  MAX(r,exp)         /* relation r, expression exp to be maximized */
  M := emptyset
  choose m from r
  add m to M
* overflow := false
  foreach t in r
  do
    if exp(t) > exp(m)
    then
      M := emptyset
*     overflow := false
      m := t
      add m to M
    else if exp(t) = exp(m)
*     if M is not full
        add t to M
*     else
*       overflow := true
  done
* if(overflow = false)
    return M
* else
*   choose m from M
*   return all tuples t from r with exp(t) = exp(m)
```

The main idea of this modification is to fall back to the "2-step-algorithm", if there is not enough memory. As soon as there are too many tuples in M, the *overflow* flag is set and no more tuples are collected until a tuple is found that exhibits a higher value for *exp*. Then, we can restart with a small M. At the end of the algorithm, we analyze the current situation and either return M directly, or—in case of an overflow—have to perform a second scan on r. This fall back technique will be applied to the next operator's implementation as well.

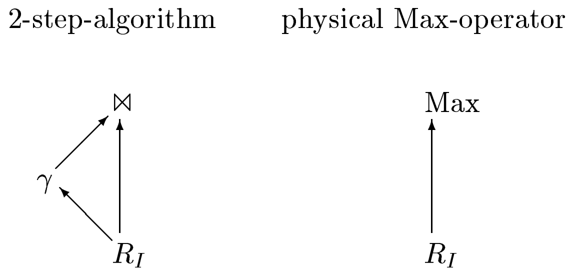

Figure 1: Execution plans for the (logical) Max-operator

3.4 Comparison of query execution plans

In figure 1 the two possible execution plans are shown. The "2-step-algorithm" consists—when described in terms of relational operators—of a γ- and a join-operator. As the smaller input of the join-operator consists of just one tuple any low-overhead join implementation is well-suited. Note however, that every tuple from R_I is accessed twice by the "2-step-algorithm": once as an input to the γ-operator and once for the right input of the join-operator. The modified implementation of the Max-operator needs each tuple once or twice depending on the size of the result. As the calculation of the maximum is computationally very easy, the time used for the production of the tuples of R_I takes usually the predominant part of the total execution time. Therefore we expect a reduction of the total execution time for queries exhibiting the global extremum pattern by 50% if the of Max-operator is applied and the result fits into main memory. The latter condition is quite likely to be fulfilled. And even if it is not the case the modified implementation reverts to the "2-step-algorithm" and is thus not slower than the conventional plan.

Remark on indices Obviously, if there is an index on the *Salary* attribute, this index can be used to answer the example query. This plan will most likely be more efficient than our plan. However, we can not expect to have an index on every attribute. Furthermore there are two cases in which we definitely do not have an index: (1) if the relation considered is produced by a subquery instead of a base relation and (2) if we do not want to maximize a mere attribute but a more complex expression.

plan	total time	CPU time
"2-step-algorithm"	39 s	9.0 s
Max-operator	19 s	5.7 s

Table 1: Execution times for the Max-operator

Remark on optimization If the query exhibits the global extremum pattern, the query evaluation performance is never deteriorated by using the Max-operator. Therefore we can introduce the Max-operator into the evaluation plan during the rewrite-phase *before* alternative plans are generated. So the introduction of the Max-operator does not lead to an increased size of the optimizer's search space.

3.5 Experimental Result

To show that the Max-operator can deliver the promised performance gain we executed the following query twice—once using the "2-step-algorithm" and once using our implementation of the Max-operator.

```
select O_Clerk, O_TotalPrice
from   Order
where  O_TotalPrice = (select max(O_TotalPrice)
                       from   Order)
```

The results presented in table 1 show the expected improvement of 50%.

4 The Γ^{max}-Operator

4.1 Semantic query pattern

Example Consider the following query:

> For every department retrieve the employees that within this department have the highest salary.

Using the relation *Employee(EmpID, Name, Salary, DeptID)* from the last section the appropriate SQL statement is:

```
select  e1.Name, e1.DeptID
from    Employee e1
where   e1.Salary = (select  max(e2.Salary)
                     from    Employee e2
                     where   e1.DeptID = e2.DeptID).
```

Depending on the quality of the optimizer there are two possibilities how this query can be evaluated by a current DBMS. The first possibility is to actually execute the nested query once for each tuple from *Employee*. This is obviously extremely slow, regardless if there are indices on *Employee* or not. The second possibility is to evaluate the query in two steps similarly to the algorithm used in the last section. The first step is to use GAgg to group *Employee* by *DeptID* and to calculate *max(Salary)* for each group. The second step is to join the result of the GAgg operator with *Employee* on *DeptID* and *Salary*.

The general semantic query pattern is similar to the pattern in the last section. It selects

"all tuples exhibiting a maximum value for a given expression and a given group."

Again, minimization can be handled analogously, so we call this pattern the *local extremum pattern*.

4.2 Operator definition

We define the $\Gamma^{\max}$-operator as follows:

Let $r(R)$ be a relation, X an ordered set of atomic values and $exp : r \to X$ an expression that should be maximized. We define

$$\begin{aligned}\Gamma^{\max}_{A;exp}(r) \quad := \{ \quad & t \in r : \\ & exp(t) = \max(exp(\{s \in r : s(A) = t(A)\})) \quad \}\end{aligned}$$

The $\Gamma^{\max}$-operator calculates for every group defined by the attributes in A the set of tuples that within their group maximize the value of the expression *exp*. Hence, the operator definition directly reflects the semantics of the local extremum pattern.

4.3 Operator implementation

The implementation for this operator is an extension of the implementation of the Max-operator. Instead of one global set of maximal tuples this implementation uses one set of maximal tuples per group. So each tuple has to be associated to a group before it is processed in the same way as for the Max-operator. The resulting implementation is:

```
GAMMAMAX(r,exp,A)       /* relation r, expression exp to be     */
                        /* maximized, A set of grouping columns */
foreach t in r
do
  if a group for t exists
  then
    M := set of maximal elements for the group of t
    choose m from M
    if exp(t) > exp(m)
    then
      M := emptyset
      add t to M
    else if exp(t) = exp(m)
      add t to M
  else
    create a new group with the set M
    M := emptyset
    add t to M
done
return the sets of all groups
```

A less detailed version of this implemenation was already described by Chatziantoniou and Ross in [CR96]. However, they gave it as an example of a further improvement of their implementation and they did not consider overflow handling.

To handle overflow for this operator we have to use two destaging levels. First, we can also use overflow flags to dynamically revert to the conventional plan. It is advisable to use one overflow flag per group, so only those groups for which an overflow actually occurred have to be joined. However if the number of groups is too large to be kept in main memory we have to destage another level and revert to hybrid hashing [GBC98].

4.4 Comparison of query execution plans

We do not discuss the first described alternative, a nested execution, because it needs $|R_I| + 1$ scans of R_I and therefore shows unacceptable performance for almost every size of R_I. The conventional plan Γ^{max}-operator

$$\begin{tikzcd}
 & \bowtie & & \Gamma^{\mathrm{Max}} \\
\Gamma \arrow[ur] & & & \\
 & R_I \arrow[uu] \arrow[ul] & & R_I \arrow[uuu]
\end{tikzcd}$$

Figure 2: Execution plans for the (logical) Γ^{max}-operator

remaining two execution plans are shown in figure 2. The conventional plan consists of a Γ- and a join-operator. As already noted for the "2-step-algorithm" in the last section, this conventional plan also accesses each tuple in R_I twice, whereas the Γ^{max}-operator needs to access the tuples a second time only if the result does not fit into main memory. As the search for a group does not add much complexity to the calculation of the maximum, the time used for the production of the tuples of R_I still takes the predominant part of the total execution time. Therefore we can achieve the same speedup as for the *Max*-operator (50%), but it is less likely to happen as the probability that the result fits into main memory degrades. However, even in the worst case, there won't be a loss compared to the conventional plan as the Γ^{max} operator is able to gracefully degrade to the conventional plan.

Remark on indices Using indexes here, is not as straightforward as in the previous case. Before we discuss a plan exploiting an index, we consider an alternative implementation of the Γ^{max} operator. Assume that the relation is sorted on both, the grouping attributes and the attributes which are to be maximized. Then, an efficient implementation of the Γ^{max} operator could simply iterate through the sorted (intermediate) relation and select the first tuples that exhibit the maximum possible value within each group (determined solely by the grouping attributes). Note that this implementation of Γ^{max} does not need any intermediate buffer. The main overhead is sorting the relation (see table 2). However, if we have a clustered multi-attribute index on the grouping attributes and the attributes to be maximized, then sorting can be replaced by an index scan. If only the grouping attributes are indexed (again clustered), then a simpler sort that only performs sorting within a single group can replace the original sort.

Remark on optimization If the query exhibits the local extremum pattern, the situation is similar to the situation of a query exhibiting the global extremum pattern. So we can introduce the $\Gamma^{\max}$-operator during rewrite as well, also without enlarging the optimizer's search space.

4.5 Experimental Result

To show the performance of the $\Gamma^{\max}$-operator we modified the query of section 3 to produce the clerk together with the amount for the order with the highest turnover *per year*.

```
select o1.O_Clerk, year(o1.O_Orderdate) as year, o1.O_TotalPrice
from   Order o1
where  o1.O_TotalPrice = (select max(o2.O_TotalPrice)
                          from   Order o2
                          where  year(o1.O_Orderdate) = year(o2.O_Orderdate))
```

We executed this query three times, once with a conventional plan, once using the hash-implementation of the $\Gamma^{\max}$-operator and once using a sort-implementation of the $\Gamma^{\max}$-operator. Concerning the comparison of the conventional plan with the hash-implementation of the $\Gamma^{\max}$-operator, the results in table 2 show two things. First, the expected improvement of 50% can actually be achieved and second, the added computational complexity that was caused by the grouping is less than 25% and therefore does not impact on the total running time (compare with table 1). Concerning the evaluation of the sort-implementation of the $\Gamma^{\max}$-operator there also two facts worth mentioning. First this implementation is much slower even than the conventional plan, and second the higher cost is mainly due to the cost of sorting. As we already gave the system enough main memory to perform the whole sorting operation in main memory, it seems that the use of sorting is only reasonable, if no sort operation is necessary (e.g. because we can get sorted tuples from a clustered index).

5 The $\Gamma^{\text{add-in}}$-operator

5.1 Semantic query pattern

Example The introductory query for our third pattern is:

plan	total time	CPU time
conventional plan	39 s	10.9 s
$\Gamma^{\max}$-operator	19 s	7.4 s
$\Gamma^{\max}$ with sorting	80 s	67 s
just sorting	78 s	65 s

Table 2: Execution times for the $\Gamma^{\max}$-operator

For each department, retrieve the average salary of the department.

In contrast to the last section we do not only want the *DeptID* but also the name. With

Employee(EmpID, Name, Salary, DeptID) and *Department(DeptID, Name)*

the query reads in SQL:

```
select     d.Name, d.DeptID, avg(e.Salary)
from       Department d, Employee e
where      d.DeptID = e.DeptID
group by   d.Name, d.DeptID
```

As was already noted by Yan and Larson [YL94, YL95] and several other authors (for example Chaudhuri and Shim [CS94] and Gupta, Harinarayan and Quass [GHQ95]) the execution time for this query can be significantly reduced by grouping *Employee* by *DeptID* first and joining the result of the aggregation with *Department* in the second step, in other words by pushing group-by in front of the join. When using hash-based operators for grouping and joining the following happens: we first build a hash table for grouping with *e.DeptID* as hash key and then we build another hash table for joining with *e.DeptID* as hash key. So we are building two hash tables with identical content and identical structure. We can obviously do better by using just one hash table for both tasks. Graefe, Bunker and Cooper also reused hash tables for "hash teams" [GBC98]. However in their presentation it seems that for a hash team consisting of a join and a grouping-operator the grouping-operator has to be executed last. Furthermore they require the grouping columns and the join columns to be identical which is not necessary for our $\Gamma^{\text{add-in}}$-operator. More specifically, we only require the existence of at least a single common column.

The general problem we are looking at here are queries that "allow pushing the grouping operation and have common grouping and join columns".

We call this the *common grouping and join columns pattern.*

5.2 Operator definition

We propose to introduce the $\Gamma^{\text{add-in}}$-operator to be used for such queries. It is defined as follows:

Let $r(R), q(Q), p(P)$ be relations and $agg : \mathcal{P}(r) \to p$ be a vector of scalar aggregate functions, then we define

$$\begin{array}{rcl} \Gamma^{\text{add-in}}_{A;agg;B}(r,q) & := & \Gamma_{A;agg}(r) \bowtie_B q \\ & = & \{t(A) \circ a \circ u : t \in r \wedge \\ & & a = agg(\{s \in r : s(A) = t(A)\}) \wedge \\ & & u \in q \wedge (t(A) \circ a)(B) = u(B)\}. \end{array}$$

While this definition is quite general, the desired efficient implementation is only possible if $A \cap B \neq \emptyset$. If this condition holds a hash table using $A \cap B$ as hash key can be used for grouping and join. This is obviously a strict condition but it covers a number of interesting queries, as in Data-Warehouse-applications both joins and groupings on foreign keys are quite common (for example in the TPC-D-benchmark [TPC95]).

5.3 Operator implementation

A simple implementation of this operator could look like this:

```
GAMMAADDIN(r, s, A, agg, B) /* relation r is grouped by the set of */
                            /* columns A and aggregation function  */
                            /* agg is applied, then the result is  */
                            /* joined with relation s on the set   */
                            /* of columns B                        */
foreach t in r
do
  if a group for t exists
  then
    add t to this group
  else
    create a new group
    initialize the new group with values from t
done
finalize all groups
foreach t in s
do
  if a group that matches t exists
  then
    join t with this group
    add the joined tuples to the result
done
```

In the first loop the search for a group is done using a hash table whose hash value is calculated using only the columns from $A \cap B$. The comparison for group equality however uses all columns from A. After the first loop the groups are finalized, which is necessary for some aggregate functions like for example *avg*. In the second loop the search is done in the same hash table—again using only the columns from $A \cap B$ to calculate the hash value—but now all columns from B are used in the comparison for join equality.

The problem of overflow handling is orthogonal to the principle of the implementation and can therefore be solved using the usual mechanisms like hybrid hashing. The important improvement is that the tuples of the second (ungrouped) input relation are probed in the hash table that was built in the grouping step.

5.4 Comparison of query execution plans

The three possible plans for the implementation of the $\Gamma^{\text{add-in}}$-operator are shown in figure 3. The traditional plan joins R_I and R_J before grouping. Pushing the grouping operation in front of the join usually reduces the size of one join input and therefore reduces the time needed to process this operation [YL94]. Especially if R_I is too large to be kept in main memory this can also lead to a reduction in I/O-costs. Two additional benefits can be achieved by

$$\begin{tikzcd}
\text{traditional plan} & & \text{pushed grouping} & & \Gamma^{\text{add-in}} & \\
& \Gamma & & \bowtie & & \Gamma^{\text{AddIn}} \\
& \bowtie \arrow[u] & \Gamma \arrow[ur] & & & \\
R_I \arrow[ur] & R_J \arrow[u] & R_I \arrow[u] & R_J \arrow[uu] & R_I \arrow[uur] & R_J \arrow[uu]
\end{tikzcd}$$

Figure 3: Execution plans for the (logical) $\Gamma^{\text{add-in}}$-operator

using the $\Gamma^{\text{add-in}}$-operator. At first there is no need to copy the tuples from one hash table to another, so some CPU-cost can be saved. The second and more important benefit is that during the execution only half of the memory compared to the use of conventional operators is needed. Therefore it is also possible to save on I/O-costs if hybrid hashing is used for the implementation.

Remark on optimization The optimization of queries that can benefit from the $\Gamma^{\text{add-in}}$-operator is a little bit different from the first two cases. As the operator can be seen as an improved implementation of the combination of a pushed grouping and a join, our optimizer replaces these combinations by the $\Gamma^{\text{add-in}}$-operator in a second rewrite-phase. This second rewrite-phase is performed *after* alternative plans have been generated and an optimal plan has been chosen. At this point we are sure that the use of the $\Gamma^{\text{add-in}}$-operator improves performance. As in the previous two cases the size of the optimizer's search space does not increase by the introduction of this operator.

5.5 Experimental Result

To show the performance improvements due to the use of the $\Gamma^{\text{add-in}}$-operator we used this query:

```
select     P_Partkey, P_Mfgr, P_Brand, P_Type, P_Retailprice,
           avg(PS_Supplycost) as avg_supplycost
from       Part, Partsupp
where      P_Partkey = PS_Partkey
group by   P_Partkey, P_Mfgr, P_Brand, P_Type, P_Retailprice
having     avg(PS_Supplycost) > 0.9 * P_Retailprice
```

plan	total time	CPU time	memory for hash tables
traditional plan	20 s	10.4 s	21.6 MB
pushed grouping	20 s	6.4 s	16.3 MB
$\Gamma^{\text{add-in}}$	20 s	5.4 s	10.7 MB

Table 3: Execution times for the $\Gamma^{\text{add-in}}$-operator

It returns all parts for which the average supply cost accounts for more than 90% of the retail price. Due to the fact that the system was I/O-bound during the execution of all three plans we can see no improvement in the total running times in table 3. However there is a substantial reduction of CPU-cost (48% compared to the traditional plan and 16% compared to the pushed grouping) and memory usage (51% compared to the traditional plan and 34% compared to the pushed grouping). The reason why the pushed grouping needs less memory than the traditional plan is, that an additional projection became possible.

6 Conclusion

We introduced the notion of semantic query pattern and identified three of them in the realm of grouping and aggregation. For these patterns we derived specifically tailored operators and gave implementations of them. We elaborated on the reasons why these implementations are an improvement over today's implementations and verified the correctness of our claims in practice using our experimental database management system AODB.

The main result however is, that improvements in the efficient evaluation of queries can not only be achieved by new implementations of well known algebraic operators—as it has been done until now—but also by extending the algebra. This approach offers a new optimization potential. Without realizing it at that time we already exploited this potential with the Diag-Join-operator, that gains its efficiency by exploiting time-of-creation ordering and by restricting itself to 1:n-relationships [HWM98].

We hope that future research will identify more cases that are special enough to allow for improvement through new operators and yet general enough to justify the inclusion of these operators into a database management system.

References

[CG85] Stefano Ceri and Georg Gottlob. Translating SQL into relational algebra: Optimization, semantics, and equivalence of SQL queries. *IEEE Transactions on Software Engineering (TSE)*, 11(5):324–344, April 1985.

[CM93] Sophie Cluet and Guido Moerkotte. Nested queries in object bases. In *Proceedings of the Fourth International Workshop on Database Programming Languages - Object Models and Languages*, pages 226–242, New York City, NY, USA, 1993.

[CR96] Damianos Chatziantoniou and Kenneth A. Ross. Querying multiple features of groups in relational databases. In *Proceedings of the Conference on Very Large Data Bases (VLDB)*, pages 295–306, Bombay, India, September 1996.

[CS94] Surajit Chaudhuri and Kyuseok Shim. Including group-by in query optimization. In *Proceedings of the Conference on Very Large Data Bases (VLDB)*, pages 354–366, Santiago, Chile, September 1994.

[Day87] Umeshwar Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *Proceedings of the Conference on Very Large Data Bases (VLDB)*, pages 197–208, Brighton, United Kingdom, September 1987.

[Eps79] R. Epstein. Techniques for processing of aggregates in relational database systems. UCB/ERL Memorandum M79/8, Univ. of California at Berkeley, February 1979.

[GBC98] Goetz Graefe, Ross Bunker, and Shaun Cooper. Hash joins and hash teams in Microsoft SQL Server. In VLDB '98 [VLD98], pages 86–97.

[GHQ95] Ashish Gupta, Venky Harinarayan, and Dallan Quass. Aggregate-query processing in data warehousing environments. In VLDB '95 [VLD95], pages 358–369.

[Gra93] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[HWM98] Sven Helmer, Till Westmann, and Guido Moerkotte. Diag-join: An opportunistic join algorithm for 1:n relationships. In VLDB '98 [VLD98], pages 98–109.

[Klu82] Anthony Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *Journal of the ACM*, 29(3):699–717, July 1982.

[Mai83] David Maier. *The theory of relational databases.* Computer Science Press, Rockville, MD, USA, 1983.

[SN95] Ambuj Shatdal and Jeffrey F. Naughton. Adaptive parallel aggregation algorithms. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 104–114, San Jose, CA, USA, June 1995.

[TPC95] Transaction Processing Performance Council TPC. TPC benchmark D (decision support). Standard Specification 1.0, Transaction Processing Performance Council (TPC), May 1995. `http://www.tpc.org/`.

[VLD95] *Proceedings of the Conference on Very Large Data Bases (VLDB)*, Zürich, Switzerland, September 1995.

[VLD98] *Proceedings of the Conference on Very Large Data Bases (VLDB)*, New York, NY, USA, August 1998.

[YL94] Weipeng P. Yan and Per-Åke Larson. Performing group-by before join. In *Proceedings IEEE Conference on Data Engineering*, pages 89–100, Houston, TX, 1994.

[YL95] Weipeng P. Yan and Per-Åke Larson. Eager aggregation and lazy aggregation. In VLDB '95 [VLD95], pages 345–357.