A Framework for Reasoning about Share Equivalence and Its Integration into a Plan Generator

Thomas Neumann #1, Guido Moerkotte *2

Max-Planck Institute for Informatics, Saarbrücken, Germany

1 neumann@mpi-inf.mpg.de

* University of Mannheim, Mannheim, Germany

2 moerkotte@informatik.uni-mannheim.de

Abstract: Very recently, Cao et al. presented the MAPLE approach, which accelerates queries with multiple instances of the same relation by sharing their scan operator. The principal idea is to derive, in a first phase, a non-shared tree-shaped plan via a traditional plan generator. In a second phase, common instances of a scan are detected and shared by turning the operator tree into an operator DAG (directed acyclic graph).

The limits of their approach are obvious. (1) Sharing more than scans is often possible and can lead to considerable performance benefits. (2) As sharing influences plan costs, a separation of the optimization into two phases comprises the danger of missing the optimal plan, since the first optimization phase does not know about sharing.

We remedy both points by introducing a general framework for reasoning about sharing: plans can be shared whenever they are *share equivalent* and not only if they are scans of the same relation. Second, we sketch how this framework can be integrated into a plan generator, which then constructs optimal DAG-structured plans.

1 Introduction

Standard query evaluation relies on tree-structured algebraic expressions which are generated by the plan generator and then evaluated by the query execution engine [Lor74]. Conceptually, the algebra consists of operators working on sets or bags. On the implementation side, they take one or more tuple (object) streams as input and produce a single output stream. The tree-structure thereby guarantees that every operator – except for the root – has exactly one consumer of its output. This flexible concept allows a nearly arbitrary combination of operators and highly efficient implementations.

However, this model has several limitations. Consider, e.g., the following SQL query:

```
select ckey
from   customer, order
where  ckey=ocustomer
group by ckey
having sum(price) = (select max(total)
   from (select ckey, sum(price) as total
         from   customer, order
         where  ckey=ocustomer
         group  by ckey))
```

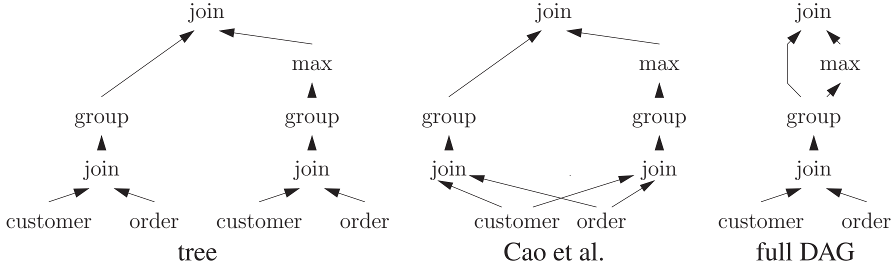

Figure 1: Example plans

This query leads to a plan like the one at the left of Fig. 1. We observe that (1) both relations are accessed twice, (2) the join and (3) the grouping are calculated twice. To (partially) remedy this situation, Cao et al. proposed to share scans of the same relation [CDCT08]. The plan resulting from their approach is shown in the middle of Fig. 1. Still, not all sharing possibilities are exploited. Obviously, only the plan at the right exploits sharing to its full potential.

Another disadvantage of the approach by Cao et al. is that optimization is separated into two phases. In a first phase, a traditional plan generator is used to generate tree-structured plans like the one on the left of Fig. 1. In a second step, this plan is transformed into the one at the middle of Fig. 1. This approach is very nice in the sense that it does not necessitate any modification to existing plan generators: just an additional phase needs to be implemented. However, as always when more than a single optimization phase is used, there is the danger of coming up with a suboptimal plan. In our case, this is due to the fact that adding sharing substantially alters the costs of a plan. As the plan generator is not aware of this cost change, it can come up with (from its perspective) best plan, which exhibits (after sharing) higher costs than the optimal plan.

In this paper, we remedy both disadvantages of the approach by Cao et al. First, we present a general framework that allows us to reason about share equivalences. This will allow us to exploit as much sharing as possible, if this leads to the best plan. Second, we sketch a plan generator that needs a single optimization phase to generate plans with sharing. Using a single optimization phase avoids the generation of suboptimal plans. The downside is that the plan generator has to be adopted to include our framework for reasoning about share equivalence. However, we are strongly convinced that this effort is worth it.

The rest of the paper is organized a follows. Section 2 discusses related work. Section 3 precisely defines the problem. Section 4 describes the theoretical foundation for reasoning about share equivalence. Section 5 sketches the plan generator. The detailed pseudocode and its discussion is given in [NM08]. Section 6 contains the evaluation. Section 7 concludes the paper.

2 Related Work

Let us start the discussion on related work with a general categorization. Papers discussing the generation of DAG-structured query execution plans fall into two broad categories. In the first category, a single optimal tree-structured plan is generated, which is then turned into a DAG by exploiting sharing. This approach is in danger of missing the optimal plan since the tree-structured plan is generated with costs which neglect sharing opportunities.

We call this *post plan generation share detection* (PPGSD). This approach is the most prevailing one in multi-query optimization, e.g. [Sel88]. In the second category, common subexpressions are detected in a first phase before the actual plan generation takes place. The shared subplans are generated independently and then replaced by an artificial single operator. This modified plan is then given to the plan generator. If several sharing alternatives exist, several calls to the plan generator will be made. Although this is a very expensive endeavor due to the (in the worst case exponentially many) calls to the plan generator. Since the partial plans below and above the materialization (temp) operator are generated separately, there is a slight chance that the optimal plan is missed. We term this *loose coupling* between the share detection component and the plan generator. In stark contrast, we present a tightly integrated approach that allows to detect sharing opportunities incrementally during plan generation.

A Starburst paper mentions that DAG-structured query graphs would be nice, but too complex [HFLP89]. A later paper about the DB2 query optimizer [GLSW93] explains that DAG-structured query plans are created when considering views, but this solution materializes results in a temporary relation. Besides, DB2 optimizes the parts above and below the temp operator independently, which can lead to suboptimal plans. Similar techniques are mentioned in [Cha98, GLJ01].

The Volcano query optimizer [Gra90] can generate DAGs by partitioning data and executing an operator in parallel on the different data sets, merging the result afterwards. Similar techniques are described in [Gra93], where algorithms like select, sort, and join are executed in parallel. However, these are very limited forms of DAGs, as they always use data partitioning (i.e., in fact, one tuple is always read by one operator) and sharing is only done within one logical operator.

Another approach using loose coupling is described in [Roy98]. A later publication by the same author [RSSB00] applies loose coupling to multi-query optimization. Another interesting approach is [DSRS01]. It also considers cost-based DAG construction for multi-query optimization. However, its focus is quite different. It concentrates on scheduling problems and uses greedy heuristics instead of constructing the optimal plan. Another loose coupling approach is described in [ZLFL07]. They run the optimizer repeatedly and use view matching mechanisms to construct DAGs by using solutions from the previous runs. Finally, there exist a number of papers that consider special cases of DAGs, e.g. [DSRS01, BBD$^+$04]. While they propose using DAGs, they either produce heuristical solutions or do not support DAGs in the generality of the approach presented here.

3 Problem Definition

Before going into detail, we provide a brief formal overview of the optimization problem we are going to solve in this paper. This section is intended as an illustration to understand the problem and the algorithm. Therefore, we ignore some details like the problem of operator selection here (i.e. the set of operators does not change during query optimization).

We first consider the classical tree optimization problem and then extend it to DAG optimization. Then, we distinguish this from similar DAG-related problems in the literature. Finally, we discuss further DAG-related problems that are not covered in this paper.

3.1 Optimizing Trees

It is the query optimizer's task to find the cheapest query execution plan that is equivalent to the given query. Usually this is done by algebraic optimization, which means the query optimizer tries to find the cheapest algebraic expression (e.g. in relational algebra) that is equivalent to the original query. For simplicity we ignore the distinction between physical and logical algebra in this section. Further, we assume that the query is already given as an algebric expression. As a consequence, we can safely assume that the query optimizer transforms one algebraic expression into another.

Nearly all optimizers use a tree-structured algebra, i.e. the algebraic expression can be written as a tree of operators. The operators themselves form the nodes of the tree, the edges represent the dataflow between the operators. In order to make the distinction between trees and DAGs apparent, we give their definitions. A *tree* is a directed, cycle-free graph $G = (V, E), |E| = |V| - 1$ with a distinguished root node $v_0 \in V$ such that all $v \in V \setminus \{v_0\}$ are reachable from v_0.

Now, given a query as a tree $G = (V, E)$ and a cost function c, the query optimizer tries to find a new tree $G' = (V, E')$ such that $G \equiv G'$ (concerning the produced output) and $c(G')$ is minimal (to distinguish the tree case from the DAG case we will call this equivalence $\equiv_T$). This can be done in different ways, either transformatively by transforming G into G' using known equivalences [Gra94, GM93, Gra95], or constructively by building G' incrementally [Loh88, SAC$^+$79]. The optimal solution is usually found by using dynamic programming or memoization. If the search space is too large then heuristics are used to find good solutions.

An interesting special case is the join ordering problem where V consists only of joins and relations. Here, the following statement holds: any tree G' that satisfies the syntax constraints (binary tree, relations are leafs) is equivalent to G. This makes constructive optimization quite simple. However, this statement does no longer hold for DAGs (see Sec. 4).

3.2 Optimizing DAGs

DAGs are directed acyclic graphs, similar to trees with overlapping (shared) subtrees. Again, the operators form the nodes, and the edges represent the dataflow. In contrast to trees, multiple operators can depend on the same input operator. We are only interested in DAGs that can be used as execution plans, which leads to the following definition. A DAG is a directed, cycle-free graph $G = (V, E)$ with a denoted root node $v_0 \in V$ such that all $v \in V \setminus \{v_0\}$ are reachable from v_0. Note that this is the definition of trees without the condition $|E| = |V| - 1$. Hence, all trees are DAGs.

As stated above, nearly all optimizers use a tree algebra, with expressions that are equivalent to an operator tree. DAGs are no longer equivalent to such expressions. Therefore, the semantics of a DAG has to be defined. To make full use of DAGs, a DAG algebra would be required (and some techniques require such a semantics, e.g. [SPMK95]). However, the normal tree algebra can be lifted to DAGs quite easily: a DAG can be transformed into an equivalent tree by copying all vertices with multiple parents once for each parent. Of course this transformation is not really executed: it only defines the semantics. This trick allows us to lift tree operators to DAG operators, but it does not allow the lifting of

tree-based equivalences (see Sec. 4).

We define the problem of optimizing DAGs as follows. Given the query as a DAG $G = (V, E)$ and a cost function c, the query optimizer has to find any DAG $G' = (V' \subseteq V, E')$ such that $G \equiv G'$ and $c(G')$ is minimal. Thereby, we defined two DAG-structured expressions to be equivalent ($\equiv_D$) if and only if they produce the same output. Note that there are two differences between tree optimization and DAG optimization: First, the result is a DAG (obviously), and second, the result DAG possibly contains fewer operators than the input DAG.

Both differences are important and both are a significant step from trees! The significance of the latter is obvious as it means that the optimizer can choose to eliminate operators by reusing other operators. This requires a kind of reasoning that current query optimizers are not prepared for. Note that this decision is made during optimization time and not beforehand, as several possibilities for operator reuse might exist. Thus, a cost-based decision is required. But also the DAG construction itself is more than just reusing operators: a real DAG algebra (e.g. [SPMK95]) is vastly more expressive and cannot e.g. be simulated by deciding operator reuse beforehand and optimizing trees.

The algorithm described in this work solves the DAG construction problem in its full generality.

By this we mean that (1) it takes an arbitrary query DAG as input (2) constructs the optimal equivalent DAG, and (3) thereby applies equivalences, i.e. a rule-based description of the algebra. This discriminates it from the problems described below, which consider different kinds of DAG generation.

3.3 Problems Not Treated in Depth

In this work, we concentrate on the algebraic optimization of DAG-structured query graphs. However, using DAGs instead of trees produces some new problems in addition to the optimization itself.

One problem area is the execution of DAG-structured query plans. While a tree-structured plan can be executed directly using the iterator model, this is no longer possible for DAGs. One possibility is to materialize the intermediate results used by multiple operators, but this induces additional costs that reduce the benefit of DAGs. Ideally, the reuse of intermediate results should not cause any additional costs, and, in fact, this can be achieved in most cases. As the execution problem is common for all techniques that create DAGs as well as for multi-query optimization, many techniques have been proposed. A nice overview of different techniques can be found in [HSA05]. In addition to this generic approach, there are many special cases like e.g. application in parallel systems [Gra90] and sharing of scans only [CDCT08]. The more general usage of DAGs is considered in [Roy98] and [Neu05], which describe runtime systems for DAGs.

Another problem not discussed in detail is the cost model. This is related to the execution method, as the execution model determines the execution costs. Therefore, no general statement is possible. However, DAGs only make sense if the costs for sharing are low (ideally zero). This means that the input costs of an operator can no longer be determined by adding the costs of its input, as the input may overlap. This problem has not been studied as thoroughly as the execution itself. It is covered in [Neu05].

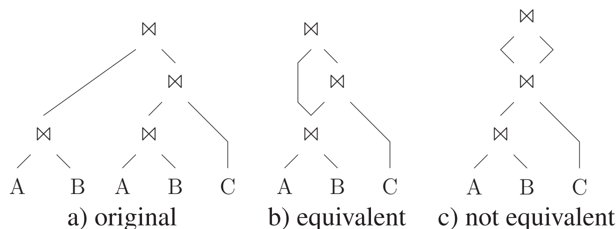

Figure 2: Invalid transformation for DAGs

4 Algebraic Optimization

In this section, we present a theoretical framework for DAG optimization. We first highlight three different aspects that differentiate DAG optimization from tree optimization. Then, we use these observations to formalize the reasoning over DAGs.

4.1 Using Tree Equivalences

Algebraic equivalences are fundamental to any plan generator: It uses them either directly by transforming algebraic expressions into equivalent ones, or indirectly by constructing expressions that are equivalent to the query. For tree-structured query graphs, many equivalences have been proposed (see e.g. [GMUW99, Mai83]). But when reusing them for DAGs, one has to be careful.

When only considering the join ordering problem, the joins are freely reorderable. This means that a join can be placed anywhere where its syntax constraints are satisfied (i.e. the join predicate can be evaluated). However, this is not true when partial results are shared. Let us demonstrate this by the example presented in Fig. 2. The query computes the same logical expression twice. In a) the join $A \bowtie B$ is evaluated twice and can be shared as shown in b). But the join with C may not be executed before the split, as shown in c), which may happen when using a constructive approach to plan generation (e.g. dynamic programming or memoization) that aggressively tries to share relations and only considers syntax constraints. That is, a join can be build into a partial plan as soon as its join predicate is evaluable which in turn only requires that the referenced tables are present. This is the only check performed by a dynamic programming approach to join ordering. Intuitively, it is obvious that c) is not a valid alternative, as it means that $\bowtie C$ is executed on both branches. But in other situations, a similar transformation is valid, e.g. selections can often be applied multiple times without changing the result. As the plan generator must not rely on intuition, we now describe a formal method to reason about DAG transformations. Note that the problem mentioned above does not occur in current query optimization systems, as they treat multiple occurrences of the same relation in a query as distinct relations. But for DAG generation, the query optimizer wants to treat them as identical relations and thus potentially avoid redundant scans.

The reason why the transformation in Fig. 2 is invalid becomes clear if we look at the variable bindings. Let us denote by $A : a$ the successive binding of variable a to members of a set A. In the relational context, a would be bound to all tuples found in relation A. As shown in Fig. 3 a), the original expression consists of two different joins $A \bowtie B$

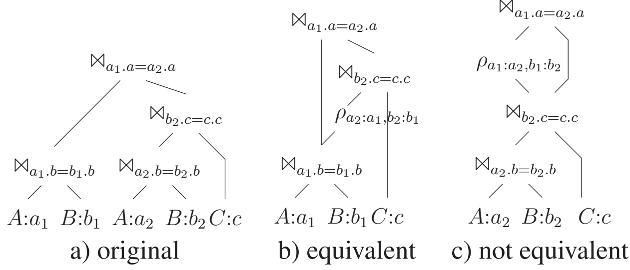

Figure 3: More verbose representation of Fig. 2

with different bindings. The join can be shared in b) by properly applying the renaming operator (ρ) to the output. While a similar rename can be used after the join $\bowtie C$ in c), this still means that the topmost join joins C twice, which is different from the original expression.

This brings us to a rather surprising method to use normal algebra semantics:

> A binary operator must not construct a (*logical*) DAG.

Here, logical means that the same algebra expression is executed on both sides of its input. Further:

> What we do allow are *physical* DAGs, which means that we allow sharing operators to compute multiple logical expressions simultaneously.

As a consequence, we only share operators after proper renames: if an operator has more than one consumer, all but one of these must be rename operators. Thus, we use ρ to pretend that the execution plan is a tree (which it is, logically) instead of the actual DAG.

4.2 Share Equivalence

Before going into more detail, we define whether two algebra expressions are *share equivalent*. This notion will express that *one expression can be computed by using the other expression and renaming the result*. Thus, given two algebra expressions A and B, we define

$$A \equiv_S B \text{ iff } \exists_{\delta_{AB}:\mathcal{A}(A)\to\mathcal{A}(B),\,\delta_{AB}\text{ bijective}}\, \rho_{\delta_{AB}}(A) \equiv_D B.$$

where we denote by $\mathcal{A}(A)$ all the attributes provided in the result of A.

As this condition is difficult to test in general, we use a constructively defined sufficient condition of share equivalence instead. First, two scans of the same relation are share equivalent, since they produce exactly the same output (with different variable bindings):

$$scan_1(R) \equiv_S scan_2(R)$$

Note that in a constructive bottom-up approach, the mapping function $\delta_{A,B}$ is unique. Therefore, we always know how attributes are mapped.

$$\begin{array}{rcll}
A \cup B & \equiv_S & C \cup D & \text{if } A \equiv_S C \wedge B \equiv_S D \\
A \cap B & \equiv_S & C \cap D & \text{if } A \equiv_S C \wedge B \equiv_S D \\
A \setminus B & \equiv_S & C \setminus D & \text{if } A \equiv_S C \wedge B \equiv_S D \\
\Pi_A(B) & \equiv_S & \Pi_C(D) & \text{if } B \equiv_S D \wedge \delta_{B,D}(A) = C \\
\rho_{a \rightarrow b}(A) & \equiv_S & \rho_{c \rightarrow d}(B) & \text{if } A \equiv_S B \wedge \delta_{A,B}(a) = c \wedge \delta_{A,B}(b) = d \\
\chi_{a:f}(A) & \equiv_S & \chi_{b:g}(B) & \text{if } A \equiv_S B \wedge \delta_{A,B}(a) = b \wedge \delta_{A,B}(f) = g \\
\sigma_{a=b}(A) & \equiv_S & \sigma_{c=d}(B) & \text{if } A \equiv_S B \wedge \delta_{A,B}(a) = c \wedge \delta_{A,B}(b) = d \\
A \times B & \equiv_S & C \times D & \text{if } A \equiv_S C \wedge B \equiv_S D \\
A \bowtie_{a=b}(B) & \equiv_S & C \bowtie_{c=d}(D) & \text{if } A \equiv_S C \wedge B \equiv_S D \wedge \delta_{A,C}(a) = c \wedge \delta_{B,D}(b) = d \\
\Gamma_{A;a:f}(B) & \equiv_S & \Gamma_{C;b:g}(D) & \text{if } B \equiv_S D \wedge \delta_{B,D}(A) = C \wedge \delta_{B,D}(a) = b \wedge \delta_{B,D}(f) = g
\end{array}$$

Figure 4: Definition of share equivalence for common operators

Other operators are share equivalent if their input is share equivalent and their predicates are equivalent after applying the mapping function. The conditions for share equivalence for common operators are summarized in Fig. 4. They are much easier to check, especially when constructing plans bottom-up (as this follows the definition).

Note that share equivalence as calculated by the tests above is orthogonal to normal expression equivalence. For example, $\sigma_1(\sigma_2(R))$ and $\sigma_2(\sigma_1(R))$ are equivalent but *not* derivable as share equivalent by testing the sufficient conditions. This will not pose any problems to the plan generator, as it will consider both orderings. On the other hand, $scan_1(R)$ and $scan_2(R)$ are share equivalent, but not equivalent, as they may produce different attribute bindings.

Share equivalence is only used to detect if exactly the same operations occur twice in a plan and, therefore, cause costs only once. Logical equivalence of expressions is handled by the plan generator anyway, it is not DAG-specific.

Using this notion, the problem in Fig. 2 becomes clear: In part b), the expression $A \bowtie B$ is shared, which is ok, as $(A \bowtie B) \equiv_S (A \bowtie B)$. But in part c), the top-most join tries to also share the join with C, which is not ok, as $(A \bowtie B) \not\equiv_S ((A \bowtie B) \bowtie C)$. Note that while this might look obvious, it is not when e.g. constructing plans bottom up and assuming freely reorderable joins, as discussed in Section 3.1.

4.3 Optimizing DAGs

The easiest way to reuse existing equivalences is to hide the DAG structure completely: During query optimization, the query graph is represented as a tree, and only when determining the costs of a tree the share equivalent parts are determined and the costs adjusted accordingly. Only after the query optimization phase the query is converted into a DAG by merging share equivalent parts. While this reduces the changes required for DAG support to a minimum, it makes the cost function very expensive. Besides, if the query graph is already DAG-structured (e.g. for bypass plans), the corresponding tree-structured representation is much larger (e.g. exponentially for bypass plans), enlarging the search space accordingly.

A more general optimization can be done by sharing operators via rename operators. While somewhat difficult to do in a transformation-based plan generator, for a constructive plan generator it is easy to choose a share equivalent alternative and add a rename operator

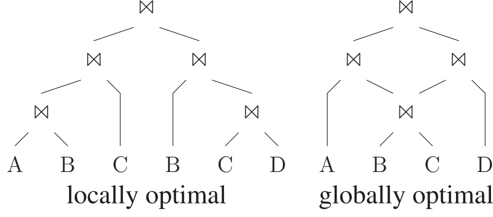

Figure 5: Possible non-optimal substructure for DAGs

as needed. Logically, the resulting plans behave as if the version without renaming was executed (i.e. as if the plan was a tree instead of a DAG). Therefore, the regular algebraic equivalences can be used for optimization. This issue will come up again when we discuss the plan generator.

4.4 (No) Optimal Substructure

Optimization techniques like dynamic programming and memoization rely on an optimal substructure of a problem (neglecting physical properties like sortedness or groupedness for a moment). This means that Bellmann's optimality principle holds and, thus, the optimal solution can be found by combining optimal solutions for subproblems. This is true for generating optimal tree-structured query graphs, but is not necessarily true for generating optimal DAGs. To see this, consider Fig. 5, which shows two plans for $A \bowtie B \bowtie C \bowtie B \bowtie C \bowtie D$. The plan on the left-hand side was constructed bottom-up, relying on the optimal substructure. Thus, $A \bowtie B \bowtie C$ was optimized, resulting in the optimal join ordering $(A \bowtie B) \bowtie C$. Besides, the optimal solution for $B \bowtie C \bowtie D$ was constructed, resulting in $B \bowtie (C \bowtie D)$. But when these two optimal partial solutions are combined, no partial join result can be reused. When choosing the suboptimal partial solutions $A \bowtie (B \bowtie C)$ and $(B \bowtie C) \bowtie D$, the expression $B \bowtie C$ can be shared, which might result in a better plan. Therefore, the optimal DAG cannot be constructed by just combining optimal partial solutions. Our approach avoids this problem by keeping track of sharing opportunities and considering them while pruning otherwise dominated plans.

4.5 Reasoning over DAGs

After looking at various aspects of DAG generation, we now give a formal model to reason about DAGs. More precisely, we specify when two DAGs are equivalent and when one DAG dominates another. Both operations are crucial for algebraic optimization.

As we want to lift tree equivalences to DAGs, we need some preliminaries: We name the equivalences for trees $\equiv_T$ and assume that the following conditions hold for all operators θ, that is the equivalence can be checked on a per operator basis.

$$
\begin{aligned}
t \equiv_T t' \quad &\Rightarrow \quad \theta(t) \equiv_T \theta(t') \\
t_1 \equiv_T t_1' \wedge t_2 \equiv_T t_2' \quad &\Rightarrow \quad (t_1 \theta t_2) \equiv_T (t_1' \theta t_2')
\end{aligned}
$$

These conditions are a fundamental requirement of constructive plan generation, but as

seen in Sec. 4.1, they do no longer hold for DAGs in general. However, they hold for DAGs (V, E) that logically are a tree (LT), i.e. that have non-overlapping input for all operators. For an expression e let $\mathcal{A}(e)$ denote the set of attributes produced by e. We then define $LT((V, E))$ for a DAG (V, E) as follows:

$$LT((V,E)) \quad \text{iff} \quad \forall v, v_1, v_2 \in V, (v, v_1) \in E, (v, v_2) \in E : \\ \mathcal{A}(v_1) \cap \mathcal{A}(v_2) = \emptyset$$

Note that this definition implies that all sharing of intermediate results must be done by renaming attributes.

Using this definition, we can now lift tree equivalences $\equiv_T$ to DAG equivalences $\equiv_D$ for DAGs d and d': We reuse the tree equivalences directly, but thereby must make sure that the input indeed behaves like a tree, as the equivalences were originally only defined on trees:

$$d \equiv_T d' \wedge LT(d) \wedge LT(d') \quad \Rightarrow \quad d \equiv_D d'$$

Note that the condition $LT(d) \Rightarrow LT(d' \subseteq d)$ holds, therefore partial DAGs that violate the logical tree constraint should be discarded immediately. As a consequence, tests for $LT(d)$ are required only when adding binary (or n-ary) operators, as unary operators cannot produce a new violation.

While lifting the tree equivalences to DAGs is important, they are not enough for DAG optimization, as they only create trees. DAGs can be created by using the notion of share equivalence defined above: If two DAGs d and d' are share equivalent, the two DAGs become equivalent by renaming the result suitably:

$$d \equiv_S d' \quad \Rightarrow \quad d \equiv_D \rho_{\mathcal{A}(d') \rightarrow \mathcal{A}(d)} d'$$

While these two implications are not exhaustive, lifting the tree equivalences to DAG equivalences and reusing intermediate results already allows for a wide range of DAG plans. Additional equivalences can be derived e.g. from [MFPR90, SPMK95].

In addition to checking if two plans are equivalent, the query optimizer has to decide which of two equivalent plans is better. Better usually means cheaper, according to a cost function. But sometimes two plans are incomparable, e.g. because one plan satisfies other ordering properties than the other. This is true for both, trees and DAGs. Here, we only look at the costs and DAG-specific limitations to keep the definitions short.

As shown in Sec. 4.4, DAGs cannot be compared by just looking at the costs, as one DAG might allow for more sharing of intermediate results than the other. To identify these plans, we require a labeling function that marks sharing opportunities. Here, we specify only the characteristics, see Section 5.3 for an explicit definition.

The DAGs are labeled using a function S. Its codomain is required to be partially ordered. We require that S assigns the same label to share equivalent DAGs. Further the partial ordering between labels must express the fact that one DAG provides more sharing opportunities than another. Thus, for two DAGs $d_1 = (V_1, E_1)$ and $d_2 = (V_2, E_2)$ we require

the following formal properties for S:

$$S(d_1) = S(d_2) \quad \text{iff} \quad d_1 \equiv_S d_2$$
$$S(d_1) < S(d_2) \quad \text{iff} \quad \exists d'_2 \subset_D d_2 : d_1 \equiv_S d'_2$$

Note that that $d'_2 = (V'_2, E'_2) \subset_D d_2$ iff $V'_2 \subset V_2 \wedge E'_2 = E_{2|V'_2} \wedge d_2$ is a DAG).

Now a plan dominates another equivalent plan if it is cheaper and offers at least the same sharing opportunities:

$$\begin{aligned} & d_1 \equiv_D d_2 \wedge costs(d_1) < costs(d_2) \wedge S(d_2) \leq S(d_1) \\ \Rightarrow \quad & d_1 \text{ dominates } d_2. \end{aligned}$$

Note that the characterization of S given above is overly conservative: one plan might offer more sharing opportunities than another, but these could be irrelevant for the current query. This is similar to order optimization, where the query optimizer only considers differences in interesting orderings [SAC+79, SSM96]. The algorithm presented in Section 5.3 improves the labeling by checking which of the operators could produce interesting shared intermediate results for the given query. As only whole subgraphs can be shared, the labeling function stops assigning new labels if any operator in the partial DAG cannot be reused. This greatly improves plan pruning, as more plans become comparable.

5 Plan Generator Skeleton

5.1 Overview

The main motivation for generating DAG structured execution plans is the ability to share intermediate results. Obviously, the fewer sharing opportunities are missed, the better. Hence, it is important to allow for a very aggressive sharing during plan generation.

In order to share intermediate results, the plan generator has to detect that partial plans produce the same (or rather equivalent) output. This is a problem for a rule-based plan generator, as the set of operators cannot be hard-coded and tests for equivalence are expensive. We overcome this problem by using a very abstract description of logical plan properties. Instead of using a complex property vector like, e.g., [Loh88], the plan generator uses a set of *logical properties*. The semantics of these logical properties is only known to the rules. They guarantee that two plans are equivalent if they have the same logical properties. A suitable representation is e.g. a bit vector with one bit for each property.

We assume that the logical properties are set properly by the optimization rules. This is all the plan generator has to know about the logical properties! However, to make things more concrete for the reader, we discuss some possible logical properties here. The most important ones are ”relation / attribute available” and ”operator applied”. In fact, these properties are already sufficient when optimizing only selections and joins as both are freely reorderable. We discuss more complex properties in Section 5.2.

As the logical properties describe equivalent plans, they can be used to partition the search space. This allows for a very generic formulation of a plan generator:

PLANGEN($goal$)
1 $bestPlan \leftarrow$ NIL
2 **for each** optimization rule r
3 **do if** r can produce a logical property in $goal$
4 **then** $rem \leftarrow goal \setminus \{p|p$ is produced by $r\}$
5 $part \leftarrow$ PLANGEN(rem)
6 $p \leftarrow$ BUILDPLAN($r, part$)
7 **if** $bestPlan =$ NIL or p is cheaper
8 **then** $bestPlan \leftarrow p$
9 **return** $bestPlan$

This algorithm performs a top-down exploration of the search space. Starting with the complete problem (i.e. finding a plan that satisfies the query), it asks the optimization rules for partial solutions and solves the remaining (sub-) problem(s) recursively. For SPJ queries, this implies splitting the big join into smaller join problems (and selections) and solving them recursively. This algorithm is highly simplified, it only supports unary operators, does not perform memoization etc. Adding support for binary operators is easy but lengthy to describe, see [NM08] for a detailed discussion. Other missing and essential details will be discussed in Section 5.4. Let us summarize the main points of the above approach:

1. The search space is represented by abstract logical properties,
2. the plan generator recursively solves subproblems, which are fully specified by property combinations, and
3. property combinations are split into smaller problems by the optimization rules.

This approach has several advantages. First, it is very extensible, as the plan generator only reasons about abstract logical properties: the actual semantics is hidden in the rules. Second, it constructs DAGs naturally: if the same subproblem (i.e. the same logical property combination) occurs twice in a plan, the plan generator produces only one plan and thus creates a DAG. In reality, this is somewhat more complex, as queries usually do not contain exactly the same subproblem twice (with the same variable bindings etc.), but as we will see in Section 5.4, the notion of share equivalence can be used to identify sharable subplans.

5.2 Search Space Organization

An unusual feature of our plan generator is that it operates on an abstract search space, as the semantics of the logical properties are only known to concrete optimization rules. To give an intuition why this is a plausible concept, we now discuss constructing suitable properties for commonly used queries.

The most essential part of choosing properties is to express the query semantics properly. In particular, two plans must only have the same logical properties when they are equivalent. For the most common and (simplest) type of queries, the selection-projection-join (SPJ) queries using inner joins, it is sufficient to keep track of the available relations (and attributes) and the applied operators. For this kind of queries, two plans are equivalent if they operate on the same set of relations and they have applied the same operators. Therefore using the available attributes/relations and the applied operators as properties is sufficient for this kind of queries. Note that the properties are not only used by the plan

generator but also by the optimization rules, e.g. to check if all attributes required by a selection are available.

For more general queries involving e.g. outer joins, the same properties are sufficient, but the operator dependencies are more complex: Using the extended eligibility list concept from [RLL+01], each operator specifies which other operators have to be applied before it becomes applicable. These operator dependencies allow for handling complex queries with relatively simple properties, and can be extended to express dependent subqueries etc. Note that we do not insist on a "dense" encoding of the search space that only allows for valid plans (which is desirable, but difficult to achieve in the presence of complex operator). Instead, we allow for a slightly sparse encoding and then guarantee that we only explore the valid search space.

Note that these properties are just examples, which are suitable for complex queries but not mandatory otherwise. The plan generator makes only two assumptions about the logical properties: Two plans with the same properties are equivalent and properties can be split to form subproblems. Further, not all information about plans have to be encoded into the logical properties. In our implementation, ordering/grouping properties are handled separately, as the number of orderings/groupings can be very large. We use the data structure described in [NM04] to represent them. This implies that multiple plans with the same logical properties might be relevant during the search, as they could differ in some additional aspects. We will see another example for additional properties in the next section.

5.3 Sharing Properties

Apart from the logical properties used to span the search space, each plan contains a *sharing* bit set to indicate potentially shared operators. It can be considered as the materialization of the labeling function S used in Section 4.5: The partial ordering on S is defined by the subset relation, that is, one plan can only dominate another plan if it offers at least the same sharing opportunities. We now give a constructive approach for computing S.

When considering a set of share equivalent plans, it is sufficient to keep one representative, as the other plans can be constructed by using the representative and adding a rename (note that all share equivalent plans have the same costs). Analogously, the plan generator determines all operators that are share equivalent (more precisely: could produce share equivalent plans if their subproblems had share equivalent solutions) and places them in equivalence classes. As a consequence, two plans can only be share equivalent if their top-most operators are in the same equivalence class, which makes it easier to detect share equivalence. The equivalence classes that contain only a single operator are discarded, as they do not affect plan sharing. For the remaining equivalence classes, one representative is selected and one bit in the *sharing* bit set is assigned to it.

For example, the query in Fig. 5 consists of 11 operators: A, B_1, C_1, B_2, C_2, D, $\bowtie_1$ (A and B_1), $\bowtie_2$ (between B_1 and C_1), $\bowtie_3$ (between B_2 and C_2), $\bowtie_4$ (between C_2 and D) and $\bowtie_5$ (between A and D). Then three equivalence classes with more than one element can be constructed: $B_1 \equiv_S B_2$, $C_1 \equiv_S C_2$ and $\bowtie_2 \equiv_S \bowtie_3$. We assume that the operator with the smallest subscript was chosen as representative for each equivalence class. Then the plan generator would set the *sharing* bit B_1 for the plan B_1, but not for the plan B_2. The plan $A\bowtie_1(B_1\bowtie_2C_1)$ would set sharing bits for B_1, C_1 and $\bowtie_2$, as the subplan can be shared, while the plan $(A\bowtie_1B_1)\bowtie_2C_1$ would only set the bits B_1 and C_1, as the join $\bowtie_2$

cannot be shared here (only whole subgraphs can be shared). The sharing bit set allows the plan generator to detect that the first plan is not dominated by the second, as the first plan allows for more sharing. This solves the problem discussed in Section 4.4.

The equivalence classes are also used for another purpose: When an optimization rule requests a plan with the logical properties produced by an operator, the plan generator first checks if a share equivalent equivalence class representative exists. For example, if a rule requests a plan with B_2, C_2 and $\bowtie_3$, the plan generator first tries to build a plan with B_1, C_1 and $\bowtie_2$, as these are the representatives. If this rewrite is possible (i.e. a plan could be constructed), the plan constructed this way is also considered a possible solution.

In general, the plan generator uses sharing bits to explicitly mark sharing opportunities: whenever a partial plan is built using an equivalence class representative, the corresponding bit is set. In more colloquial words: the plan *offers* to share this operator. Note that it is sufficient to identify the selected representative, as all other operators in the equivalence class can be built by just using the representative and renaming the output. As sharing is only possible for whole subplans, the bit must only be set if the input is also sharable. Given, for example, three selections σ_1, σ_2 and σ_3, with $\sigma_1(R) \equiv_S \sigma_2(R)$. The two operator rules for σ_1 and σ_2 are in the same equivalence class, we assume that σ_1 was selected as representative. Now the plan $\sigma_1(R)$ is marked as "shares σ_1", as it can be used instead of $\sigma_2(R)$. The same is done for $\sigma_3(\sigma_1(R))$, as it can be used instead of $\sigma_3(\sigma_2(R))$. But for the plan $\sigma_1(\sigma_3(R))$ the *sharing* attribute is empty, as σ_1 cannot be shared (since σ_3 cannot be shared). The plans containing σ_2 do not set the *sharing* property, as σ_1 was selected as representative and, therefore, σ_2 is never shared.

Note that the sharing bits are only set when the whole plan can be shared, but the already existing sharing bits are still propagated even if it is no longer possible to share the whole plan. Otherwise a slightly more expensive plan might be pruned early even though parts of it could be reused for other parts of the query.

Explicitly marking the sharing opportunities serves two purposes. First, it is required to guarantee that the plan generator generates the optimal plan, as one plan only dominates another if it is cheaper and offers at least the same sharing opportunities. Second, sharing information is required by the cost model, as it has to identify the places where a DAG is formed (i.e. the input overlaps). This can now be done by checking for overlapping *sharing* properties. It is not sufficient to check if the normal logical properties overlap, as the plans pretend to perform different operations (which they do, logically), but share physical operators.

5.4 Search Space Exploration under Sharing

After describing the general approach for plan generation, we now present the specific steps required to handle shared plans. Unfortunately, the plan generator is not a single, concise algorithm but a set of algorithms that are slightly interwoven with the optimization rules. This is unavoidable, as the plan generator has to be as generic as possible and, therefore, does not understand the semantics of the operators. However, there is still a clear functional separation between the different modules: The *plan generator* itself maintains the partial plans, manages the memoization and organizes the search. The *optimization rules* describe the semantics of the operators and guide the search with their requirements. For the specific query, several optimization rules are *instantiated*, i.e. annotated with the

query specific information like selectivities, operator dependencies etc. Typically each operator in the original query will cause a rule instantiation. Note that we only present a simplified plan generator here to illustrate the approach, in particular the search space navigation. The detailed algorithms are discussed in [NM08].

Within the search space, the plan generator tries to find the cheapest plan satisfying all logical properties required for the final solution. Note that in the following discussion, we ignore performance optimization to make the conceptual structure clearer. In practice, the plan generator prunes plans against known partial solutions, uses heuristics like KBZ [KBZ86] to get upper bounds for costs etc. However, as these are standard techniques, we do not elaborate on them here.

The core of the plan generator itself is surprisingly small and only consists of a single function that finds the cheapest plans with a given set of logical properties. Conceptually this is similar to the top-down optimization strategy known from Volcano [Gra95, GM93]. The search phase is started by requesting the cheapest plan that provides the goal properties.

```
PLANGEN(goal)
 1  plans ← memoizationTable[goal]
 2  if plans already computed
 3     then return plans
 4  plans ← create a new, empty PlanSet
 5  shared ← goal rewritten using representatives
 6  if shared ∩ goal = ∅
 7     then plans ← PLANGEN(shared)
 8  for each r in instantiated rules
 9  do filter ← r.produces ∪ r.required
10      if filter ⊆ goal
11         then sub ← PLANGEN(goal \ r.produced)
12               plans ← plans ∪ {r(p)|p ∈ sub}
13  memoizationTable[goal] ← plans
14  return plans
```

What is happening here is that the plan generator is asked to produce plans with a given set of logical properties. First, it checks the memoization data structure (e.g. a hash table, logical properties→plan set) to see if this was already done before. If not, it creates a new set (initially empty) and stores it in the memoization structure. Then, it checks if the goal can be rewritten to use only representatives from equivalence classes as described in Section 5.3 (if the operators o_1 and o_2 are in the same equivalence class, the logical properties produced by o_2 can be replaced by those produced by o_1). If the rewrite is complete, i.e., the new goal is disjunct from the original goal ($shared \cap goal = \emptyset$), the current problem can be formulated as a new one using only share equivalent operators. This is an application of the DAG equivalence given in Section 4.5. Thus, the plan generator tries to solve the new problem and adds the results to the set of usable plans. Afterwards, it looks at all rule instances, checks if the corresponding filter is a subset of the current goal (i.e. the rule is relevant) and generates new plans using this rule. Note that the lines 8-12 are very simplified and assume unary operators. In practice, the optimizer delegates the search space navigation (here a simple $goal \setminus r.produced$) to the rules.

6 Evaluation

In the previous sections, we discussed several aspects of optimizing DAG-structured query graphs. However, we still have to show three claims: 1) creating DAG-structured query plans is actually beneficial for common queries, 2) sharing base scans as proposed by Cao et al. [CDCT08] is inferior to full DAG support, and 3) the overhead of generating DAG-structured plans is negligible. Therefore, we present several queries for which we create tree-structured and DAG-structured query plans. Both the compile time and the runtime of the resulting plans are compared to see if the overhead for DAGs is worthwhile. For the DAG plans, we created the full DAGs in a single phase optimizer using our DAG reasoning, and the scan sharing plans by a two phase optimizer based upon the tree plans as proposed in [CDCT08]. All experiments were executed on a 2.2 GHz Athlon64 system running Windows XP. The plans were executed using a runtime system that could execute DAGs natively [Neu05].

Each operator (join, group-by etc.) is given 1MB buffer space. This is somewhat unfair against the DAG-structured query plans, as they need fewer operators and could therefore allocate larger buffers. But dynamic buffer sizes would affect the cost model, and the space allocation should probably be a plan generator decision. As this is beyond the scope of this work, we just use a static buffer size here.

As a comparison with the state of the art in commercial database systems, we included results for DB2 9.2.1, which factorizes common subexpressions using materialization of intermediate results. As we were unable to measure the optimization time accurately enough, we only show the total execution time (which includes optimization) for DB2.

6.1 TPC-H

The TPC-H benchmark is a standard benchmark to evaluate relational database systems. It tests ad-hoc queries which result in relatively simple plans allowing for an illustrative comparison between tree- and DAG-structured plans. We used a scale factor 1 database (1GB).

Before looking at some exemplary queries, we would like to mention that queries without sharing opportunities are unaffected by DAG support: The plan generator produces exactly the same plans with and without DAG support, and their compile times are identical. Therefore, it is sufficient to look at queries, which potentially benefit from DAGs.

Query 11

Query 11 is a typical query that benefits from DAG-structured query plans. It determines the most important subset of suppliers' stock in a given country (Germany in the reference query). The available stock is determined by joining `partsupp`, `supplier` and `nation`. As the top fraction is requested, this join is performed twice, once to get the total sum and once to compare each part with the sum. When constructing a DAG, this duplicate work can be avoided. The compile time and runtime characteristics are shown below:

	tree	Scan Sharing	Full DAG	DB2
compilation [ms]	10.5	10.5	10.6	-
execution [ms]	4793	4256	2436	3291

While the compile time is slightly higher when considering DAGs (profiling showed this is due to the checks for share equivalence), the runtime is much smaller. The corresponding plans are shown in Fig. 6: In the tree version, the relations `partsupp`, `supplier` and `nation` are joined twice, once to get the total sum and once to get the sum for each part. In the DAG version, this work can be shared, which nearly halves the execution time. The scan sharing approach from [CDCT08] only slightly better than the tree plan as it still has to join twice. DB2 performs between trees and full DAGs, as it reuses intermediate results but materializes to support multiple reads.

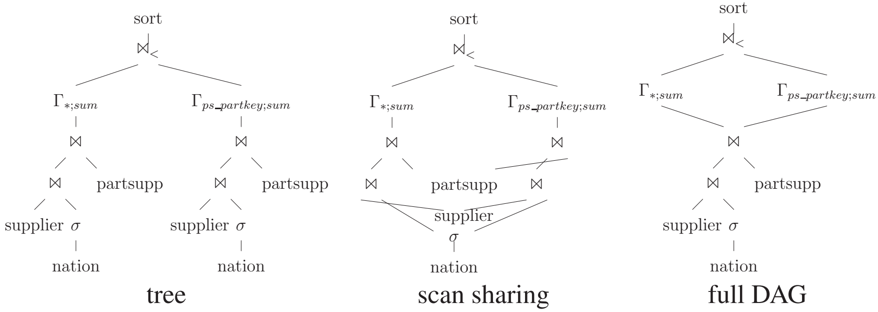

Figure 6: Execution plans for TPC-H Query 11

Query 2

Query 2 selects the supplier with the minimal supply costs within a given region. Structurally, this query is similar to Query 11, as it performs a large join twice, once for the result and once to get the minimum. However, it is more complex, as the nested minimum query depends on the outer query. We assume that the rewrite step unnests the query (by grouping and using a join), but still the nested query lacks a relation used in the outer query, which prevents sharing the whole join. To allow for greater sharing, the relation (and the corresponding predicates) can be re-added by a magic set like transformation [MFPR90]. Here, we consider three alternatives: Normal tree construction, DAG construction and DAG construction with rules for magic set transformation enabled. We omit the sharing approach from [CDCT08] here, as it is nearly identical to the second alternative for this query (one additional (cheap) join). Compile time and runtime are shown below.

	tree	DAG	DAG + magic set	DB2
compilation [ms]	9.3	9.2	9.7	-
execution [ms]	11933	7480	3535	8705

The compile times for tree and DAG are about the same (the DAG is slightly faster, as it can ignore some dominated alternatives), while the magic set variant is about 5% slower due to the increased search space. The runtime behavior of the alternatives is very different. Fig. 7 shows the plans. The tree variant simply calculates the outer query and the nested query independently and joins the result. The DAG variant tries to reuse some intermediate results (reducing the runtime by 37%). It still performs most of the joins in both parts, as the subqueries are not identical. It would be possible to share more by applying the join with *part* later, but this does not pay off due to selectivity of the join. When using the magic set transformation, large parts of the query become equivalent, which results in much greater sharing and also reduced aggregation effort. The consequence is a runtime reduction by 70% compared to the tree variant. DB2 performs better than the tree plan (again due to materializing intermediate results), but worse than both DAG plans.

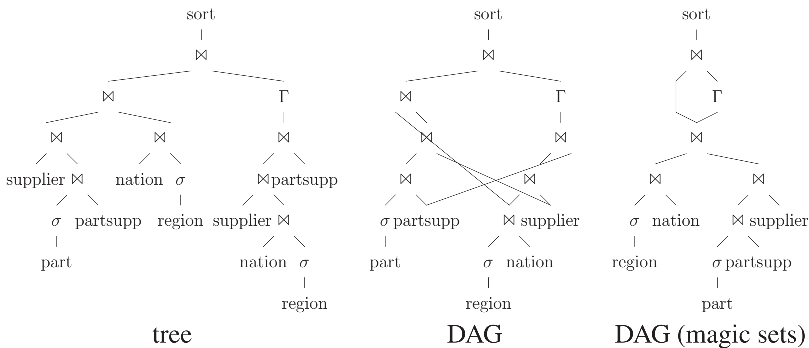

Figure 7: Execution plans for TPC-H Query 2

6.2 Conclusion

The experiments have shown that by considering sharing intermediate results, the compile time is mainly unaffected. It is somewhat affected when additionally magic sets are used (Query 2). Still, the compile time is negligible and clearly dominated by the runtime. The runtime effect of DAG support is non-negligible, as sharing can drastically reduce the runtime of many queries. Further, the real benefit comes from sharing intermediate results. Simply sharing scans as proposed in [CDCT08] improves runtimes a bit, but not nearly as much as full DAG support.

7 Conclusion

We presented the first plan generator which exploits the full potential of shared subplans by being able to explore the complete search space in a fashion that integrates reasoning about sharing and plan generation.

As we have seen, generating DAG-structured query evaluation plans is not easy. To solve the problem, we introduced the novel notion of *share equivalence* and saw that it is orthogonal to the standard notion of equivalence. Rules to reason about share equivalence made this hard to decide notion manageable for the plan generator.

In order to be able to build upon well-known algebraic equivalences, we separated regular equivalence and share equivalence. The former notion is applied only to logical plans which thus remain tree-structured. Reasoning about sharing is moved exclusively to the physical level. The central idea here was to annotate sharing possibilities explicitly and let the plan generator reason about them. To make this more efficient, plans are grouped into equivalence classes where those plans leading to the same sharing possibilities find themselves in the same class. Thereby, the reasoning step was reduced to a simple and efficient rewrite of the optimization goal. The resulting plan generator is the first one that guarantees two important features: (1) all sharing possibilities are detected and considered, and (2) the resulting plan is guaranteed to be optimal. Finally, experiments demonstrated that compared to generating tree-structured plans, the plan generation overhead is negligible, while the gains during query execution are tremendous.

There is plenty of room for future research. Let us mention just two important areas. The first is to develop and explore more optimization techniques that inherently require DAG support. With bypass plans, we have already seen an example of them. The second area where DAGs play a crucial role is stream processing.

Acknowledgment: We thank Simone Seeger for her help preparing the manuscript.

References

[BBD+04] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Dilys Thomas. Operator scheduling in data stream systems. *VLDB J.*, 13(4):333–353, 2004.

[CDCT08] Y. Cao, G. Das, C.-Y. Chan, and K.-L. Tan. Optimizing Complex Queries with Multiple Relation Instances. In *SIGMOD*, pages 525–538, 2008.

[Cha98] Don Chamberlin. *A complete guide to DB2 universal database*. San Francisco, CA, USA, 1998.

[DSRS01] Nilesh N. Dalvi, Sumit K. Sanghai, Prasan Roy, and S. Sudarshan. Pipelining in Multi-Query Optimization. In *PODS*, 2001.

[GLJ01] César A. Galindo-Legaria and Milind Joshi. Orthogonal Optimization of Subqueries and Aggregation. In *SIGMOD*, 2001.

[GLSW93] Peter Gassner, Guy M. Lohman, K. Bernhard Schiefer, and Yun Wang. Query Optimization in the IBM DB2 Family. *IEEE Data Eng. Bull.*, 16(4):4–18, 1993.

[GM93] Goetz Graefe and William J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *ICDE*, pages 209–218, 1993.

[GMUW99] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database System Implementation*. 1999.

[Gra90] Goetz Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. In *SIGMOD*, pages 102–111, 1990.

[Gra93] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[Gra94] Goetz Graefe. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994.

[Gra95] Goetz Graefe. The Cascades Framework for Query Optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[HFLP89] Laura M. Haas, Johann Christoph Freytag, Guy M. Lohman, and Hamid Pirahesh. Extensible Query Processing in Starburst. In *SIGMOD*, pages 377–388, 1989.

[HSA05] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. QPipe: A Simultaneously Pipelined Relational Query Engine. In *SIGMOD*, pages 383–394, 2005.

[KBZ86] Ravi Krishnamurthy, Haran Boral, and Carlo Zaniolo. Optimization of Nonrecursive Queries. In *VLDB'86*, pages 128–137, 1986.

[Loh88] Guy M. Lohman. Grammar-like Functional Rules for Representing Query Optimization Alternatives. In *SIGMOD*, pages 18–27, 1988.

[Lor74] Raymond A. Lorie. XRM - An Extended (N-ary) Relational Memory. *IBM Research Report*, G320-2096, 1974.

[Mai83] David Maier. *The Theory of Relational Databases*. 1983.

[MFPR90] Inderpal Singh Mumick, Sheldon J. Finkelstein, Hamid Pirahesh, and Raghu Ramakrishnan. Magic is Relevant. In *SIGMOD*, pages 247–258, 1990.

[Neu05] Thomas Neumann. *Efficient Generation and Execution of DAG-Structured Query Graphs*. PhD thesis, Universität Mannheim, 2005.

[NM04] Thomas Neumann and Guido Moerkotte. An Efficient Framework for Order Optimization. In *ICDE*, pages 461–472, 2004.

[NM08] Thomas Neumann and Gudio Moerkotte. Single Phase Construction of Optimal DAG-structured QEPs. Research Report MPI-I-2008-5-002, Max-Planck-Institut für Informatik, 2008.

[RLL+01] Jun Rao, Bruce G. Lindsay, Guy M. Lohman, Hamid Pirahesh, and David E. Simmen. Using EELs, a Practical Approach to Outerjoin and Antijoin Reordering. In *ICDE*, pages 585–594, 2001.

[Roy98] Prasan Roy. Optimization of DAG-Structured Query Evaluation Plans. M.tech. thesis, Dept. of Computer Science and Engineering, IIT-Bombay, January 1998.

[RSSB00] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and Extensible Algorithms for Multi Query Optimization. In *SIGMOD*, 2000.

[SAC+79] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access Path Selection in a Relational Database Management System. In *SIGMOD*, pages 23–34, 1979.

[Sel88] Timos K. Sellis. Multiple-Query Optimization. *ACM Trans. Database Syst.*, 13(1):23–52, 1988.

[SPMK95] Michael Steinbrunn, Klaus Peithner, Guido Moerkotte, and Alfons Kemper. Bypassing Joins in Disjunctive Queries. In *VLDB*, pages 228–238, 1995.

[SSM96] David E. Simmen, Eugene J. Shekita, and Timothy Malkemus. Fundamental Techniques for Order Optimization. In *SIGMOD*, pages 57–67, 1996.

[ZLFL07] Jingren Zhou, Per-Åke Larson, Johann Christoph Freytag, and Wolfgang Lehner. Efficient exploitation of similar subexpressions for query processing. In *SIGMOD*, pages 533–544, 2007.