Single Phase Construction of Optimal DAG-structured QEPs

Thomas Neumann
Guido Moerkotte

MPI–I–2008–5–002 June 2008

Authors' Addresses

Thomas Neumann
Max-Planck-Institut für Informatik
Campus E1 4
66123 Saarbrücken
Germany

Guido Moerkotte
Universität Mannheim
B6, 29
68131 Mannheim
Germany

Abstract

Traditionally, database management systems use tree-structured query evaluation plans. They are easy to implement but not expressive enough for some optimizations like eliminating common algebraic subexpressions or magic sets. These require directed acyclic graphs (DAGs), i.e. shared subplans.

Existing approaches consider DAGs merely for special cases and not in full generality. We introduce a novel framework to reason about sharing of subplans and, thus, DAG-structured query evaluation plans. Then, we present the first plan generator capable of generating optimal DAG-structured query evaluation plans. The experimental results show that with no or only a modest increase of plan generation time, a major reduction of query execution time can be achieved for common queries.

Keywords

Query Optimization, DAG-structured Query Plans

Contents

1 Introduction **3**

2 Problem Definition **6**
2.1 Optimizing Trees . . . 6
2.2 Optimizing DAGs . . . 7
2.3 Related Problems . . . 8
2.4 Problems Not Treated in Depth . . . 9

3 Algebraic Optimization **11**
3.1 Using Tree Equivalences . . . 11
3.2 Share Equivalence . . . 13
3.3 Optimizing DAGs . . . 14
3.4 (No) Optimal Substructure . . . 15
3.5 Reasoning over DAGs . . . 15

4 Leveraging Existing Techniques **18**
4.1 Identifying Common Subexpressions . . . 18
4.2 Compensation Plans . . . 19

5 Plan Generator Skeleton **21**
5.1 Overview . . . 21
5.2 Search Space Organization . . . 22
5.3 Sharing Properties . . . 23
5.4 Effects on Search Space . . . 25
5.5 Plan Generation Overview . . . 27

6 Plan Generation Algorithms **30**
6.1 Preparation Phase . . . 30
6.1.1 Properties . . . 30
6.1.2 Share Equivalence . . . 33
6.2 Search Phase . . . 35
6.3 Reconstruction . . . 37

7 Cost Calculation **39**
7.1 Calculation for Trees 40
7.2 Problems when using DAGs 40
7.3 Calculation for DAGs 41
7.4 Calculation in Linear Time 44
7.5 Calculation in Linear Time and Space 46
7.6 Comparison 47
7.7 Full Algorithm 47

8 Execution **49**
8.1 Execution Strategies 49
8.1.1 Using Trees 49
8.1.2 Using Temp 49
8.1.3 Share Only Materializing Operators 50
8.1.4 Parallel Execution 51
8.1.5 Pushing 51
8.2 Pushing Tuples Up 52
8.2.1 Scheduling 56

9 Evaluation **57**
9.1 TPC-H 57
9.2 Bypass Plans 59
9.3 Conclusion 61
9.4 Related Work 61
9.5 Conclusion 63

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 [22]. Conceptually, the algebra consists of operators working on sets, bags, or sequences. 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, for example, the following SQL view and query:

```
create view cvalue(ckey,total) as
   select ckey, sum(price)
   from   customer, order
   where  ckey=ocustomer
   group  by ckey

select ckey
from   cvalue
where  total > (select avg(total)
                from cvalue)
```

When executing the query, the view `cvalue` is accessed twice; once to calculate the average revenue and once to produce the final result. Without precautions, this directly results in evaluating it twice because in a tree-structured plan, results of a single operator cannot be consumed by multiple other operators. A typical tree-structured query execution plan for the query is given on the left-hand side of Fig. 1.1. The resulting duplicate work can be avoided by reusing the result of the view. This yields a DAG-structured execution plan (right-hand side of Fig. 1.1).

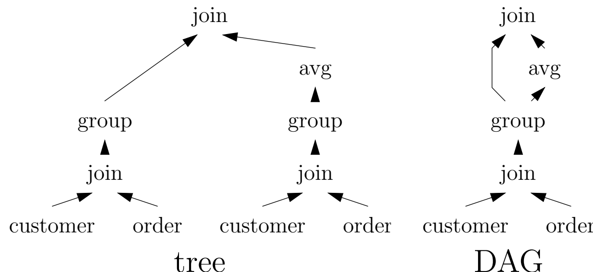

Figure 1.1: Example plans

This kind of optimization is often used for view execution in commercial database systems [11]. However, current implementations have three severe limitations. First, at query compilation time, the results to be shared are determined before the actual plan generation takes place. Thus, there is a high likelihood of producing suboptimal plans. For example, it is hard to choose between sharing the view and pushing predicates into the view. Second, at query execution time, the sharing of results is implemented by materializing them, e.g. using a temp operator. This is obviously an expensive endeavor. Third, primarily views and only some other well-defined cases are shared. Personal communication [8, 20] suggests that commercial database system can share more than just views (we discuss some examples in Section 2.3), but this is not described well in the literature. Therefore, we assume that although they use DAGs in some cases, DAG generation is not fully integrated into query optimization; the principal approach outlined in [11] is still used.

A further motivation for considering DAG-structured plans are optimization techniques such as magic sets [24] and bypassing [4], which directly lead to DAGs. Our contributions are as follows. We first show that common algebraic equivalences are no longer valid when plans are DAG-structured. This motivates our first contribution: a novel framework for reasoning about sharing. More specifically, we introduce the notion of *share equivalence*, which is orthogonal to the well-known notion of equivalence, where two plans are equivalent if and only if they produce the same result under all circumstances. Building upon this framework, we then propose the first plan generator that (1) allows sharing whenever possible (i.e. with no restriction to special situations like views) and (2) guarantees to generate the overall optimal plan. Neither of the two goals is reached by any current plan generator (see related work for a discussion).

The rest of the paper is organized a follows. Section 2 briefly describes the setting of this work and compares it with other settings discussed in the literature. Section 3 describes the theoretical foundation for reasoning

about DAGs. The general idea of how this can be used for rule-based plan generation is shown in Section 5. It gives a bird's-eye view of our rule-based plan generation algorithm. Its details are discussed in Section 6. The execution of DAG structured query plans is discussed in Section 8. Experimental results demonstrate the benefits of our approach and quantify the overhead of our plan generator compared to a conventional one (Section 9). Finally, Section 9.4 discusses related work, and Section 9.5 summarizes our work and gives a brief outlook.

2 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.

2.1 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 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 E into E' using known equivalences [14, 16, 15], or constructively by building EG' incrementally [21, 33]. 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. 3).

2.2 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. [36]). 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. 3).

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. [36]) 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.

2.3 Related Problems

The most often found form of DAG construction in commercial database system is the elimination of common subexpressions (e.g. [2, 9]). The optimizer detects some predefined situations where DAG construction is beneficial and prefers to reuse an intermediate result instead of computing the result twice. Examples include nested queries, OLAP operators in DB2, multiple aggregations over the same data in SQL server, etc. While beneficial, these techniques are hard-coded into the optimizer and not the result of a generic cost-based DAG construction strategy. The optimizer only detects some known cases before the actual plan generation starts. In contrast, our approach always considers DAGs as one possible execution strategy during plan generation.

A variation of the problem of eliminating common subexpression is that of view merging [11]. Here, the optimizer has to decide if and how to reuse common view expressions. This is more difficult, as sometimes predicates can be pushed down into a view. This could prevent sharing but might also improve execution time. Currently, this decision is made before plan generation by using heuristics. But it should be obvious that the proper solution is to make this decision cost-based during plan generation. Again this requires a more generic DAG support in the query optimizer.

DAG construction is also common in multi-query optimization [5, 32, 34].

However, DAG construction for multi-query optimization is quite different from single query optimization. Multi-query optimization is much more static in that it concentrates on reusing overlapping parts of the queries where the overlap is determined before plan generation starts. Hence, the focus is on optimally sharing intermediate results. Further, the potential overlap can be easily identified by looking at the query [34]. In this work, we take a much broader approach to reusing intermediate results. As we will see DAGs are not only constructed to avoid performing the same operation twice, but also to reuse any equivalent intermediate result, where the notion of equivalence is broader than for trees. This means that (suitable) non-identical intermediate results are shared, which is not yet done in multi-query optimization. Another difference is that for multi-query optimization, the queries are not executed simultaneously, at least the operators are not as strictly coupled as for single query evaluation, i.e. scheduling is more important. This means that the optimizer has to decide e.g. when to use materialization to decouple parts, which is not relevant for single query optimization. These scheduling problems greatly affect the search space [5]. This is probably the reason why multi-query optimization uses much simpler DAG construction techniques, as the increased search space prevents more difficult optimizations.

2.4 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 [18]. In addition to this generic approach, there are many special cases like e.g. application in parallel systems [12]. The more general usage of DAGs is considered in [31] and [25], which describe runtime systems for DAGs. We discuss the fundamental aspects of executing DAGs in Section 8.

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 [25]. We will give an overview of the basic cost calculation methods in Section 7.

3 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.

3.1 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. [10, 23]). 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. 3.1. 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,

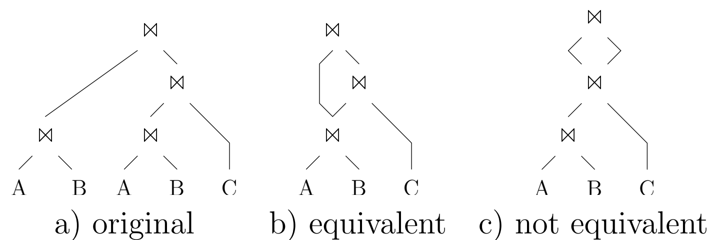

Figure 3.1: Invalid transformation for DAGs

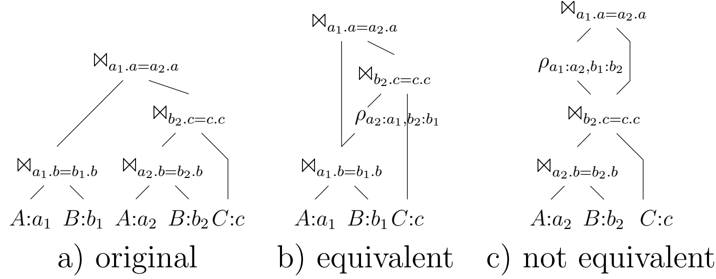

Figure 3.2: More verbose representation of Fig. 3.1

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 occurences 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. 3.1 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. 3.2 a), the original expression consists of two different joins $A \bowtie B$ 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.

3.2 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

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

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.

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. 3.3. 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.

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

Figure 3.3: Definition of share equivalence for common operators

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. 3.1 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 reordable joins, as discussed in Section 2.1.

3.3 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 as needed. Logically, the

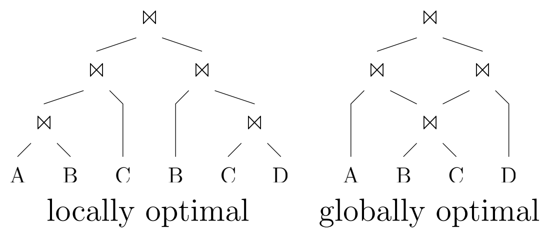

Figure 3.4: Possible non-optimal substructure for DAGs

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.

3.4 (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. 3.4, 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.

3.5 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.

$$\begin{aligned} t \equiv_T t' &\Rightarrow \theta(t) \equiv_T \theta(t') \\ t_1 \equiv_T t'_1 \wedge t_2 \equiv_T t'_2 &\Rightarrow (t_1 \theta t_2) \equiv_T (t'_1 \theta t'_2) \end{aligned}$$

These conditions are a fundamental requirement of constructive plan generation, but as seen in Sec. 3.1, 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:

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

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:

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

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:

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

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 [24, 36].

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. 3.4, 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:

$$\begin{aligned} S(d_1) = S(d_2) &\quad \text{iff} \quad d_1 \equiv_S d_2 \\ S(d_1) < S(d_2) &\quad \text{iff} \quad \exists d_2' \subset_D d_2 : d_1 \equiv_S d_2' \end{aligned}$$

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 [33, 35]. 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.

4 Leveraging Existing Techniques

Many aspects of DAG generation, occur in different contexts, too, for example multi-query optimization and view selection. In particular detecting common (or similar) subexpressions is a well-studied field. We now discuss several of these existing techniques, as they are important for generating useful DAGs. Note that the purpose of this paper is not so much the detection common expressions (which we discuss below), but the usage of this information during plan generation to generate the optimal DAGs. The techniques below are used during the preparation phase of our algorithm (see Section 6.1) to form the search space of the plan generator.

4.1 Identifying Common Subexpressions

A commonly used technique is the elimination of common (scalar) subexpressions. The optimizer tries to detect duplicate computations and then reuses the already computed values if possible. Besides saving computations, eliminating common subexpressions can help in other cases like exploiting indices more efficiently etc. [3]. The main problem here is detecting identical computations, which can be hard. We discuss a more general approach in the next paragraph, usually the expressions are normalized into a canonical form and then the optimizer searches for exact matches. Interestingly elimination of common subexpressions has a great effect on query optimization: Is has been shown [26] that even even the relatively simple problem of ordering selections and maps (i.e. scalar computations) becomes NP hard when taking the elimination of common subexpressions into account. This is a strong indicator that the decision about elimination of common subexpressions, and thus, in the DAG context, the decision about reusing intermediate results, has to be made by the plan generator itself. Fixing the elimination of

common subexpressions in a separate steps risks losing the optimal solution.

The more general problem is finding parts of a query that can be answered by other query fragments. An early paper that studies this problem is [7]: It first normalizes the query predicates and then computes a signature as a filter. If the signature indicates that parts of the query might be answered by a known fragment it performs more expensive tests, ultimately checking if the query predicate implies the fragment predicate. Note that this goes beyond detection of duplicate computations, as the algorithm searches for fragments that compute a superset of the required tuples, potentially requiring to check the query predicate again. Other multi-query optimization papers use similar techniques to identify subqueries that imply other subqueries (e.g. [34]).

Similar matching algorithms are also used for materialized view matching. An interesting concept is shown in [39]: It contains matching rules plus compensation actions for each rule that describe how a materialized view with certain structures can be used for certain queries. As the materialized view might not produce exactly the tuples required for the query, the compensation steps adjust the view output to match the desired result. This fuzzy matching is very important to get the maximum benefit (i.e. match as many queries as possible) from the materialized view.

We assume that these matching techniques are used before running the plan generator itself, i.e. the information about potentially equivalent subexpressions is available to the plan generator. As emphasized in [32], the two problems of detecting and exploiting common expressions are orthogonal, we concentrate on the second part.

4.2 Compensation Plans

Several of the multi-query optimization or view matching techniques find query pairs that are not identical but where some compensation steps are required to transform one query result into the other. Similar compensation steps are used in aggregation queries to express one aggregation using another. But the plan generator discussed in the following sections only identifies identical (or, more precisely, share equivalent) plan fragments. This discrepancy is caused by the different abstraction levels: The plan generator tries to directly reuse the output of algebraic operators for a different part of the query, which is only possible if the output is the same. The matching algorithms operate on a higher abstraction layer and therefore consider necessary compensation steps, which would be very expensive during plan generation. Still, the compensation plans can be used, they only have to be made known to the plan generator: When part of a query can be replaced

by another part including compensation actions, the preparation step adds a *Choice* operator with both the original query fragment and the new query fragments as input. The *Choice* operator is a pure logical operator that only reads the cheapest of its inputs, which implies that the final physical plan will only contain the cheapest of the two alternatives. Thus the plan generator tries both the original plan and the compensation plan, where part of the compensation plan will allow for sharing with other parts of the query.

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. [21], 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 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 Section 6.1 for a detailed discussion. Other missing and essential details will also be discussed in Section 5.5. 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 5.5, 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 is operators 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 tack 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 [29], each operator specifies which other operators have to be applied before it becomes applicable. We discuss this computation in Section 6.1.1. 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, that ordering/grouping properties are handled separately, as the number of orderings/groupings can be very large. We use the data structure described in [28] 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 3.5: 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 (see Section 6.1.2). 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. 3.4 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 $A \bowtie_1 (B_1 \bowtie_2 C_1)$ would set sharing bits for B_1, C_1 and $\bowtie_2$, as the subplan can be shared, while the plan $(A \bowtie_1 B_1) \bowtie_2 C_1$ 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 3.4.

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 Effects on Search Space

An interesting question is how the search space (and thus the optimization time) is affected by DAG support. Unfortunateley there is no easy answer. We split the question in two, and first study the effect of *sharing* properties and then the effect of DAG handling on the search strategy.

The *sharing* properties can increase the search space, as they render plans incomparable. Or rather half comparable, as a plan with more sharing opportunities can still dominate a plan with less sharing opportunities. The lower bound for this effect is a factor of 1 (i.e. no increase), as the plans with more sharing possibilities could still be optimal in themselves, dominating all plans with less sharing possibilities. A loose upper bound would be a factor of 2^n for n sharing bits. We can improve this bound by observing that not all sharing combinations are valid for a given plan: An operator can only be shared if all of its input operators can be shared too.

For each sharable operator there are two cases: Either the operator sets its sharing bit, which requires that all its descendants have set it, too, or it does not set its bit and its children can decide for themselves. Note that in reality there is no decision: The bit is set whenever possible. However, as this is query dependent let us pretend that not setting the bit is a possibility. This way, we can derive an upper bound. For unary operators the upper bound is n, as at most each possible tail of the operator chain can be shared. This observations lead to the following recurrence f_U for an upper bound for queries with n sharable operators (more precisely: n equivalence classes, we only have to consider the equivalence class representatives).

$$
\begin{aligned}
f_U(0) &= 1 \\
f_U(1) &= 2 \\
f_U(n) &= 1 + \max_{1 \leq i < n} [f_U(i) * f_U(n-1-i)].
\end{aligned}
$$

This grows slower than 2^n, but is still exponential in the number of equivalence classes ($\approx 1.5^n$). Note that only equivalence classes with at least two elements are relevant here. The upper bound seems unfortunate, but is still not too bad for real queries. In particular this is true because the number of equivalence classes with at least two elements is at most half the number of sharable operators. Thus, for a query with 20 operators forming at most 10 equivalence classes the search space is in the worst case increased by a factor ≤ 81.

While this upper bound is still not tight, future analysis of DAG construction complexities should concentrate an some kind average case results, as the worst case is very unlikely to occur. For SPJ queries it even seems to be the case that an exponential number of non-sharable operators is required to trigger the exponential growth in the number of equivalence classes. As average result studies will have to be query type specific. This analysis is beyond the scope of the current paper.

The equivalence classes increase the search space through the sharing properties, but they also help the search space exploration, as problems are solved using equivalence class representatives whenever possible. This avoids duplicate work. The lower bound for this is again a factor of 1 (no effect), but the upper bound for this speedup is the size of the largest equivalence class. Intuitively, the optimizer will try to solve shared subproblems only once, saving time. Often subproblems will not be exactly the same, e.g. due to selections that can be pushed down in one case and not the other, but even then duplicate computations can occur for smaller subproblems. For this reason DAG construction can be faster than tree construction, simply because the optimizer detects duplicate problems.

query		operators
select $a.a_2$,$b.a_2$	a	scan of $R_1(a_1,\ldots,a_4)$
from R_1 a,R_2 b	b	scan of $R_2(a_1,a_2,a_3)$
where $a.a_1 = b.a_1$ and	c	scan of $R_1(a_1,\ldots,a_4)$
$a.a_3$ >(	$\bowtie$	join on $a.a_1 = b.a_1$
select avg($c.a_3$) from R_1 c	Γ	calculate avg $c.a_3$
)	$\ltimes$	selection $a.a_3$ >avg

Figure 5.1: Example query as illustration

The first factor increases the search space while the second factor shrinks the explored the search. The overall effect is difficult to predict. For queries with a large number of equivalence classes the increase will dominate, i.e. DAG construction will be expensive. But for queries with a handful of equivalence classes the speedup will be more noticeable, rendering DAG construction quite cheap.

5.5 Plan Generation Overview

After describing the general approach for plan generation, we now present the actual algorithms used. 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 the next section. As a running example, we use the query shown in Figure 5.1: It joins two relations and performs a selection that checks an attribute against the average, reading the involved relation twice. Note that the operators on the right hand side were chosen to illustrate certain aspects of the algorithm later on, in practice the translation might look different. It is sufficient to use the produced attributes and operators

as logical properties. All dependencies can be expressed this way.

Within the search space, the plan generator tries to find the cheapest plan satisfying all logical properties required for the final solution. In our simple example, this means that all operators have been applied, as there are no choices between operators. 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 [19] 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 [15, 16]. The search phase is started by requesting the cheapest plan that provides the goal properties.

PLANGEN(*goal*)
1 $plans \leftarrow memoizationTable[goal]$
2 **if** *plans* already computed
3 **then return** *plans*
4 $plans \leftarrow$ create a new, empty *PlanSet*
5 $shared \leftarrow goal$ rewritten using representatives
6 **if** $shared \cap goal = \emptyset$
7 **then** $plans \leftarrow$ PLANGEN($shared$)
8 **for each** r **in** instantiated rules
9 **do** $filter \leftarrow r.produces \cup r.required$
10 **if** $filter \subseteq goal$
11 **then** $sub \leftarrow$ PLANGEN($goal \setminus r.produced$)
12 $plans \leftarrow plans \cup \{r(p)|p \in sub\}$
13 $memoizationTable[goal] \leftarrow plans$
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 3.5. 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 Plan Generation Algorithms

After giving a high-level view of generating DAG-structured query plans we now go into more detail about the algorithms. Note that this section mainly deals with technical details, the general issues have been discussed in the previous sections.

The plan generation is divided into three phases: The preparation phase examines the query and gathers relevant information, the search phase uses this information to find the optimal execution plan, and the reconstruction phase builds a complete plan out of the condensed representation used by the search phase. We will now look at these different phases in more detail.

6.1 Preparation Phase

The preparation phase examines the query and collects the information necessary for plan generation. The main tasks are: Identification of relevant operators, identification and minimization of logical properties, identification of share equivalence, and precomputation of the ordering and grouping inference as described in [27]. Here, we examine two other complex steps: the property minimization and the construction of equivalence classes for share equivalence.

6.1.1 Properties

To make the plan representation more compact (and also to reduce the search space), the preparation phase prunes and minimizes the property specification. This consists of three steps: First, all logical properties are collected. Thereby, produced and required logical properties are kept separate, resulting in two sets of sets. The algorithm keeps each individual logical property combination (and, therefore, produces a set of sets instead of a set), as this is required to check which logical property combinations can be produced.

```
COLLECTPROPERTIES()
1  produced ← ∅
2  required ← ∅
3  for each r in instantiated operator rules
4      do produced ← produced ∪ {r.produced}
5         required ← required ∪ {r.required}
6  return (produced, required)
```

For the running example in Figure 5.1, the set of produced properties is $\{a.a_1, \ldots, a.a_4, b.a_1, b.a_2, b.a_3, c.a_1, \ldots, c.a_4,\ avg, \bowtie, \Gamma, \ltimes, sigma\}$, that is, all attributes and all operators. The set of required properties is $\{a.a_1, \ldots, a.a_3, b.a_1, b.a_2,\ c.a_3, avg, \Gamma\}$, the attributes required for the operators and the aggregation operator (as it cannot be reordered freely).

Note that collecting the operator properties can be somewhat involved, depending on the operators used. For simple SPJ queries with inner joins and selections, the properties can be specified easily. For more complex operators, e.g. outer joins, more information about dependencies is required. A very nice approach to get these based on extended eligibility lists (EELs) is described in [29]. The algorithm examines the original query bottom up, and expresses reordering constraints for operators by computing the set of relations that has to be part of the input before an operator is eligible. This is computed in addition to the "normal" eligibility constraints caused by variable bindings etc. (therefore the name extended eligibility lists). From these EELs, the required properties of an operator can be derived directly: Each relation scan produces a certain logical property, and each operator requires the logical properties corresponding to entries in its EEL. The EEL construction described in [29] only covers outer joins and anti joins, but it can be generalized to include e.g. semi joins. The collection of operator properties is inherently operator specific, as it expresses the semantic of an operator, but the logical properties are a very generic and flexible concept to express arbitrary operator constraints.

Besides the property specifications, the preparation phase also determines the goal of the query (i.e. the logical properties the final plan must satisfy). This could be done in several ways. One possibility is to collect all logical properties of the logical operators in the original query. In our simple example, this is identical to the *produced* set calculated above.

```
BUILDGOAL()
1  goal ← ∅
2  for each r in instantiated operator rules
3      do if r represents a logical operator in the query
```

```
4         then goal ← goal ∪ r.produced
5   return goal
```

Now this information can be used to prune properties. The algorithm checks which logical properties can be used to construct the *goal* properties (potentially transitively) and removes all logical properties that are not useful. Afterwards, it checks which logical properties can be satisfied by the remaining produced properties and eliminates all those that cannot be satisfied. Note that while the algorithm shown below only minimizes the property sets, the preparation phase afterwards removes all rules whose logical properties are no longer relevant or cannot be satisfied. The parameters *produced*, *required* and *goal* are the result of COLLECTPROPERTIES and BUILDGOAL. As our example contains no operator alternatives, no properties can be removed (*goal*=*produced*). However, redundant properties will be merged in the next step.

```
PRUNEPROPERTIES(produced, required, goal)
 1  useful ← goal
 2  for each r in instantiated rules
 3     do if r.required ∈ required∧
 4           r.produced ∩ useful ≠ ∅
 5           then useful ← useful ∪ r.required
 6  for each r in instantiated rules
 7     do if r.produced ∩ useful = ∅
 8           then produced ← produced \ {r.produced}
 9  possible ← ∅
10  for each r in instantiated rules
11     do if r.produced ∈ produced∧
12           r.required ⊆ possible
13           then possible ← possible ∪ r.produced
14  for each r in instantiated rules
15     do if r.required ⊈ possible
16           then required ← required \ {r.required}
17  if produced or required were modified in this pass
18    then goto line 1
19  return (produced, required)
```

The remaining logical properties are now minimized. While all remaining logical properties can be produced, they might still contain irrelevant entries (if other entries in the same set are relevant). These are removed, and all logical properties that are always produced together are merged. In our

running example, this removes the properties $a.a_4, b.a_3, c.a_1, c.a_2, c.a_4$ and merges the properties $a.a_1, \ldots, a.a_3$ and avg, Γ. As a result, the remaining properties can be described as $\{a, b, c, \bowtie, avg, \ltimes, \sigma\}$.

```
MINIMIZEPROPERTIES(produced, required)
 1  result ← ∅
 2  R ← ⋃_{r∈required} r
 3  for  each p in produced
 4      do for  each b in p, b ∉ R
 5              do p′ ← p \ {b}
 6                  produced ← (produced \ {p}) ∪ {p′}
 7                  p ← p′
 8                  result ← result ∪ {b → ∅}
 9  P ← ⋃_{p∈produced} p
10  for  each a in P
11      do for  each b in P, a ≠ b
12              do A ← {p|p ∈ produced ∧ a ∈ p}
13                  B ← {p|p ∈ produced ∧ b ∈ p}
14                  if A = B
15                     then result ← result ∪ {ab → a}
16  return result
```

The produced map function is then used to adjust the logical properties of the remaining rules. When calculating the map function as shown, the mapping is ambiguous (when two logical properties can be merged, any of the two can be chosen). In practice, this is solved by defining an arbitrary total ordering among logical properties and changing $a \neq b$ in line 11 into $a < b$.

6.1.2 Share Equivalence

After determining the relevant operator rules, the preparation phase decides which rules are share equivalent. This requires the definition of share equivalence from Section 5.3. The plan generator uses this notion for rules (which can consist of multiple operators), but in practice, this is no problem: The rules are treated as if they were (complex) operators. In general, the plan generator tests if two rules are structurally identical (i.e. belong to the same class, have predicates with the same structure etc.). Then, if a mapping can be found such that one rule can be replaced by the other rule and a rename, they are considered share equivalences:

$\equiv_S$RULES(r_1, r_2)

```
1  if r_1 and r_2 are structurally identical
2     then P_1 ← {r|r ∈ rules ∧ r.produced ⊆ r_1.required}
3          P_2 ← {r|r ∈ rules ∧ r.produced ⊆ r_2.required}
4          if ∀r_i ∈ P_1 ∃r_j ∈ P_2 : ≡_S RULES(r_i, r_j)
5             then if ∀r_i ∈ P_2 ∃r_j ∈ P_1 : ≡_S RULES(r_i, r_j)
6                     then return true
7  return false
```

In our running examples a and c are share equivalent: They scan the same relation, producing only different attribute bindings. To keep the example simple, we ignore the problem of projections here (c can project away more attributes than a). If projections should be considered, this can be expressed by instantiating two variants of c, one that projects early and one that does not.

The algorithm builds the mapping implicitly, by recursively testing if the input of the two operators is share equivalent. It might even be easier to construct the mapping explicitly in a bottom-up fashion, starting with share equivalent scans and then transitively considering their consumers. The equivalence relation is now used to construct equivalence classes, selecting one rule as a representative and discarding equivalence classes with only one element (as no sharing is possible then).

```
CONSTRUCTEQUIVALENCECLASSES()
1  C ← ∅
2  for each r in instantiated rules
3     do if ∃(a, b) ∈ C : ≡_S RULES(a, r)
4           then C ← C \ {(a, b)} ∪ {(a, b ∪ {r})}
5           else C ← C ∪ {(r, {r})}
6  for each (a, b) in C
7     do if |b| = 1
8           then C ← C \ {(a, b)}
9  return C
```

In our example, a and c are placed in the same equivalence class, we choose a as representative. The equivalence classes are used for two purposes: First, each representative is assigned a bit in the *sharing* property of the plans. Second, for all other entries in an equivalence class a mapping from their produced properties to the properties produced by the representative is constructed. Thus, the plan generator can rewrite a logical property set in terms of equivalence class representatives (by applying all applicable mappings) to try using a shared plan.

```
PLANGEN(goal)
 1  plans ← memoizationTable[goal]
 2  if plans is undefined
 3    then mask ← ∅
 4         for each s in rules
 5         do if s.produced ⊆ goal
 6              then mask ← mask ∪ s.produced
 7         if mask ≠ goal
 8           then return ∅
 9         plans ← a new PlanSet with properties = goal
10         shared ← goal rewritten to use ≡_S representatives
11         if shared ∩ goal = ∅
12           then plans ← plans ∪ PLANGEN(shared)
13         for each s in instantiated rules
14         do if s.filter ⊆ goal
15              then s.SEARCH(plans, goal)
16         memoizationTable[goal] ← plans
17  return plans
```

Figure 6.1: Algorithm for plan generation

6.2 Search Phase

After the preparation phase, the search is started top-down with the minimized *goal* of the query; the pseudo-code is shown in Figure 6.1. In each search step, the plan generator first checks the memoization table whether the problem was already solved. If not, it checks if the logical property combination could be produced by any rule combination, and stops the search if not (lines 3-8). This greatly reduces the search space. To share equivalent plans, the plan generator now uses the mapping from operators to their equivalence class representatives constructed in the preparation phase to rewrite *goal* in terms of equivalences class representatives (lines 10-12). If this is completely possible (the new goal is disjoint from the old one), the plan generator solves the new goal and uses the result as the result for the current problem. The check is $shared \cap goal = \emptyset$ instead of $shared \neq goal$, as only whole subproblems can be shared. If *shared* and *goal* overlap, at least one operator remains that has to be scheduled first before sharing is possible (so sharing will be tried later during the recursive search). Note that the rename operator required for using an equivalent problem is added implicitly. The search phase

only checks that a rename is possible, the operator itself is added during the reconstruction phase.

In our example, this means that the problem c could be solved by using a instead, as they are share equivalent. However, the problem avg, c could not be solved by using a rename, the rewriting is incomplete (avg is not in an equivalence class). The partial problem c could still be solved using a, but only in a later step where avg were split off already.

After checking for share equivalence, the plan generator looks at all known rule instances, checks if their *filter* is a subset of *goal* (if not, the rule cannot produce a plan with the desired logical properties). If yes, it asks the rule to build a plan and to store it in the plan set. The plan set is passed down instead of the rule returning a plan, as there can be more than one plan for a given goal and, besides, the rules can use already known plans for pruning.

The rules direct the navigation of the search space. Consider a simple selection operator. When the plan generator asks a selection rule to generate a plan, the selection asks the plan generator to produce a plan without the selection and then adds the selection (note that *plans* is an instance of *PlanSet*, which automatically prunes dominated plans):

```
SELECT::SEARCH(plans, goal)
1  for each p ∈ PLANGEN(goal \ produced)
2      do p′ ← add the selection to p
3         plans ← plans ∪ {p′}
```

For binary operators like joins, the navigation is more complex. However, most of the required functionality is the same for all binary operators and, therefore, can be factored in common rule fragments in an implementation. The general rule for joins is shown below:

```
JOIN::SEARCH(plans, goal)
1  space ← goal \ (produced ∪ reqLeft ∪ reqRight)
2  for each lp ⊂ space
3      do rp ← space \ lp
4         for each l in PLANGEN(lp ∪ reqLeft)
5             do for each r in PLANGEN(rp ∪ reqRight)
6                    do p ← join l and r
7                       plans ← plans ∪ {p}
```

The joins first check which logical properties can be satisfied arbitrarily, i.e., are neither produced nor required by itself. These are called *space* here, as they actually describe the search space for the join rule. The rule has to decide which of these logical properties must be satisfied by the left subplan and which by the right subplan. This is done by enumerating all subsets of

space and asking the plan generator for plans satisfying these requirements (and the requirements of the join itself). These plans are then combined to a new partial plan and memoized if they are not dominated by some existing plans.

Consider, for example, the partial problem with two relations a and b, the join $\bowtie$ between them and the avg condition σ (we use a selection instead of the actual semijoin here to keep the example simple). Now the join rule is asked to produce a plan with the logical properties $a, b, \bowtie, \ltimes$. The join itself produces the property $\bowtie$ and it requires a and b. The only remaining logical property is σ, so *space* $= \sigma$ (see JOIN::SEARCH). The join rule does not know on which relation the selection can be applied, so it first asks the plan generator to produce plans with a and σ, and then asks for plans with b, which are joined. Afterwards, it asks for plans with a and plans with b and σ, which are also joined. As the selection can only be applied to one relation, one of these sets will be empty, but the join rule does not understand the semantics of the selection and, therefore, tries both possibilities. Note that this does not imply that selections are pushed down: The selection rule itself could have been scheduled before, resulting in *space* $= \emptyset$. In this case, the join rule would consider a single plan.

6.3 Reconstruction

The reconstruction phase is mostly straight-forward, as the plan generator simply calls the recursive BUILDALGEBRA function on the root of the final plan to produce an expression of the physical algebra. The only problem is that some additional information is required: First, the plans have no reference to their enclosing set of equivalent plans, so all information stored there (especially the logical properties) is not available. Second, the graph forms a DAG, which means that plan nodes are visited multiple times, although they correspond to only one physical operator. Third, the renames due to share equivalence are only implicit during plan generation and have to be converted into explicit renames. Still, this can be done easily by storing the required information in a hash table (called *recTable*) that is used during reconstruction. The hash table is a map plan $\rightarrow$ (algebra expression, logical properties) and is filled during a depth-first search. For a selection, the reconstruction code is sketched below:

SELECT::BUILDALGEBRA(*plan*)
1 $(algebra, properties) \leftarrow recTable[plan]$
2 **if** $(algebra, properties)$ is undefined
3 **then** $input \leftarrow plan.left.rule.$

4 		buildAlgebra($plan.left$)
5 	$ip \leftarrow recTable[plan.left].properties$
6 	**if** $required \not\subseteq ip$
7 		**then**
8 			$nip \leftarrow$ rename ip by checking $plan.left$
9 			$input \leftarrow \rho_{ip \rightarrow nip}(input)$
10 			$ip \leftarrow nip$
11 	$algebra \leftarrow$ add new select operator to $input$
12 	$properties \leftarrow ip \cup produced$
13 	$recTable[plan] \leftarrow (algebra, properties)$
14 **return** $algebra$

The rule first checks if the physical algebra expression has already been constructed. If not, it requests the physical algebra expression of its input and looks up the corresponding properties. If the requirements of the rule cannot be satisfied by these properties, a rename is required. The subplan is scanned to find this rename (the plan must contain equivalence class representatives, which are equivalent to rules that can produce the required properties) and a physical rename expression is added. Afterwards, the selection can be applied, so the physical expression is added, the new properties are calculated and both are stored in the hash table.

The reconstruction is similar for binary operators like joins: Both input plans are examined recursively, renames are added as needed, and the two expressions are combined using a physical operator.

7 Cost Calculation

Query optimization always requires to decide whether one plan is better than another plan. This requires calculating the costs of a plan, where the costs can have different meanings. A reasonable choice would be expected query execution time, but the exact meaning is not relevant for the calculation itself. This cost calculation is relatively straight-forward for trees, but becomes much more complex for DAGs. In this section, we first look at the simple tree case, then explain why DAGs are more difficult, and finally present algorithms to calculate the costs for DAGs. We do not assume a concrete cost model here, but base the calculation on two assumptions. First, each operator produces two kinds of costs: the first amounts for producing its result and the other for reading the (already produced) output again. For example, reading the output of a sort operator twice is not twice as expensive as producing and reading it once. Second, we assume that multiple parallel reads (due to the DAG structure) do not cause additional costs. This can be achieved by using a suitable runtime system [25], sketched in Section 8.

During plan generation, the costs are calculated incrementally. This means that the partial plans are annotated with the costs they cause, and when a rule creates a new partial plan, it takes the costs caused by its input and adds the costs for the newly added operator. The costs for the operator itself are the same for trees and DAGs. However, the costs caused by the input can be different (due to sharing, as we will see below). Therefore, we only discuss how to calculate the costs caused by reading the input here. Note that this is only a problem for binary operators, as unary operators cannot construct DAGs (the input can be a DAG, but it can be treated as a scan with given costs). Therefore, the cost calculation requires a function

INPUTCOSTS(*left, leftReads, right, rightReads*) : *cost_t*

which calculates the costs of reading *left leftReads* times and *right rightReads* times. This function can then be used to calculate the costs for arbitrary binary operators.

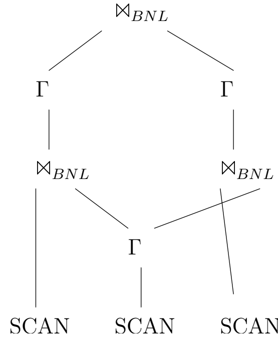

Figure 7.1: DAG-structured query plan

7.1 Calculation for Trees

If it is clear that the operators actually form a tree (i.e. the two subgraphs are disjoint; they themselves may form a DAG) the costs can be computed easily: Just multiply the costs with the number of reads and take into account that the first read might be more expensive.

inputCostsTree($l, lReads, r, rReads$)
1 $lCosts \leftarrow lReads * l.costs.further$
2 $rCosts \leftarrow rReads * r.costs.further$
3 $lDelta \leftarrow l.costs.first - l.costs.further$
4 $rDelta \leftarrow r.costs.first - r.costs.further$
5 $result.further \leftarrow lCosts + rCosts$
6 $result.first \leftarrow lCosts + rCosts + lDelta + rDelta$
7 **return** *result*

Here, `first` denotes the first construction of a result and `further` any further access to it.

7.2 Problems when using DAGs

This simple approach does not work for DAGs. Consider the DAG-structured query plan in Figure 7.1 (Γ is a group-by operator). Here, the cost of the final blockwise nested loop join (BNL) cannot be calculated in a straightforward top-down calculation. In particular, it cannot be determined by combining the cost of the two input operators. When the topmost join is treated like a normal join in an operator tree, the costs are overestimated, as the shared group-by operator is not taken into account. Since this operator serves two joins simultaneously, its costs should only be counted once. What

complicates the calculation even more is that the shared operator appears twice on the right-hand side of a nested loop and, therefore, is read multiple times. The actual number of reads can only be determined by looking at all operators occurring in the plan. This makes calculating the costs complex and expensive. Ideally, the cost calculation should touch each operator at most once (resulting in linear runtime) and require only constant additional memory in the plans, as memory is usually tight during plan generation. Achieving both at the same time is difficult, thus, we present three different cost algorithms in the next sections that choose different time/space trade-offs. We start with one that only requires constant additional space per operator but has an overall exponential worst-time complexity, over one that has a linear worst-case complexity (and is very fast) but requires additional memory, until we reach one that guarantees linear runtime with only constant additional memory.

7.3 Calculation for DAGs

For DAGs, the calculation is much more complex than for trees. Shared subgraphs that are read by multiple operators have to be taken into account. For the cost calculation, the actual number of passes (i.e. number of plan executions) has to be calculated, which can be lower than the number of reads in the case of sharing.

The main idea is to use a *passes* entry embedded in the plan state to keep track of the number of reads which is reset to zero after each cost calculation (it could also be stored in a side table just used for each calculation). The two input plans are traversed top-down. When visiting an operator for the first time, the costs are calculated as normal and the number of passes is stored. Further traversals can determine that the operator was already used and now only need to check if they require additional passes and calculate the costs accordingly. Some care is needed to accurately compute the number of passes, especially concerning (potential chains of) nested loops and materializing operators like temp.

inputCostsDAG($l, lReads, r, rReads$)
1 $rc \leftarrow r.rule.$dagCosts$(r, rReads)$
2 $lc \leftarrow l.rule.$dagCosts$(l, lReads)$
3 reset *passes* to zero
4 $result.first \leftarrow lc.first + rc.first$
5 $result.further \leftarrow lc.further + rc.further$
6 **return** $result$

Now the operator rules have to describe how the costs propagate through the operator tree. For the basic scan operations, this is trivial, it just needs to examine *passes* to detect shared subgraphs and to check if additional reads are required:

```
SCAN::dagCosts(plan, reads)
 1  if plan.passes = 0
 2     then result ← I/O costs for reads passes
 3          plan.passes ← reads
 4          return result
 5  if reads > plan.passes
 6     then inc ← reads − plan.passes
 7          result.first ← plan.cost.further ∗ inc
 8          result.further ← plan.cost.further ∗ inc
 9          plan.passes = reads
10          return result
11  return zero costs
```

Simple unary operators like a selection basically behave the same way, they pass the number of passes down to their children. Binary operators basically behave like unary operators. However, they have to visit both input operators. As will be discussed in the next paragraph, the order of traversal is actually important.

```
NESTEDLOOP::dagCosts(plan, reads)
 1  if plan.passes = 0
 2     then result ← r.rule.dagCosts(r, reads ∗ l.card)
 3          result ← result + l.rule.dagCosts(l, reads)
 4          plan.passes ← reads
 5          return result
 6  if reads > plan.passes
 7     then result ← r.rule.dagCosts(r, reads ∗ l.card)
 8          result ← result + l.rule.dagCosts(l, reads)
 9          plan.passes ← reads
10          return result
11  return zero costs
```

Note that the nested loop code shown can require an exponential runtime, as both input sources are visited (given a chain of nested loops where $left = right$, this could result in a runtime of 2^n). However, only the right input source is actually read multiple times. By visiting the right input first, we make sure that the *passes* entry is set to a large value. Ideally, all further

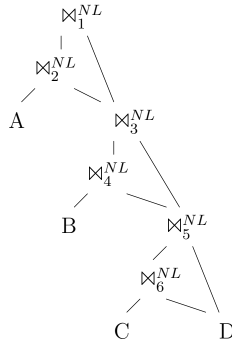

Figure 7.2: A DAG requiring exponential runtime

visits require a lower or equal number of passes, resulting in a linear time consumption.

While this is true most of the time, some extreme trees indeed trigger the exponential behavior. An example for this is shown in Figure 7.2. Depending on the selectivities and the cardinalities of the relations, the left-hand side might actually read the shared operators more often than the right-hand side: Assume that C consists of 100 tuples, D of 1 tuple and $C \bowtie D$ of 10 tuples. During the cost calculation, $\bowtie_5$ is asked to calculate its costs. As its left-hand side consists of 10 tuples, it asks D for the costs of reading D 10 times. Afterwards, it asks $\bowtie_6$ for its costs, which determines that D has to be read 100 times, which is larger than the previous value, requiring a revisit. The same situation can happen with B, visiting the right-hand side of $\bowtie_3$ twice and thus D four times (as the revisit increases the *passes* entries in the whole subgraph). The same happens for A, which leads to eight accesses to D, doubling with each additional nested loop pair.

In reality, this kind of DAGs do not occur, resulting in linear runtime of INPUTCOSTSDAG. However, as they might occur and it is unsatisfactory to have an exponential step during cost calculation, we present two algorithms with linear bounds in the next sections. However, these algorithms are more involved and slower for most DAGs. Therefore, it might be preferable to try the exponential algorithm first. Since the linear case visits each plan node at most twice, it is safe to abort the algorithm after visiting more than $2n$ operators and switch to a linear algorithm. This guarantees both good best case and worst case performance.

7.4 Calculation in Linear Time

The problem with the algorithm described above is that plan nodes are visited multiple times and require a retraversal of their children if the number of passes increases. This potentially triggers a cascade of retraversals. Actually, this is not required. We now present an algorithm that computes the costs in quadratic time. This algorithm can then be transformed into an algorithm requiring linear time.

The operators that (potentially indirectly) produce the input of a certain operator o can be divided in two groups: The first group of operators is executed a constant number of times, independently of the actual number of reads of the output of o. The usual reason for this is that they are placed below memoizing operators like temp. The second group of operators is executed a number of times proportional to the number of reads of the output of o. It does not matter if these operators are read multiple times themselves, doubling the number of reads of o also doubles the number of reads of these operators. Note that these groups may overlap, as shared operators might be read both a fixed and a proportional number of times.

This observation leads to an algorithm that computes the costs in quadratic time and space. The idea is to compute the list of operators read in a partial plan together with the number of fixed and proportional reads for each operator. For each operator, the list contains all operators occuring in its input together with the number of times these operators are read, broken down into a fixed and a proportional part. Thus, the list at the root of a DAG contains the number of reads for each operator in the DAG. The lists can be efficiently build bottom-up: For scans and unary operators this is trivial, binary operators can compute it by merging the lists of their two input operators. By using an arbitrary fixed total ordering of the operators, this merge can be done very efficiently, resulting in a total runtime of $O(n^2)$.

The algorithm is shown below, it stores the lists as *reads* in each partial plan. Note that the algorithm creates a temporary plan node as the root of the two input operators that behaves like a nested loop join with a given number of passes over each side. This is not strictly necessary, but avoids adding a special case to order the left and the right side. Besides, we use the function LOCALCOSTS to calculate the costs for n reads of one operator without the costs of its input.

INPUTCOSTSQUADRATIC($l, lReads, r, rReads$)
1 $root \leftarrow$ create a new NLJOIN($l, lReads, r, rReads$)
2 $list \leftarrow$ topological sort of $root$ and its subgraph
3 **for each** p **in** $list$ (*backwards*)

```
 4      do p.rule.buildReadOperators(p)
 5
 6  result ← zero costs
 7  for each t in root.readOperators
 8      do calls ← t.fixed + t.proportional
 9         result ← result + t.part.localCosts(calls)
10  return result
```

We assume that the list of read operators consists of triplets [*part*, *fixed*, *proportional*], where *part* specifies the plan part, *fixed* the fixed number of reads of this plan part and *proportional* the number of reads that is proportional to the number of reads of the operator itself. For scans, the list of read operators is simply empty:

```
TableScan::buildReadOperators(p)
1  p.reads ← <>
```

Simple unary operators like a selection just copy the list of their input and add the input itself:

```
Select::buildReadOperator(p)
1  p.reads ← merge(p.input.reads, < [p.input, 0, 1] >)
```

Materializing operators like a temp operator behave in the same way, but they change the proportional number of reads into a fixed number of reads:

```
Temp::buildReadOperator(p)
1  p.reads ← merge(p.input.reads, < [p.input, 0, 1] >)
2  for each t in p.reads
3      do t.fixed ← t.fixed + t.proportional
4         t.proportional ← 0
```

Joins merge the lists of their input operators, adjusting the number of reads if required:

```
NestedLoop::buildReadOperators(p)
1  lReads ← merge(p.left.reads, < [p.left, 0, 1] >)
2  rReads ← merge(p.right.reads, < [p.right, 0, 1] >)
3  for each t in rReads
4      do t.proportional ← t.proportional * p.left.card
5  p.reads ← merge(lReads, rReads)
```

Finally, the merge step simply merges the lists by taking the maximum

of $proportional$ and $fixed$:

```
MERGE(l, r)
 1  result ←<>
 2  while |l| > 0 ∧ |r| > 0
 3      do hl ← first entry of l
 4         hr ← first entry of r
 5         if hl.part < hr.part
 6            then result ← result∘ < hl >
 7                 l ← l\ < hl >
 8         if hl.part > hr.part
 9            then result ← result∘ < hr >
10                 r ← r\ < hr >
11         if hl.part = hr.part
12            then f ← max(hl.fixed, hr.fixed)
13                 p ← max(hl.proportional,
14                         hr.proportional)
15                 result ← result∘ < [hl.part, f, p] >
16                 l ← l\ < hl >
17                 r ← r\ < hr >
18  return result ∘ l ∘ r
```

So the algorithm scans the operator DAG in a bottom-up way to determine the operators transitively consumed by each operator. In the worst case, the plan is a list (e.g. n selections), resulting in quadratic runtime.

A nice property of this algorithm is that the *reads* lists do not depend on the parents of an operator and never change. Therefore, the lists can be maintained incrementally during plan generation, resulting in amortized linear runtime for each cost calculation but quadratic space. This also eliminates the need for the topological sort, as the plans are constructed bottom-up anyway (even during a top-down search).

7.5 Calculation in Linear Time and Space

The algorithm described in Sec. 7.3 does not need any additional memory besides linear space on the stack, but might require exponential runtime. The algorithm described in Sec. 7.4 guarantees linear time, but requires quadratic space. We now describe an algorithm that requires both linear time and space, with only somewhat larger constants than the algorithm requiring quadratic space.

The only reason why the first algorithm is exponential instead of linear

is that a plan node might be visited again with a $reads > passes$. If we can always guarantee that this does not happen, each node is visited at most twice, resulting in a linear runtime.

This can be achieved by visiting the plan nodes in topological order: Each plan node passes the number of reads down to its children (iteratively, not recursively). Since the nodes are visited in topological order, the number of reads does not increase after the node is visited, resulting in linear time and only requiring linear space for the topological sort. The only disadvantage is that the topological sort has to be repeated for each cost calculation. Although this can be done in linear time, it results in larger constants than the incremental approach described in the previous section.

7.6 Comparison

Experiments have shown that the incremental list algorithm is clearly the fastest. However, it requires a quadratic amount of space, which is quite large: For 1000 operators, it needs 19 MB and for 10000 operators 1.9 GB! While the more realistic example of 50 operators only requires 48 KB, this is still 1 KB per partial plan, which is too much for today's main memory. So while the incremental list algorithm is the fastest, its space requirements will probably prevent its usage in the near future.

The other algorithms are somewhat slower but only require constant additional memory. Most of the time the exponential algorithm was faster than the linear algorithm, as non-linear runtime is unlikely. For plans with 100 operators, the exponential alorithm was about twice as fast as the linear algorithm on average, and even the non-linear case was slower only about a factor of 10. The linear algorithm is a safe choice, but trying the exponential algorithm first and using the linear algorithm as fallback, as discussed in Sec. 7.3, is probably the fastest choice.

7.7 Full Algorithm

Regardless of the actual algorithm used, calculating the costs of a DAG is much more expensive than calculating the costs of a tree. Therefore, the cost model should avoid this expensive step whenever possible. This leads to the following cost function:

inputCosts($l, lReads, r, rReads, alternatives$)
1 **if** the subgraphs of l and r are disjoint
2   **then return** inputCostTree($l, lReads, r, rReads$)

3
4 $lProp \leftarrow left.costs.further * (leftReads - 1)$
5 $rProp \leftarrow right.costs.further * (rightReads - 1)$
6 $lb.first \leftarrow max(left.costs.first, right.cost.first)$
7 $lb.further \leftarrow max(leftProp, rightProp)$
8 **if** *lb* is dominated by one entry in *alternatives*
9 **then return** *lb*
10
11 **return** INPUTCOSTDAG$(l, lReads, r, rReads)$

At first, the algorithm checks if the input plans overlap. This can be done easily by inspecting if the *sharing* properties of the input plans overlap, as they mark sharable operators present in the graphs. If they do not overlap, we have the cheap tree case. Otherwise, we can compute a lower bound for the actual costs by taking the maximum costs of each input. If this lower bound is already dominated by a known alternative, the cost calculation can be canceled. Only if the plan seems interesting, the expensive calculation is performed.

Note that the tree costs can always be used as an upper bound. This is useful e.g. for bounds based pruning.

8 Execution

After discussing the creation of DAG-structured query plans, we briefly cover the execution of these plans. Using DAGs optimally, i.e. sharing intermediate results without additional costs (e.g. for materializing intermediate results) for any kinds of plans, requires changes to the runtime system. Limited forms of DAGs can be executed easier, we therefore first discuss different execution strategies and then show how a generic DAG support can be implemented.

8.1 Execution Strategies

While a direct execution of a DAG-structured query plan is usually not possible in a classical database management system, there still exist strategies to execute these plans, even in a system designed for trees. In the following, we present some of these strategies, beginning with the least invasive and ending with a very invasive but also very efficient execution strategy. As an illustrating example, we use the execution plan shown on the left-hand side of Figure 8.1.

8.1.1 Using Trees

The simplest way to execute a DAG-structured query plan is to first convert it into a tree. This is done bottom-up by creating a copy of every shared tree until all trees have unique parents. For our example, the result is shown in the second column of Figure 8.1. This strategy is not really an option. While it can handle arbitrary plans, it eliminates all advantages of the original DAG plan.

8.1.2 Using Temp

The reason why a normal database system cannot execute DAGs directly is that the same data has to be read multiple times by different consumers,

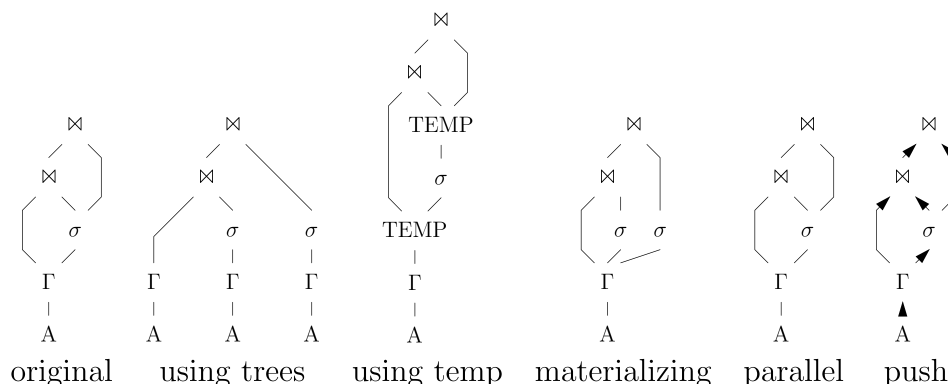

Figure 8.1: Different execution strategies

which does not work if the data is only passed between operators. However, what these systems usually support is reading the same data for different consumers if the data is stored on disk (e.g. in a (temporary) relation). This can be used to execute DAGs: Every operator that is read by multiple operators must be a scan (that is a relation, an index or a `temp` operator that spools its input to disk). The transformed plan for our example is shown in the third column of Figure 8.1: Both the output of the group-by and the output of the selection are spooled to disk, as they are read by multiple operators. While this strategy is often better than converting the DAG into a tree, the overhead for the `temp` operator is considerable. For the group-by in our example it might be still be advantageous (as a group-by is expensive and also reduces the cardinality), but for the selection it is probably not.

8.1.3 Share Only Materializing Operators

Using `temp` is expensive due to spooling, however, many operators (especially the expensive ones) spool the result or at least intermediate results to disk anyway: The sort or group-by operator, for example, have to go to disk if the data is too large, otherwise they keep the whole data in memory. In both cases, it is possible to compute the result multiple times without reading the input multiple times. The operators that do not spool to disk are usually much cheaper (but beware, a counter example is given below) and can, therefore, be executed multiple times without too much additional work. So this strategy is a combination of the previous two strategies: Shared operators that materialize their result can be shared directly and the other operators must be duplicated. The result for our example can be seen in the fourth column of Figure 8.1. This approach avoids the overhead of `temp` operators

and still allows for sharing partial results. While some minor modifications might be required to allow for multiple readers of operators, this strategy is probably the best compromise possible without major changes in an existing database management system. The disadvantage is that not all operators can be shared. While most of the operators that do not materialize are cheap, some can be very expensive (e.g. a `djoin`). Then it might make sense to add a `temp` operator, of course with the same overhead as in the previous strategy.

8.1.4 Parallel Execution

A completely different evaluation strategy that also supports DAGs is the parallel execution of operators: If every operator is a separate process (possibly even on a different computer), multiple readers are usually not a problem: As they have to synchronize their work anyway, either by a simple rendezvous protocol or by some more elaborate means, synchronizing with more than one reader is not very different. This kind of execution has been done in the past by distributed or parallel systems [12]. The disadvantage of this strategy is that overhead for passing data between operators is quite high. This might not be a problem for distributed or parallel systems, but for a single processor system executing the plans in this way is quite wasteful. And even for a distributed system it makes sense to execute the local part of a plan with reduced overhead.

8.1.5 Pushing

The main problem with executing DAGs is that the same data is consumed by multiple operators. Using the standard iterator model [22], this means that multiple consumers want to iterate over the same data independently, which usually cannot be done without buffering or spooling. This problem can be avoided by reversing the control flow: Instead of the operators iterating over their input, the input "pushes" the data up the DAG, i.e., when an operator has produced some data, it hands it to all its consumers at the same time. This is shown on the right-hand side of Figure 8.1. When using a push model, an arbitrary number of consumers can be served at the same time without any buffering or spooling. In a way, this is similar to the rendezvous protocol used for parallel execution, as each operator notifies its consumers of available data. The great advantage of the push model is that DAGs with arbitrary operators can be handled with minimal overhead and without copying data.

8.2 Pushing Tuples Up

The standard interface offered by algebraic operators basically consists of `open/next/close` methods that allow for iterating over the data, pulling the data on demand. This is not well suited for DAGs, as operators that share an input operator steal tuples from each other.

When the data is pushed bottom-up instead of pulled, the operators can no longer request data from their input. Instead, they have to wait for events, either that new data has arrived (which is then processed, potentially creating new events) or that all data has been produced and the computation can be finished. A similar strategy has been proposed in the context of XQuery processing [6]. In our approach the operators provide two callback methods for notification: DATAEVENT is called when a new tuples is available, and ENDOFDATAEVENT is called when an input is exhausted. When an operator has produced a tuple, it calls the DATAEVENT method of all its consumers; including a parameter *source* specifying the producing operator. Similarly, the ENDOFDATAEVENT method is used to notify the end of data.

The main problem of the push model is the control flow. First it has to determine which operator should produce data first. Then this operator pushes data up to its consumers, until, e.g., a join decides that another operator should produce data. Now the control flow should change, but the current operator might not be finished yet, so the control flow might change back to the current operator later on. The implementation sketched here uses an explicit scheduling mechanism that solves the control flow problems. Although this provides a push model, the scheduling itself is done in a very pull-like way (as we will see below). Therefore, we called this method "push by pull".

The operators receive data as events and they produce new data, creating events for other operators. This results in a very complex control flow. It can be formulated easily by explicitly scheduling the events. For example, it would be possible to organize all events in a priority queue using some criteria and during query processing always remove the most important event, activate the corresponding operator and enqueue the newly produced events. Such a scheme allows for arbitrarily complex data flow, and is easy to implement. However, the overhead is very high compared with pull model: the data associated with the data events has to be materialized if operators place more than one event at the same time into the queue and also the queue management itself consumes CPU.

To avoid this overhead, we restrict the scheduling in two ways. First, operators only produce new data events after their existing data events have been consumed (this avoids materializing intermediate results). Second, the

control flow changes only if it has to because of missing data (this reduces the scheduling costs). In practice, this means that operators trigger their consumers (with a direct method call) as long as possible. For example, consider a blockwise nested loop join between two tablescans. The left scan starts producing data and pushes it into the join. The join consumes this data (by storing it in a buffer), but otherwise does nothing and allows the scan to continue producing data as long as the buffer is not full. If the buffer is full, the right scan must produce data, which means that the left scan must stop. Thus, when handling an event, an operator reports if its producer should stop or continue producing data. If the producer stops, a scheduling component selects the next operator that should produce data.

This model requires a slightly more complex interface than sketched above. First, there is a separate scheduling component: The scheduler selects an operator that should produce data and this operator pushes its data up to its consumers. Second, the event methods (DATAEVENT / ENDOFDATAEVENT) can return a boolean value. If one operator requires a reschedule because it needs input from some other operator, it simply returns `false` as the result of an event. If the event was created from within another event method, this method also returns `false` etc., until the control flow reaches the scheduler which triggered the first event. The scheduler now selects the next operator and data is produced until the next reschedule is required etc. This way, the scheduler is only activated as needed, reducing the overhead to a minimum. The extended interface is shown below:

Operator
  activateNext : *Operator*

  ACTIVATE(*source*,*newSource* : *Operator*) : *boolean*
  REPORTDATA() : *boolean*
  REPORTENDOFDATA() : *boolean*

  DATAEVENT(*source* : *Operator*) : *boolean*
  ENDOFDATAEVENT(*source* : *Operator*) : *boolean*
  STARTPUSH() : *void*

The attribute *activateNext* is a hint for the scheduler: if it is set it points to the operator that should be activated instead of this operator (which usually means that the other operator has to produce data for the current operator first). It is set by the ACTIVATE method, that activates a requested operator:
OPERATOR::ACTIVATE(*source*, *newSource*)

```
1  activateNext ← newSource
2  return source = newSource
```

It sets *activateNext* and checks if the requested operator is the same as the source of the current event. If not, it returns `false`, which causes all callers to drop back to the scheduler, which can now activate the proper operator.

When an operator creates new data or reaches the end of data, it has to notify its consumers. This is done by the small helper functions REPORTDATA and REPORTENDOFDATA that trigger the corresponding events in the consumers of the current operator. If any of the event functions return `false` (i.e. request a reschedule), the report functions also return false, which triggers a fallback to the scheduler. The pseudo code is shown below:

```
OPERATOR::REPORTDATA()
1  result ← true
2  for each c in consumers
3      do if c.DATAEVENT(this) = false
4            then result ← false
5  return result
```

The REPORTENDOFDATA function is nearly identical, it calls ENDOFDATAEVENT instead and clears *activateNext*, as no input is required after all data has been produced.

The DATAEVENT and ENDOFDATAEVENT functions were already discussed in above, the additional return value is used to request a reschedule. The new STARTPUSH method is called by the scheduler if it determines that the operator should start producing data.

To illustrate the mechanism, consider the following simple selection operator:

```
SELECTION::DATAEVENT(source)
1  if the data satisfies the predicate
2     then if REPORTDATA() = false
3             then return false
4  return ACTIVATE(source, input)
```

```
SELECTION::ENDOFDATAEVENT(source)
1  while true
2      do if REPORTENDOFDATA() = false
3            then return false
```

```
SELECTION::STARTPUSH()
1  ACTIVATE(NIL, input)
```

For a selection the STARTPUSH method makes no sense, as it always requires data. It simply activates its input. If it gets data, it checks the predicate, and if this is satisfied, it pushes the data up using REPORTDATA. If one of its consumers requests a reschedule, it drops back to the scheduling component, otherwise it uses ACTIVATE to request more data. If it gets an end of input event, it simply pushes this fact upwards until some operator requests a reschedule.

Binary operators are more complex. We consider here a simple hash join:

```
HASHJOIN::DATAEVENT(source)
1  if source = left
2    then store the data in the left hash table
3         return ACTIVATE(source, left)
4  if source = right
5    then store the data in the right hash table
6         return ACTIVATE(source, right)
```

```
HASHJOIN::ENDOFDATAEVENT(source)
 1  if source = left
 2    then return ACTIVATE(source, right)
 3  if source = right
 4    then activateNext ← NIL
 5         join the hash tables
 6         for all matches
 7             do if REPORTDATA() = false
 8                   then return false
 9         while true
10             do if REPORTENDOFDATA() = false
11                   then return false
```

```
HASHJOIN::STARTPUSH()
1  if already joining
2    then continue joining the hash tables
3         for all matches
4             do if REPORTDATA() = false
5                   then return
6
7  ACTIVATE(NIL, left)
```

The DATAEVENT and ENDOFDATAEVENT methods simply store the incoming data in the hash table and perform the join. The only interesting

detail is that *activateNext* is reset before joining the hash tables, as the operator does not need input anymore. The STARTPUSH method is called when the scheduler has determined that the operator should produce data. Then, there are two cases: Either the operator is already joining the entries of the hash table (in which case it continues to do so), or it requires more input, in which case it activates its left input operator (this is somewhat arbitrary, it could start with the right hand side as well).

The scheduling component required is very simple, it just tries to activate the root of the query plan until the whole result has been produced:

```
SCHEDULER::RUN()
1  while root did not get an end of data event
2     do iter ← root
3        while iter.activateNext ≠ NIL
4           do iter ← iter.activateNext
5        iter.STARTPUSH()
```

This causes the pull-like scheduling: When an operator needs input from another operator, it sets *activateNext* and falls back to the scheduler. This causes an execution order similar to the pull model.

8.2.1 Scheduling

While the scheduling algorithm shown above works, it is very simple. It does not try to satisfy any goal besides correctness, especially it ignores any resource consumption. However, when a query plan includes pipeline breakers, there are usually different scheduling alternatives. For example, the hash join shown above could fall back to the scheduler after getting all data instead of directly producing matches. Then, all input operators for the join could release their resources, the scheduler could activate some other part of the query plan, and later on the join would be activated again to produce the matches for another operator. This might result in a much more conservative resource usage, especially for main memory.

In this work we concentrated on query optimization and, therefore, ignored more advanced scheduling techniques, but if the runtime system uses explicit scheduling anyway, it might be worthwhile to make use of it.

More on scheduling as well as on the implementation of other physical operators can be found in [25].

9 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, 2) situations where DAGs are beneficial are common and not constructed, 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. 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 [25].

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.

9.1 TPC-H

The TPC-H benchmark [37] 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	DAG	DB2
compilation [ms]	10.5	10.6	-
execution [ms]	4793	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. 9.1: 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. DB2 performs between these two, as it reuses intermediate results but uses a temp operator to support multiple reads.

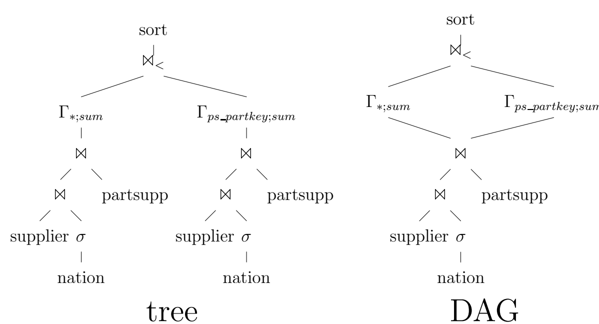

Figure 9.1: Execution plans for TPC-H Query 11

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 [24]. Here, we consider three alternatives: Normal tree construction, DAG construction and DAG construction with rules for magic set transformation enabled. 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. 9.2 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 worth than both DAG plans.

9.2 Bypass Plans

Bypass plans [4] use a DAG structure to efficiently handle disjunctive queries. For example, we assume that we want to find all orders that are either finished or of which at least one item is finished. The SQL representation is shown in Fig. 9.3. Note that for this very simple query, plan generation is basically pointless (compile time $< 0.1ms$), as few alternatives exist. The main work

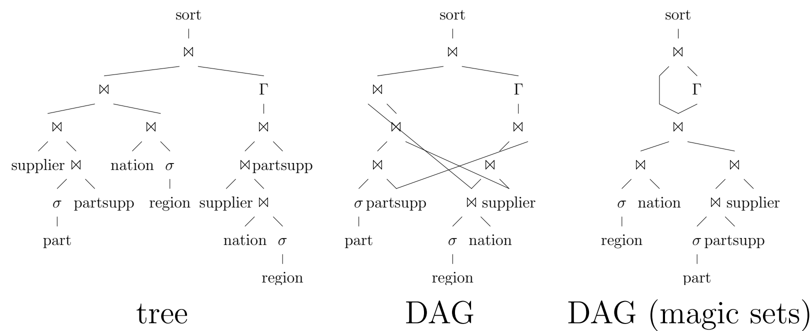

Figure 9.2: Execution plans for TPC-H Query 2

```
select *
from   order
where  o_orderstatus='F' or
       exists(select *
              from   lineitem
              where  l_linestatus='F' and
                     l_orderkey=o_orderkey)
```

Figure 9.3: Disjunctive query

is done during query rewrite. The compile time and runtimes are:

	tree	tree (part.)	DAG	DB2
compilation [ms]	< 0.1	< 0.1	< 0.1	-
execution [ms]	46683	25273	22450	> 5 min

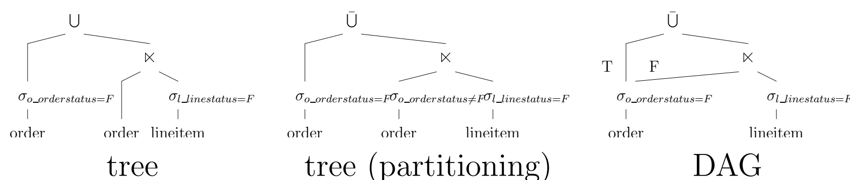

Figure 9.4: Execution plans for Fig. 9.3

The direct translation of the SQL query into an execution plan would require a dependent join. We did not consider this alternative here, as a dependent join of `order` and `lineitem` would be prohibitively expensive. Instead, the `exists` expression is unnested and converted into a semi-join. As the other part of the disjunctive condition also has to be checked, the parts are evaluated independently and the results are combined. The resulting

plan is shown on the left-hand side of Fig. 9.4. The approach has two disadvantages: First, it requires an expensive duplicate elimination and second, it performs the join for tuples that have already qualified. Both problems can be avoided by negating the first condition in the second branch, which guarantees non-overlapping results (second column of Fig. 9.3). This transformation greatly reduces the runtime of the query, but is not trivial to do for all queries: Consider a query with two disjunctive `exists` conditions. In this case, negating the condition in the second branch would be prohibitively expensive. A more flexible approach is to use bypass plans which evaluate the first condition, return all qualifying tuples as result and pass only the other tuples to the second part of the evaluation plan (right-hand side of Fig. 9.3). This is even faster than the second approach and can be used efficiently even for very expensive conditions.

9.3 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.

In fact, DAGs can be considered a clear win over tree-structured query plans. The compile time costs are minimal, the resulting plans are never worse than tree-structured plans, and often the plans are much better than the equivalent tree-structured plans.

9.4 Related Work

We already mentioned related work about different DAG problems in Sec. 2.3. Therefore, we concentrate on related work that concerns generic DAG construction here. Very few papers about generating DAG-structured query graphs exist and the techniques described there have a very limited scope. A Starburst paper mentions that DAG-structured query graphs would be nice, but too complex [17]. A later paper about the DB2 query optimizer [11] explains that DAG-structured query plans are created when considering views, but this solution uses buffering. Buffering means that the database system stores the intermediate result (in this case the view) in a temporary relation, either in main memory or on disk if the relation is too big. This buffering is

expensive, either because it consumes precious main memory which could be used for other operators or – even worse – because the data has to be spooled to disk. Besides, DB2 optimizes the parts above and below the buffering independently, which can lead to suboptimal plans. Although not optimal, this is still a useful optimization and probably state of the art in commercial database management systems. Similar techniques are mentioned in [2, 9].

The Volcano query optimizer [12] 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 [13], 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.

A plan generator that explicitly handles DAG constructions is described in [31]. It uses a two-step approach that reduces the problem of generating DAGs to the problem of generating trees. In the first step, the query is analyzed, and all operators that might be shared are determined. Then, the subset of operators that should be shared is determined (either by exhaustive search or by using some heuristics, which might require running the following steps multiple times) and the shared operators are duplicated. The duplicates provide the same properties as the original operator, but report their costs as zero, so that additional consumers produce no costs. Then, a normal tree-based plan generation is performed. Last, in the result, the duplicates are merged back into the original operator. This results in a cost-based DAG generation, but the approach has some drawbacks. First, it is unfortunate that the selection of shared operators has to be done beforehand, as this requires running the expensive plan generation step multiple times. This selection cannot be omitted easily (e.g. by assuming that all possible operators are shared), as some operators are alternatives (i.e. only one of them makes sense). For example, when considering materialized views, either a scan over the materialized view and a plan to recalculate the view could be used. If the view is read several times, both alternatives are duplicated to enable sharing. By producing too many duplicates, the plan generator will choose only the duplicates without the original operators, as they pretend to cause no costs. Second, the concept that additional consumers cause no costs is only valid if the plan generator does not consider nested loops. If data is read multiple times, the plan generator has to determine the maximum number of reads, and in the model described above the duplicates can be read an arbitrary number of times without causing costs. This severely underestimates costs, especially for very expensive operators. And nested loops cannot be completely avoided, as both dependent joins and joins with

non-equijoin predicates (e.g. `a like b`) use them.

A later publication [32] by the same author considers DAG construction in the context of multi-query optimization: It first constructs the optimial individual execution plans, identifies common fragments in these plans, considers the effect of materializing them and repeats the process. This results in a cost based DAG generation, but separates the optimization itself from DAG construction. Another interesting approach is [5]. 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. A recent publication that considers DAGs for single query optimization is [40] that runs the optimizer repeatedly and uses 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. [5, 38, 1]. While they propose using DAGs, they either produce heuristical solutions or do not support DAGs in the generality of the approach presented here.

9.5 Conclusion

We presented the first plan generator which exploits the full potential of shared subplans by being able to explore the complete search space. Up to now, the state of the art has been to introduce sharing only in special (hand-crafted) cases either before or after, but never during plan generation.

As we have seen, generating DAG-structured query evaluation plans is not easy. First, blindly applying well-known algebraic equivalences in the context of DAG-structured plans easily leads to incorrect plans, i.e. plans that are not equivalent to the original query. To overcome this 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.

To alleviate the problem of generating incorrect plans while still being 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.

Bibliography

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

[2] Don Chamberlin. *A complete guide to DB2 universal database*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[3] Surajit Chaudhuri, Prasanna Ganesan, and Sunita Sarawagi. Factorizing complex predicates in queries to exploit indexes. In *SIGMOD*, pages 361–372, 2003.

[4] Jens Claußen, Alfons Kemper, Guido Moerkotte, Klaus Peithner, and Michael Steinbrunn. Optimization and evaluation of disjunctive queries. *IEEE Trans. Knowl. Data Eng.*, 12(2):238–260, 2000.

[5] Nilesh N. Dalvi, Sumit K. Sanghai, Prasan Roy, and S. Sudarshan. Pipelining in multi-query optimization. In *PODS*, 2001.

[6] Leonidas Fegaras. The joy of sax. In *XIME-P*, pages 61–66, 2004.

[7] Sheldon J. Finkelstein. Common subexpression analysis in database applications. In *SIGMOD*, pages 235–245, 1982.

[8] Cesar Galindo-Legaria. personal communication.

[9] César A. Galindo-Legaria and Milind Joshi. Orthogonal optimization of subqueries and aggregation. In *SIGMOD*, 2001.

[10] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database System Implementation*. Prentice-Hall, Inc., 1999.

[11] 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.

[12] Goetz Graefe. Encapsulation of parallelism in the volcano query processing system. In *SIGMOD*, pages 102–111. ACM Press, 1990.

[13] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[14] Goetz Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994.

[15] Goetz Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[16] Goetz Graefe and William J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *ICDE*, pages 209–218. IEEE Computer Society, 1993.

[17] Laura M. Haas, Johann Christoph Freytag, Guy M. Lohman, and Hamid Pirahesh. Extensible query processing in starburst. In *SIGMOD*, pages 377–388. ACM Press, 1989.

[18] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. Qpipe: A simultaneously pipelined relational query engine. In *SIGMOD*, pages 383–394, 2005.

[19] Ravi Krishnamurthy, Haran Boral, and Carlo Zaniolo. Optimization of nonrecursive queries. In *VLDB'86*, pages 128–137. Morgan Kaufmann, 1986.

[20] Guy Lohman. personal communication.

[21] Guy M. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *SIGMOD*, pages 18–27. ACM Press, 1988.

[22] Raymond A. Lorie. Xrm - an extended (n-ary) relational memory. *IBM Research Report*, G320-2096, 1974.

[23] David Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.

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

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

[26] Thomas Neumann, Sven Helmer, and Guido Moerkotte. On the optimal ordering of maps and selections under factorization. In *ICDE*, pages 490–501, 2005.

[27] Thomas Neumann and Guido Moerkotte. A combined framework for grouping and order optimization. In *VLDB*, pages 960–971. IEEE Computer Society, 2004.

[28] Thomas Neumann and Guido Moerkotte. An efficient framework for order optimization. In *ICDE*, pages 461–472. IEEE Computer Society, 2004.

[29] 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.

[30] Arnon Rosenthal and César A. Galindo-Legaria. Query graphs, implementing trees, and freely-reorderable outerjoins. In *SIGMOD*, pages 291–299. ACM Press, 1990.

[31] Prasan Roy. Optimization of dag-structured query evaluation plans. M.tech. thesis, Dept. of Computer Science and Engineering, IIT-Bombay, January 1998.

[32] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *SIGMOD*, pages 249–260. ACM, 2000.

[33] 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 Philip A. Bernstein, editor, *SIGMOD*, pages 23–34. ACM, 1979.

[34] Timos K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, 1988.

[35] David E. Simmen, Eugene J. Shekita, and Timothy Malkemus. Fundamental techniques for order optimization. In H. V. Jagadish and Inderpal Singh Mumick, editors, *SIGMOD*, pages 57–67. ACM Press, 1996.

[36] Michael Steinbrunn, Klaus Peithner, Guido Moerkotte, and Alfons Kemper. Bypassing joins in disjunctive queries. In Umeshwar Dayal, Peter

M. D. Gray, and Shojiro Nishio, editors, *VLDB*, pages 228–238. Morgan Kaufmann, 1995.

[37] Transaction Processing Performance Council, 777 N. First Street, Suite 600, San Jose, CA, USA. *TPC Benchmark H*, 2003. Revision 2.1.0. http://www.tpc.org.

[38] Satyanarayana R. Valluri, Soujanya Vadapalli, and Kamalakar Karlapalem. Sprinkling selections over join dags for efficient query optimization. *CoRR*, cs.DB/0202035, 2002.

[39] Markos Zaharioudakis, Roberta Cochrane, George Lapis, Hamid Pirahesh, and Monica Urata. Answering complex sql queries using automatic summary tables. In *SIGMOD*, pages 105–116, 2000.

[40] 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.

Below you find a list of the most recent technical reports of the Max-Planck-Institut für Informatik. They are available by anonymous ftp from `ftp.mpi-sb.mpg.de` under the directory `pub/papers/reports`. Most of the reports are also accessible via WWW using the URL `http://www.mpi-sb.mpg.de`. If you have any questions concerning ftp or WWW access, please contact `reports@mpi-sb.mpg.de`. Paper copies (which are not necessarily free of charge) can be ordered either by regular mail or by e-mail at the address below.

Max-Planck-Institut für Informatik
Library
attn. Anja Becker
Stuhlsatzenhausweg 85
66123 Saarbrücken
GERMANY
e-mail: `library@mpi-sb.mpg.de`

	F.M. Suchanek, G. Kasneci, G. Weikum	Yago: A Core of Semantic Knowledge
MPI-I-2006-RG1-001	S. Hirth, C. Karl, C. Weidenbach	Automatic Infrastructure for ..... Analysis
MPI-I-2006-5-005	R. Angelova, S. Siersdorfer	A Neighborhood-Based Approach for Clustering of Linked Document Collections
MPI-I-2006-5-004	F. Suchanek, G. Ifrim, G. Weikum	Combining Linguistic and Statistical Analysis to Extract Relations from Web Documents
MPI-I-2006-5-003	V. Scholz, M. Magnor	Garment Texture Editing in Monocular Video Sequences based on Color-Coded Printing Patterns
MPI-I-2006-5-002	H. Bast, D. Majumdar, R. Schenkel, M. Theobald, G. Weikum	IO-Top-k: Index-access Optimized Top-k Query Processing
MPI-I-2006-5-001	M. Bender, S. Michel, G. Weikum, P. Triantafilou	Overlap-Aware Global df Estimation in Distributed Information Retrieval Systems
MPI-I-2006-4-010	A. Belyaev, T. Langer, H. Seidel	Mean Value Coordinates for Arbitrary Spherical Polygons and Polyhedra in R^3
MPI-I-2006-4-009	J. Gall, J. Potthoff, B. Rosenhahn , C. Schnoerr, H. Seidel	Interacting and Annealing Particle Filters: Mathematics and a Recipe for Applications
MPI-I-2006-4-008	I. Albrecht, M. Kipp, M. Neff, H. Seidel	Gesture Modeling and Animation by Imitation
MPI-I-2006-4-007	O. Schall, A. Belyaev, H. Seidel	Feature-preserving Non-local Denoising of Static and Time-varying Range Data
MPI-I-2006-4-006	C. Theobalt, N. Ahmed, H. Lensch, M. Magnor, H. Seidel	Enhanced Dynamic Reflectometry for Relightable Free-Viewpoint Video
MPI-I-2006-4-005	A. Belyaev, H. Seidel, S. Yoshizawa	Skeleton-driven Laplacian Mesh Deformations
MPI-I-2006-4-004	V. Havran, R. Herzog, H. Seidel	On Fast Construction of Spatial Hierarchies for Ray Tracing
MPI-I-2006-4-003	E. de Aguiar, R. Zayer, C. Theobalt, M. Magnor, H. Seidel	A Framework for Natural Animation of Digitized Models
MPI-I-2006-4-002	G. Ziegler, A. Tevs, C. Theobalt, H. Seidel	GPU Point List Generation through Histogram Pyramids
MPI-I-2006-4-001	R. Mantiuk	?
MPI-I-2006-2-001	T. Wies, V. Kuncak, K. Zee, A. Podelski, M. Rinard	On Verifying Complex Properties using Symbolic Shape Analysis
MPI-I-2006-1-007	I. Weber	?
MPI-I-2006-1-006	M. Kerber	Division-Free Computation of Subresultants Using Bezout Matrices
MPI-I-2006-1-004	E. de Aguiar	?
MPI-I-2006-1-001	M. Dimitrios	?
MPI-I-2005-5-002	S. Siersdorfer, G. Weikum	Automated Retraining Methods for Document Classification and their Parameter Tuning
MPI-I-2005-4-006	C. Fuchs, M. Goesele, T. Chen, H. Seidel	An Emperical Model for Heterogeneous Translucent Objects

MPI-I-2005-4-005	G. Krawczyk, M. Goesele, H. Seidel	Photometric Calibration of High Dynamic Range Cameras
MPI-I-2005-4-004	C. Theobalt, N. Ahmed, E. De Aguiar, G. Ziegler, H. Lensch, M.A.,. Magnor, H. Seidel	Joint Motion and Reflectance Capture for Creating Relightable 3D Videos
MPI-I-2005-4-003	T. Langer, A.G. Belyaev, H. Seidel	Analysis and Design of Discrete Normals and Curvatures
MPI-I-2005-4-002	O. Schall, A. Belyaev, H. Seidel	Sparse Meshing of Uncertain and Noisy Surface Scattered Data
MPI-I-2005-4-001	M. Fuchs, V. Blanz, H. Lensch, H. Seidel	Reflectance from Images: A Model-Based Approach for Human Faces
MPI-I-2005-2-004	Y. Kazakov	A Framework of Refutational Theorem Proving for Saturation-Based Decision Procedures
MPI-I-2005-2-003	H.d. Nivelle	Using Resolution as a Decision Procedure
MPI-I-2005-2-002	P. Maier, W. Charatonik, L. Georgieva	Bounded Model Checking of Pointer Programs
MPI-I-2005-2-001	J. Hoffmann, C. Gomes, B. Selman	Bottleneck Behavior in CNF Formulas
MPI-I-2005-1-008	C. Gotsman, K. Kaligosi, K. Mehlhorn, D. Michail, E. Pyrga	Cycle Bases of Graphs and Sampled Manifolds
MPI-I-2005-1-008	D. Michail	?
MPI-I-2005-1-007	I. Katriel, M. Kutz	A Faster Algorithm for Computing a Longest Common Increasing Subsequence
MPI-I-2005-1-003	S. Baswana, K. Telikepalli	Improved Algorithms for All-Pairs Approximate Shortest Paths in Weighted Graphs
MPI-I-2005-1-002	I. Katriel, M. Kutz, M. Skutella	Reachability Substitutes for Planar Digraphs
MPI-I-2005-1-001	D. Michail	Rank-Maximal through Maximum Weight Matchings
MPI-I-2004-NWG3-001	M. Magnor	Axisymmetric Reconstruction and 3D Visualization of Bipolar Planetary Nebulae
MPI-I-2004-NWG1-001	B. Blanchet	Automatic Proof of Strong Secrecy for Security Protocols
MPI-I-2004-5-001	S. Siersdorfer, S. Sizov, G. Weikum	Goal-oriented Methods and Meta Methods for Document Classification and their Parameter Tuning
MPI-I-2004-4-006	K. Dmitriev, V. Havran, H. Seidel	Faster Ray Tracing with SIMD Shaft Culling
MPI-I-2004-4-005	I.P. Ivrissimtzis, W.-. Jeong, S. Lee, Y.a. Lee, H.-. Seidel	Neural Meshes: Surface Reconstruction with a Learning Algorithm
MPI-I-2004-4-004	R. Zayer, C. Rssl, H. Seidel	r-Adaptive Parameterization of Surfaces
MPI-I-2004-4-003	Y. Ohtake, A. Belyaev, H. Seidel	3D Scattered Data Interpolation and Approximation with Multilevel Compactly Supported RBFs
MPI-I-2004-4-002	Y. Ohtake, A. Belyaev, H. Seidel	Quadric-Based Mesh Reconstruction from Scattered Data
MPI-I-2004-4-001	J. Haber, C. Schmitt, M. Koster, H. Seidel	Modeling Hair using a Wisp Hair Model
MPI-I-2004-2-007	S. Wagner	Summaries for While Programs with Recursion
MPI-I-2004-2-002	P. Maier	Intuitionistic LTL and a New Characterization of Safety and Liveness
MPI-I-2004-2-001	H. de Nivelle, Y. Kazakov	Resolution Decision Procedures for the Guarded Fragment with Transitive Guards
MPI-I-2004-1-006	L.S. Chandran, N. Sivadasan	On the Hadwiger's Conjecture for Graph Products
MPI-I-2004-1-005	S. Schmitt, L. Fousse	A comparison of polynomial evaluation schemes
MPI-I-2004-1-004	N. Sivadasan, P. Sanders, M. Skutella	Online Scheduling with Bounded Migration
MPI-I-2004-1-003	I. Katriel	On Algorithms for Online Topological Ordering and Sorting
MPI-I-2004-1-002	P. Sanders, S. Pettie	A Simpler Linear Time 2/3 - epsilon Approximation for Maximum Weight Matching