Extending Dynamic-Programming-Based Plan Generators: Beyond Pure Enumeration

Inauguraldissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
der Universität Mannheim

vorgelegt von

Marius Eich, M.Sc.
aus Karlsruhe

Mannheim, 2017

Dekan:	Professor Dr. Heinz Jürgen Müller, Universität Mannheim
Referent:	Professor Dr. Guido Moerkotte, Universität Mannheim
Korreferent:	Professor Dr. Thomas Neumann, Technische Universität München

Tag der mündlichen Prüfung: 11. September 2017

Abstract

The query optimizer plays an important role in a database management system supporting a declarative query language, such as SQL. One of its central components is the plan generator, which is responsible for determining the optimal join order of a query. Plan generators based on dynamic programming have been known for several decades. However, some significant progress in this field has only been made recently. This includes the emergence of highly efficient enumeration algorithms and the ability to optimize a wide range of queries by supporting complex join predicates. This thesis builds upon the recent advancements by providing a framework for extending the aforementioned algorithms. To this end, a modular design is proposed that allows for the exchange of individual parts of the plan generator, thus enabling the implementor to add new features at will. This is demonstrated by taking the example of two previously unsolved problems, namely the correct and complete reordering of different types of join operators as well as the efficient reordering of join operators and grouping operators.

Zusammenfassung

Der Anfrageoptimierer spielt eine wichtige Rolle in Datenbanksystemen, die deklarative Anfragesprachen wie SQL unterstützen. Eine der zentralen Komponenten des Anfrageoptimierers ist der Plangenerator, der für die Optimierung der Join-Reihenfolge zuständig ist. Plangeneratoren, die auf dynamischer Programmierung basieren, gibt es bereits seit einigen Jahrzehnten. Einige entscheidende Durchbrüche auf diesem Gebiet wurden allerdings erst in den letzten Jahren gemacht. Beispiele hierfür sind effiziente Aufzählungsalgorithmen für Teilpläne und die Unterstützung von komplexen Join-Prädikaten. Aufbauend auf diesen neuen Erkenntnissen beschreibt die vorliegende Dissertation einen Ansatz zur Erweiterung derartiger Plangeneratoren. Dazu wird zunächst ein modulares Design zum einfachen Austausch von Teilen des Plangenerators beschrieben, welches das Hinzufügen neuer Funktionen erleichtert. Dies wird demonstriert am Beispiel zweier bisher ungelöster Probleme, nämlich der korrekten und vollständigen Optimierung der Reihenfolge verschiedenartiger Join-Operatoren und der Optimierung der Reihenfolge von Join- und Gruppierungsoperatoren.

Contents

List of Tables

List of Figures

1 Introduction

Query languages such as SQL are declarative. They are meant to describe a certain query result desired by the user without specifying how it should be obtained. Typically, there are many equivalent execution orders for the operations necessary to answer a query. They all lead to the correct result, but they differ largely in the resources they consume in doing so.

The task of query optimization is to find the best *query evaluation plan* for a given query according to some metric. A query evaluation plan precisely defines how a query should be processed by specifying the order in which the necessary operations are executed and how. In other words, a query evaluation plan is an operator tree with physical algebraic operators as nodes. Physical in the sense that these operators are not only logical representations of algebraic operators, but they comprise the information needed to apply them including a specific implementation.

Query optimization has been a topic of research for many decades. However, some significant progress in this field has only been made in the last decade. This includes the development of highly efficient algorithms for solving one of the most important and oldest problems in query optimization, namely the optimization of the join order [10, 11, 12, 13, 14, 30, 32]. This is the responsibility of the so-called *plan generator*.

This thesis builds upon the recent advancements by providing a framework for extending the aforementioned algorithms. To this end, a modular design is proposed that allows for the exchange of individual parts of the plan generator, thus enabling the implementor to add new features at will. This is demonstrated by taking the example of two previously unsolved problems, namely the correct and complete reordering of different types of join operators as well as the efficient reordering of join operators and grouping operators.

The thesis is structured as follows: Chapter 2 contains some preliminaries consisting of a brief overview of query optimization in general, the introduction to the notation used throughout this work and the description of some basic data structures such as query graphs and join trees. Chapter 3 presents the building blocks of current plan generators. The modular architecture described in this chapter is one of the contributions of this thesis. The problem of reordering non-inner joins is covered in Chapter 4 by proposing a novel approach that allows the enumeration of all valid and only valid query plans containing join operators that are not freely reorderable. In Chapter 5 we turn our attention to the problem of optimizing the execution order of join operators and grouping operators. The proposed solution serves as an example of how the functionality of the plan generator can be extended while keeping its complexity in check. Some examples of similar problems and solution approaches can be found in Chapter 6 along with a conclusive summary of this work.

2 Preliminaries

This chapter summarizes the basics of query optimization, including common data structures such as query graphs and join trees. We begin with a cursory overview of query optimization in general.

2.1 Query Optimization in a Nutshell

Conceptually, query optimization is often split into two phases, namely a logical and a physical phase. In the logical phase the optimal ordering of operators is determined by exploiting certain algebraic equivalences. In the physical phase the resulting operator tree is annotated with additional information such as the implementations of the contained operators. While this separation helps to better understand these two aspects of query optimization, it is typically not applied in practice. That is because decisions on the logical and physical level often influence each other. For example, the optimal join order of a query might depend on the join implementations or indices available in the system. But in theory, optimizing the join order would be attributed to logical query optimization, while the questions whether or not a certain relation should be accessed using an index and which join implementation to use for a certain operator are clearly seen as part of physical optimization.

The first step of transforming a declarative query into an executable query evaluation plan typically is to rewrite the query in some way. For example, the rewrite can include unnesting of nested queries or pushing selections down. Then, the rewritten query is translated into an internal representation, which is passed to a plan generator. The plan generator finally turns it into an optimal query evaluation plan.

There are two basic approaches to plan generation, namely the transformation-based and the generative approach. The former transforms one plan into another equivalent plan by applying certain transformation rules. These can be derived from algebraic equivalences. The generative approach works by building query plans from smaller subplans, adding one algebraic operator after the other until a complete plan is assembled. This means that an executable plan is available only after all necessary operators have been added to the plan.

One disadvantage of transformation-based plan generators is that they typically produce duplicate plans because often the same plan can be derived from other plans through different sequences of transformations. In general, the generative approach is more efficient but also less extendable. Introducing new transformation rules into a transformation-based plan generator is typically much easier than incorporating them in a generative plan generator. One prominent example of a transformation-based plan generator is the EXODUS optimizer [20]. A generative query optimizer was first described as part of the

System R research prototype at IBM [40]. We will take a more detailed look at it in Chapter 3.

Generative plan generators can be further categorized into those based on *memoization* and those based on *dynamic programming* (DP). A memoization-based plan generator works recursively by partitioning every set S of relations into subsets and considering all join trees between such a subset S_1 and its complement S_2. The best tree for each set is stored in a solution table, because the same subset can be contained in more than one relation set. In the end, an optimal solution for the whole relation set can be looked up in the table.

Dynamic programming, on the other hand, works bottom-up by iteratively constructing larger join trees from smaller join trees. Again, the best tree for each subproblem is stored in a so-called *DP table* and looked up later to build the overall solution. In this thesis the focus lies on DP-based plan generators. The problems we will discuss subsequently are unique to generative plan generators, but they occur regardless of whether the plan generator works bottom-up or top-down. Consequently, the solutions to these problems are mostly applicable to both memoization-based and DP-based plan generators with only slight adaptations.

Chapter 3 provides several examples of DP-based plan generators and describes the different components they typically consist of. What makes generative plan generators so efficient is the fact that the problem of optimizing the join order falls into a category of problems that can be solved using *Bellman's Principle of Optimality*. It can be stated as follows:

> Let T be an optimal join tree for relations $R_1, \ldots, R_n$. Then, every subtree S of T must be an optimal join tree for the relations it contains.

In other words, an optimal solution containing all relations $R_1, \ldots, R_n$ can be constructed by combining optimal solutions for the contained subsets of relations, which is exactly what dynamic programming and memoization do. As we will see in Section 2.4, Bellman's Principle of Optimality no longer holds when plan properties are taken into account.

2.2 Relational Algebra

Since SQL is declarative and therefore not suitable for describing a query evaluation plan, queries have to be translated into an imperative language before they can be optimized. One such language is relational algebra. It consists of a set of operators with relations as their input and output. These operators are typically arranged in an operator tree, which indicates a distinct ordering of the contained operators and thereby specifies an evaluation strategy for the corresponding query. According to the set-based definition of relations, relational algebra is generally defined with set semantics, i.e., input and output relations are expected to be duplicate-free. We deviate from this convention and define our algebraic operators on bags instead of sets, meaning that by our definition,

relations can contain duplicates. With this, we follow SQL's bag-oriented definition of relations. If necessary, we explicitly state whether a certain operator produces duplicates in its output, or not.

2.2.1 Algebraic Operators

The first operator to consider is the selection operator $\sigma_p(e)$. It returns all tuples resulting from expression e that satisfy the selection predicate p. Formally, it is defined as

$$\sigma_p(e) := \{x | x \in e, p(x)\}.$$

We continue with the duplicate-removing projection, which we denote by $\Pi^D_A(e)$ for a set of Attriutes A and an algebraic expression e. The resulting relation only contains values for those attributes from e that are contained in A and no duplicate values. Analogously, we denote the duplicate-preserving projection by $\Pi_A(e)$.

The map operator χ extends every input tuple by new attributes. The values of the new attributes are determined by expressions of any type. For example, this could be scalar functions. In the following definitions we denote tuple concatenation by $\circ$:

$$\chi_{a_1:e_1,\ldots,a_n:e_n}(e) := \{t \circ [a_1 : e_1(t), \ldots, a_n : e_n(t)] | t \in e\}$$

We can now move on to the grouping operator Γ. It can be defined as

$$\begin{aligned}\Gamma_{\theta G;g:f}(e) \quad := \quad & \{y \circ [g : x] \mid y \in \Pi^D_G(e), \\ & x = f(\{z | z \in e,\ z.G\ \theta\ y.G\})\}\end{aligned}$$

for a set of grouping attributes G, a single attribute g, an aggregate function f and a comparison operator $\theta \in \{=, \neq, \leq, \geq, <, >\}$.

The function f is applied to groups of tuples taken from $\Pi^D_G(e)$. The groups are determined by the comparison operator θ. Afterwards, a new tuple consisting of the grouping attributes' values and an attribute g holding the corresponding value returned by the aggregate function f is constructed.

The grouping operator can also introduce more than one new attribute by applying several aggregate functions. We define

$$\begin{aligned}\Gamma_{\theta G;b_1:f_1,\ldots,b_k:f_k}(e) := \quad & \{y \circ [b_1 : x_1, \ldots, b_k : x_k] \mid y \in \Pi_G(e), \\ & x_i = f_i(\{z | z \in e,\ z.G\ \theta\ y.G\})\},\end{aligned}$$

where the attribute values $b_1, \ldots, b_k$ are obtained by applying the aggregation vector $F = (f_1, \ldots, f_k)$, consisting of k aggregate functions, to the tuples grouped according to θ. The grouping criterion may also be defined on several attributes. If all θ equal '=', we abbreviate $\Gamma_{=G;g:f}$ by $\Gamma_{G;g:f}$.

Next, we go over the set of join operators consisting of (inner) join ($\bowtie$), left semijoin ($\ltimes$), left antijoin ($\triangleright$), left outerjoin ($\leftouterjoin$), full outerjoin ($\fullouterjoin$), and left groupjoin ($\leftgroupjoin$). Subsequently, we will use the name LOP (short for "left operators") when referring to this operator set. Most of the operators are rather standard. Nonetheless, their definitions are provided in Figure 2.1. There, $\perp_A$

$$
\begin{aligned}
e_1 \times e_2 &:= \{r \circ s | r \in e_1, s \in e_2\} && (2.1)\\
e_1 \bowtie_p e_2 &:= \{r \circ s | r \in e_1, s \in e_2, p(r,s)\} && (2.2)\\
e_1 \ltimes_p e_2 &:= \{r | r \in e_1, \exists s \in e_2, p(r,s)\} && (2.3)\\
e_1 \triangleright_p e_2 &:= \{r | r \in e_1, \nexists s \in e_2, p(r,s)\} && (2.4)\\
e_1 \rtimes_p e_2 &:= (e_1 \bowtie_p e_2) \cup ((e_1 \triangleright_p e_2) \times \{\bot_{\mathcal{A}(e_2)}\} && (2.5)\\
e_1 \rtimes\!\ltimes_p e_2 &:= (e_1 \bowtie_p e_2) \\
&\quad \cup ((e_1 \triangleright_p e_2) \times \{\bot_{\mathcal{A}(e_2)}\} \\
&\quad \cup (\{\bot_{\mathcal{A}(e_1)}\} \times (e_2 \triangleright_p e_1)) && (2.6)\\
e_1 \bowtie\!\!\!\!\Gamma_{p;g:f} e_2 &:= \{r \circ [g:G] | r \in e_1, G = f(\{s | s \in e_2, p(r,s)\})\} && (2.7)
\end{aligned}
$$

Figure 2.1: Join operators

denotes a tuple containing only null values in all attributes contained in the attribute set A.

The last row defines the left groupjoin $\bowtie\!\!\!\!\Gamma$, which was introduced by von Bültzingsloewen [43]. First, for a given tuple $t_1 \in e_1$, it determines the sets of all join partners in e_2 according to the join predicate p. Then, it applies the aggregate function f to these tuples and extends t_1 by a new attribute g containing the result of this aggregation. Figure 2.2 provides examples for all operators.

2.2.2 Predicates and Expressions

When dealing with predicates or other expressions such as aggregation vectors, we use the following notation for accessing relations, attributes and operators referenced therein.

We denote by $\mathcal{A}(e)$ the set of attributes or variables provided by some expression e and by $\mathcal{F}(e)$ the set of free attributes or variables in some expression e. For example, for a join predicate p with $p \equiv R.a + S.b = S.c + T.d$ we get $\mathcal{F}(p) = \{R.a, S.b, S.c, T.d\}$.

For a binary operator $\circ$, left($\circ$) denotes the left subtree of $\circ$ and right($\circ$) denotes the right subtree of $\circ$.

For a set of attributes A, $\mathcal{T}(A)$ denotes the set of tables to which these attributes belong. We abbreviate $\mathcal{T}(\mathcal{F}(e))$ by $\mathcal{F}_{\mathcal{T}}(e)$. For p defined as above we get $\mathcal{F}_{\mathcal{T}}(e) = \mathcal{T}(\mathcal{F}(e)) = \{R, S, T\}$.

Let $\circ$ be an operator in the initial operator tree. We denote by left($\circ$) (right($\circ$)) its left (right) child. STO($\circ$) denotes the operators contained in the operator subtree rooted at $\circ$. $\mathcal{T}(\circ)$ denotes the set of tables contained in the subtree rooted at $\circ$.

Two important properties of join predicates are *null-rejection* and *degeneration*.

Definition 1. *A predicate is null-rejecting for a set of attributes A if it evaluates to false or unknown on every tuple in which all attributes in A are null.*

R_1

a	b	c
0	0	1
1	0	1
2	1	3
3	2	3

R_2

d	e	f
0	0	1
1	1	1
2	2	1
3	4	2

$R_1 \bowtie_{R_1.b=R_2.d} R_2$

a	b	c	d	e	f
0	0	1	0	0	1
1	0	1	0	0	1
2	1	3	1	1	1
3	2	3	2	2	1

$R_1 \triangleright_{R_1.a=R_2.e} R_2$

a	b	c
3	2	3

$R_1 \ltimes_{R_1.b=R_2.d} R_2$

a	b	c
0	0	1
1	0	1
2	1	3
3	2	3

$R_1 \; ⟕_{R_1.a=R_2.f;g:sum(R_2.f)} \; R_2$

a	b	c	g
1	0	1	3
2	1	3	2

Figure 2.2: Examples of different join operators

For example, the predicate $a =$ null is null-rejecting for $\{a\}$. The predicate "a is null", on the other hand, is not null-rejecting for $\{a\}$. Some common synonyms for null-rejecting are *null-intolerant*, *strong*, and *strict*. Subsequently, we denote by $NR(p)$ the set of attributes p rejects nulls for.

Degenerate join predicates are those that do not reference tables from all join arguments.

Definition 2. *Let p be a predicate associated with a binary operator $\circ$ and $\mathcal{F}_\mathcal{T}(p)$ the tables referenced by p. Then, p is called degenerate if $\mathcal{T}(left(\circ)) \cap \mathcal{F}_\mathcal{T}(p) = \emptyset \; \vee \; \mathcal{T}(right(\circ)) \cap \mathcal{F}_\mathcal{T}(p) = \emptyset$ holds.*

For example, in $\bowtie_{true}$ the predicate *true* is degenerate. Moreover, the expression is equivalent to a cross product.

2.3 Query Graphs and Join Trees

The query graph is a convenient way of representing the structure of a query. All plan generators we will discuss subsequently expect a query graph as their input and produce an algebraic operator tree as their output. Before we can define query graphs, we have to define hypergraphs:

Definition 3. *A hypergraph is a pair $H = (V, E)$ such that*

1. *V is a non-empty set of nodes and*
2. *E is a set of hyperedges, where a hyperedge is an unordered pair (u,v) of non-empty subsets of V ($u \subset V$ and $v \subset V$) with the additional condition that $u \cap v = \emptyset$.*

A hyperedge (u, v) is simple if $|u| = |v| = 1$. A hypergraph is simple if all its hyperedges are simple.

With this, a query graph is defined as follows:

Definition 4. *The query graph for a query Q is a hypergraph $H = (V, E)$, such that*

1. *V represents the set $\mathcal{R} = \{R_1, \ldots, R_n\}$ of relations referenced in Q and*
2. *for every $(u, v) \in E$, u and v represent (sets of) relations referenced by a join predicate in Q.*

In other words, for every join predicate in Q, the query graph contains an edge (u, v) with u/v representing the relations referenced on the left/right side of the predicate. This definition makes clear why the query graph is a hypergraph in general: in SQL a join predicate can reference more than two relations. For an example, consider the following predicate referencing relations R_0, R_1 and R_2:

$$R_0.a + R_1.b = R_2.c$$

Queries can be classified by the shape of their query graph. Typically, at least four different shapes are distinguished:

- chain queries, where the query graph is acyclic and every node has a link to at most two other nodes,
- star queries, where all nodes are linked to one hub,
- cycle queries, where the query graph contains a cycle and
- clique queries, where every node is linked to every other node.

Figure 2.3 provides an example of each class. Hybrid forms of every type are conceivable and often seen in practice.

The task of the plan generator is to find an optimal join tree for a given query. In contrast to a query graph, a join tree represents an execution order of joins. A join tree typically is a binary tree whose inner nodes represent the join operators. In our case these can be any of the join operators introduced in Section 2.2. The leaf nodes of a join tree represent the relations referenced in the corresponding query.

Much like query graphs, join trees can be distinguished by their shape. Figure 2.4 provides some examples of different shapes. The output of the plan generator is sometimes deliberately restricted to join trees of a certain shape

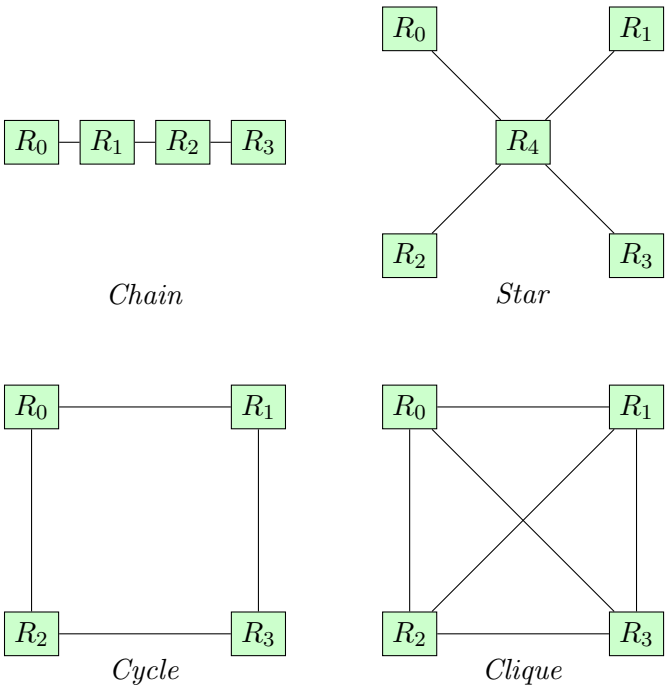


Figure 2.3: Different shapes of query graphs

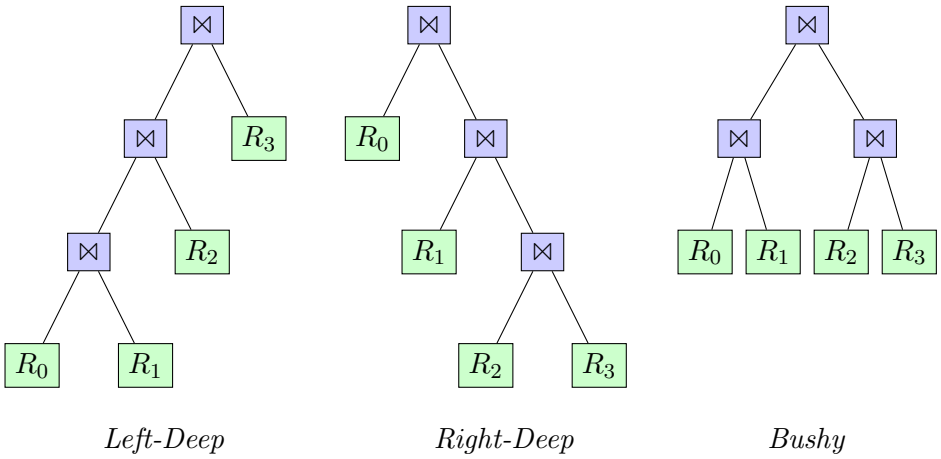


Figure 2.4: Different shapes of join trees

to reduce the size of the search space the plan generator has to cover. Clearly, this means that the optimal join tree will not be found if it is a bushy tree, but the plan generator is only capable of producing left-deep trees. We will return to this issue again in Section 3.1

For now, it is most important to notice that the number of possible join trees for a query is closely related to the shape of the query graph if the plan generator does not introduce cross products. This can easily be seen by considering that the edges in the query graph determine which sets of relations can be joined without applying a cross product. The more edges there are in the graph, the

```
select *
from
R join S on r1 = s2
 join T on s1 = t1;
```

Figure 2.5: Example query with simple predicates

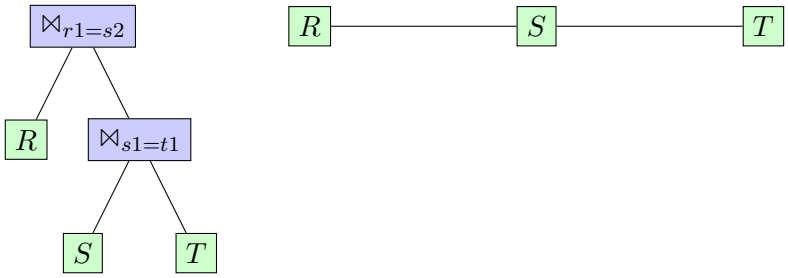


Figure 2.6: Join tree and query graph for query in Figure 2.5

more possible join trees there are for the respective query. Thus, for a given number of relations, there are more possible join trees for clique queries than for chain queries. This makes the problem of finding the optimal join tree more complex for the former class of queries. If cross products are allowed, we can join any pair of relations, effectively turning every query into a clique query.

For an example, Figure 2.5 shows a small SQL query joining the three relations R, S and T. Figure 2.6 shows the corresponding join tree and query graph. In the example all attributes belong to the relation with the same letter, i.e., $r1$ belongs to R, $s1$ belongs to S and so on. The query graphs we have seen so far are all examples of simple query graphs. As we said earlier, this can be seen as a special case, since in general, the query graph is a hypergraph. One possible source of hyperedges are complex predicates. Figure 2.7 shows a slightly modified version of our example query. It contains a complex predicate $r1 + s2 = t1$. Figure 2.8 shows the corresponding join tree and query graph with one complex edge.

The presence of hyperedges reduces the number of connected components in the query graph and thereby also reduces the number of valid join trees for the given query. Thus, hyperedges shrink the search space of the plan generator, facilitating the task of finding an optimal plan for the query. In our example graph the pairs of relation sets $(\{R\}, \{T\})$ and $(\{S\}, \{T\})$ are not

```
select *
from
R join S on r2 = s1
 join T on r1 + s2 = t1;
```

Figure 2.7: Example query with complex predicate

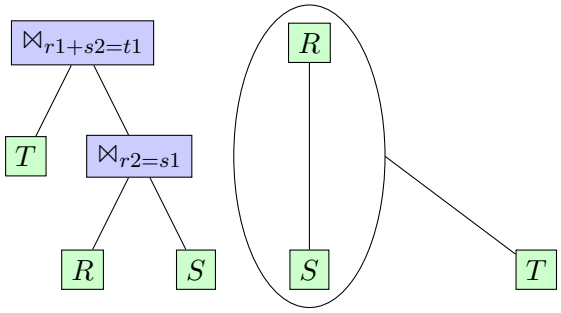


Figure 2.8: Join tree and query graph for query in Figure 2.7

connected. The only edge connecting T to the rest of the graph is the complex edge $(\{R, S\}, \{T\})$. In the figure this is symbolized by the ellipsis surrounding R and S, from which an edge goes out to T. The join tree shown in the figure illustrates the search space reduction resulting from the complex predicate: clearly, we can no longer join S with T before joining the result with R, as was the case in the previous example. This is because both R and S have to be available before the complex predicate can be evaluated.

Several publications deal with the influence of the query graph shape on the complexity of plan enumeration [32, 34]. The effect of hyperedges on the search space size is thoroughly described in a paper by Moerkotte and Neumann [30]. There and in Chapter 4 of this work, it is shown how this effect can be exploited to allow for an efficient reordering of non-inner joins.

When dealing with query graphs, one concept of particular interest is that of *connected-subgraph–complement-pairs*, or csg-cmp-pairs. Before we can define them, we need to define subgraphs and connectedness:

Definition 5. *Let $H = (V, E)$ be a hypergraph and $V' \subseteq V$ a subset of nodes. The node-induced subgraph $H|_{V'}$ of H is defined as $H|_{V'} = (V', E')$ with $E' = \{(u, v) | (u, v) \in E, u \subseteq V', v \subseteq V'\}$.*

Definition 6. *Let $H = (V, E)$ be a hypergraph. H is connected if $|V| = 1$ or if there exists a partitioning V', V'' and a hyperedge $(u, v) \in E$ such that $u \subseteq V', v \subseteq V''$ and both $H|_{V'}$ and $H|_{V''}$ are connected.*

With this, we are ready to define csg-cmp-pairs as follows:

Definition 7. *Let $H = (V, E)$ be a hypergraph and S_1, S_2 two non-empty subsets of V with $S_1 \cap S_2 = \emptyset$. Then, the pair (S_1, S_2) is called a csg-cmp-pair if the following conditions hold:*

1. *S_1 and S_2 induce a connected subgraph of H and*
2. *there exists a hyperedge $(u, v) \in E$ such that $u \subseteq S_1$ and $v \subseteq S_2$.*

The number of csg-cmp-pairs for a given query graph is equal to the number of join (sub-)trees that have to be considered to find an optimal join tree for the corresponding query. Thereby, it defines a lower bound for the complexity of a plan generator that explores the complete search space. Subsequently, we will often use the shorter term *ccp* for csg-cmp-pair.

2.4 Plan Classes, Cost and Properties

A join tree, as defined in the previous section, is an incomplete logical representation of a query evaluation plan. A complete plan typically contains more information than a join tree. In essence, a query plan is an operator tree not only containing join operators, but physical algebraic operators of all kinds as nodes. However, as the join order is in many cases the determining factor for the runtime of a query, a join tree describes an important part of the query plan's structure.

During plan generation, plans are grouped in plan classes. All plans in one class are equivalent according to an equivalence relation. For example, all plans producing the same result can be put in one class. As we will see in Chapter 5, other equivalence criteria are conceivable as well. Plan properties can be distinguished into logical and physical properties, where logical properties are those that are equal for all plans in the same plan class and physical properties are those that can differ between plans in the same class. In the aforementioned scenario, where plans are classified by their result, the result cardinality is a logical plan property, whereas the tuple order produced by the plan is a physical property. It can differ between plans in the same class depending on how the contained relations are accessed. Ideally, we only store one plan per plan class at any point in time, namely the cheapest plan in this class found so far. As we will see in the following sections, this is not always possible.

Many "interesting" plan properties such as the tuple order produced by a plan, or the functional dependencies holding in the plan's result are conceivable. However, taking plan properties into consideration tends to increase the complexity of plan generation by violating Bellman's Principle of Optimality. Subsequently, we will see some examples of how interesting properties can be identified and efficiently handled in a plan generator.

To compare different plans in the same plan class and decide which of them is the best, some cost metric has to be available. The cost of a plan is determined by a cost function. The cost function should ideally take into account all factors that determine the work necessary to evaluate a certain plan in the respective system. For example, a cost function applied in a disk-based database system should model the cost of accessing the harddisk as precisely as possible. One such cost model has been described by Haas et al. [21].

Subsequently, we will use a rather simple cost function. This allows for comprehensible examples because the cost of a plan can easily be calculated mentally. The cost function we use is called C_{out} and is recursively defined as follows:

$$C_{out}(T) = \begin{cases} 0 & \text{if T is a single relation} \\ |T| + C_{out}(T_1) + C_{out}(T_2) & \text{if } T = T_1 \circ T_2 \end{cases}$$

Here, $|T|$ denotes the result cardinality of a plan or operator tree T and $\circ$ acts as a placeholder for some join operator. Clearly, this function can only be applied to operator trees containing only joins and base relations. If necessary, we will extend it to take other algebraic operators into account as well. With C_{out},

the cost of a join tree is determined only by the cardinalities of the contained relations and intermediate results. While other, more complex cost functions typically depend on other factors as well, the cardinality of intermediate results always is one of them. Since we cannot execute the query to determine the cardinalities in order to find an optimal execution plan, we have to rely on estimated values.

Because all decisions made during plan generation ultimately depend on these cardinality estimates, it is of crucial importance to make them as precise as possible. Cardinality estimation is an ongoing research topic in its own right and has to this date not been solved satisfyingly. Most current approaches rely on two rather unrealistic assumptions: (1) that the data is uniformly distributed and (2) that predicates applied in the query are uncorrelated. Since most real data sets and queries do not conform to these assumptions, optimizers typically have to work with inaccurate estimates. This frequently leads to wrong decisions that can negate the effort invested in optimizing a query [25].

In this thesis we will not take these limitations into account because we consider the problem of cardinality estimation to be independent of that of query optimization.

3 Building Blocks of Current Plan Generators

This chapter provides an overview of the architecture of generative plan generators. We start with the enumerator, which is responsible for enumerating pairs of relation sets (ccps) that can be joined to build a (sub-)plan. The next subsection deals with the conflict detector (first presented in [30]), which is needed to correctly reorder different join operators and finally we take a look at plan properties and how they can be used to extend the functionality of the plan generator.

3.1 Enumeration

A plan generator based on dynamic programming was first presented by Selinger et al. as part of the System R research prototype developed at IBM [40]. They identified the ordering of join operators as a major influence factor on the runtime of a query. Consequently, a plan generator capable of optimizing the join order formed the core of their optimizer. In general, the problem of finding an optimal join order for a given query is NP-hard [22]. Some trade-offs are typically made to reduce the complexity by deliberately limiting the search space of the plan generator at the risk of missing an optimal solution. For example, in System R the search space was restricted to left-deep trees.

Without these limitations, the number of bushy join trees containing cross products for a query with n relations can be calculated as follows: The number of binary trees with n leaves is given by $C(n-1)$ where C denotes the Catalan numbers: $\mathcal{C}(n) = \sum_{k=0}^{n-1} \mathcal{C}(k)\mathcal{C}(n-k-1)$. The n relations can then be attached to the leaves of each binary tree in any possible order. Thus, the total number of bushy join trees with cross products for n relations is $\mathcal{C}(n-1)n!$.

Selinger et al. proposed to enumerate join trees in the order of increasing size. For this, the DP table is first initialized with access paths to single relations before combining them to build plans containing two relations. These plans are then used to build plans of size three by adding one of the remaining relations and so on. When choosing the next relation to be added to an existing plan, cross products are deferred by always preferring relations that are connected through a join predicate to one of the other relations already present in the plan. Thereby, cross products are only considered if they are required by the input query and they are applied as late as possible.

Hereafter, we adopt the naming scheme introduced by Moerkotte and Neumann. They label a slightly modified version of the plan generator described above *DPsize* [32]. This name reflects the abovementioned size-driven enumeration order of plans. DPsize differs from the original System R optimizer in the

```
DPsize
    // Input: a connected simple query graph with relations
          R = {R_0, ..., R_{n-1}}
    // Output: an optimal bushy operator tree
 1  for all R_i ∈ R
 2    DPTable[{R_i}] = R_i
 3  for all 1 < s ≤ n ascending // size of plan
 4    for all 1 ≤ s_1 < s // size of left subplan
 5      s_2 = s − s_1 // size of right subplan
 6      for all S_1 ⊂ R : |S_1| = s_1
                S_2 ⊂ R : |S_2| = s_2
 7        P_1 = DPTable[S_1]
 8        P_2 = DPTable[S_2]
 9        if P_1 == NULL ∨ P_2 == NULL
10          continue
11        if not (S_1 connected to S_2)
12          continue
13        BuildPlan(S_1, S_2, ⋈_p)
14  return DPTable[R]
```

Figure 3.1: Plan generator DPsize

```
BuildPlan(S_1, S_2, ∘_p)
1  OptimalCost = ∞
2  S = S_1 ∪ S_2
3  P_1 = DPTable[S_1]
4  P_2 = DPTable[S_2]
5  if DPTable[S] ≠ NULL
6    OptimalCost = Cost(DPTable[S])
7  if Cost(P_1 ∘_p P_2) < OptimalCost
8    DPTable[S] = (P_1 ∘_p P_2)
```

Figure 3.2: Pseudo code for BuildPlan

sense that it enumerates not only left-deep but bushy join trees and it expects a fully connected query graph. Thus, cross products are not supported. The pseudo code for the algorithm is given in Figure 3.1.

First, the DP table is initialized with the access paths to single relations. Next, plans of increasing size are enumerated, after which the table entries for the two sets are retrieved. If for one of the two no entry is found, the loop continues without building a plan. That is because a new plan joining S_1 and S_2 can only be built if plans for the two sets have already been enumerated.

If, for example, no plan for S_1 is available at this point, this means that S_1 does not induce a connected subgraph of the query graph. Otherwise, a plan would be available since $|S_1| < |S|$ and plans are enumerated in the order of increasing size. The second if-statement ensures that there is a join predicate between S_1 and S_2. In summary these tests ensure that (S_1, S_2) is a ccp, as defined in Definition 7.

If all tests succeed, the subroutine BuildPlan is called. It is given in Figure 3.2 and is a common building block of all DP-based plan generators. The routine expects S_1 and S_2 and a join operator $\circ$ with an associated predicate p as arguments. Inside the routine the plans already stored for S_1 and S_2 are retrieved from the DP table. Then, the cost of joining these two plans is determined by calling a cost function. If the new plan is cheaper than an existing plan for $S_1 \cup S_2$, or there is none, it is inserted into the DP table. Otherwise, the plan is discarded and the routine returns. In the end DPsize returns the table entry for the relation set R which contains all relations referenced in the input query.

In order to remove the limitations imposed by Selinger et al., Vance and Maier proposed a DP-based plan generator capable of enumerating bushy trees with cross products [41]. They tried to cope with the vast search space by choosing a different enumeration strategy. Instead of building plans of a certain size, they came up with an efficient way of enumerating subsets of a given relation set by representing relation sets as bitvectors. First, each relation is mapped to a number. Then, an integer is used as a bitvector encoding a set of relations by representing the relation associated with a certain number i with the bit at position i in the bitvector.

To enumerate all subsets of a given set in an order suited for dynamic programming, the integer is continuously incremented and the resulting numbers are interpreted as bitvectors, each representing a relation set. Again, we adopt the name Moerkotte and Neumann assigned to this algorithm. They chose the name *DPsub* to reflect the subset-driven enumeration order implemented by the plan generator [32]. Figure 3.3 provides the pseudo code. Aside from the different enumeration approach, the algorithm works identically to DPsize.

While the two algorithms described so far use different enumeration strategies, none of them is optimal in the sense that it reaches the theoretical lower complexity bound for the problem it solves. As Moerkotte and Neumann pointed out, the complexity of the join ordering problem is determined by the shape of the query graph [32]. Therefore, they proposed a different approach with their DP-based plan generator *DPccp*. The key idea is to consider joining only those sets of relations that are connected in themselves and to each other by join predicates. This means that the tests for connectedness we have seen in the loops in Figures 3.1 and 3.3 can be avoided. Instead, connectedness is ensured by traversing the query graph and efficiently enumerating ccps (see Definintion 7). Expecting a connected query graph as input and only considering the connected components of the latter, DPccp is not suited for queries containing cross products, or for the introduction of cross products for the sake of plan optimality. However, the plan generator is capable of producing all kinds of join trees, including bushy trees.

DPsub

```
   // Input: a connected simple query graph with relations
        R = {R_0, ..., R_{n-1}}
   // Output: an optimal bushy operator tree
 1 for all R_i ∈ R
 2   DPTable[{R_i}] = R_i
 3 for 1 ≤ i < 2^n − 1 ascending
 4   S = {R_j ∈ R|(⌊i/2^i⌋ mod 2) = 1
 5   if not (S induces a csg)
 6     continue
 7   for all S_1 ⊂ S, S_1 ≠ ∅
 8     S_2 = S \ S_1
 9     P_1 = DPTable[S_1]
10     P_2 = DPTable[S_2]
11     if P_1 == NULL ∨ P_2 == NULL
12       continue
13     if not (S_1 connected to S_2)
14       continue
15     BuildPlan(S_1, S_2, ⋈_p)
16 return DPTable[R]
```

Figure 3.3: Plan generator DPsub

DPccp

```
   // Input: a connected simple query graph with relations
        R = {R_0, ..., R_{n-1}}
   // Output: an optimal bushy operator tree
1  for all R_i ∈ R
2    DPTable[{R_i}] = R_i
3  for all ccps (S_1, S_2)
4    BuildPlan(S_1, S_2, ⋈_p)
5    BuildPlan(S_2, S_1, ⋈_p)
6  return DPTable[R]
```

Figure 3.4: Plan generator DPccp

Since the details of the graph traversal are not of particular interest for this work, it is merely treated as a black-box in the pseudo code for DPccp given in Figure 3.4. One important detail is that ccps are symmetric, i.e., only one of the pairs (S_1, S_2) and (S_2, S_1) is enumerated. Therefore, commutativity has to be handled explicitly by calling BuildPlan twice for each ccp.

Besides the specification of a DP-based plan generator that meets the lower

```
DPhyp
   // Input: a connected query graph with relations R = {R_0, ..., R_{n-1}}
   // Output: an optimal bushy operator tree
1  for all R_i ∈ R
2    DPTable[{R_i}] = R_i
3  for all ccps (S_1, S_2)
4    BuildPlan(S_1, S_2, ⋈_p)
5    BuildPlan(S_2, S_1, ⋈_p)
6  return DPTable[R]
```

Figure 3.5: Plan generator DPhyp

complexity bound for an arbitrarily shaped query, the authors also analyzed the strengths and weaknesses of the other two algorithms. They came to the conclusion that DPsize is superior to DPsub for chain and cycle queries and vice versa for star and clique queries. As expected, DPccp is superior to both alternatives in the sense that it performs better or equally as good for all kinds of query shapes.

However, all of the plan generators discussed so far, including DPccp, lack the ability to handle hypergraph queries. In Section 2.3 we have already seen how non-binary predicates result in a query hypergraph and as we will see in Chapter 4, hypergraphs also play a crucial role when it comes to optimizing the join order in the presence of outerjoins. Therefore, Moerkotte and Neumann went one step further and extended DPccp in such a way that it would be able to work with hypergraphs. The resulting algorithm is called *DPhyp* [30]. For the sake of completeness, its pseudo code is given in Figure 3.5. Since the only difference to DPccp lies in the way the ccps are enumerated and we do not go into detail on this aspect of the plan generator, the only difference between Figures 3.4 and 3.5 is that the algorithm shown in the latter accepts a hypergraph as its input.

When comparing the different algorithms presented in this section, we notice that they consist of two independent modules. The first one is an enumerator for pairs of relation sets. The enumerator is the distinguishing part of the plan generator and ultimately determines its efficiency. Since a highly efficient hypergraph-aware enumerator exists in DPhyp, we consider this problem solved. The second part is the plan builder that is responsible for creating a plan joining the two sets provided by the enumerator and inserting the newly built plan into the DP table. This part of the plan generator is equal in all the variants presented above. In Chapter 5 we modify it to incorporate the optimal placement of grouping operators.

The modular design allows for an easy replacement of certain parts of the plan generator and we follow it throughout the rest of this work by encapsulating all extensions we propose in a similar fashion. Thus, they can be incorporated in any of the plan generators described above or possible future approaches. The

d_id	name
0	Sales
1	R&D

(a) Departments

e_id	name	d_id
0	Doe	0
1	Smith	1

(b) Employees

c_id	e_id
0	0

(c) Cars

Figure 3.6: Example Relations

next section deals with a third component of a state-of-the-art plan generator, namely the conflict detector.

3.2 Conflict Detection

The plan generators discussed so far are all designed under the assumption that the join operators contained in the input query are freely reorderable. What this means is that changing their order does not change the result of the query. The only constraint these algorithms put on the join order is implied by the fact that they only allow for plans without cross products.

In general, only cross products provide full reorderability. Their ordering only affects the cost of the resulting plan but not its result. Inner joins are a bit more restrictive because of the syntactic constraints imposed by their join predicate. The relations referenced in the predicate have to be available to make the join applicable. In the plan generators we have seen so far, this is implicitly guaranteed by ensuring that there is a join predicate between two sets of relations that are considered as a join pair. As long as these constraints are fulfilled, any ordering of inner joins is possible without affecting the correctness of the query result. Clearly, all join operators with attached predicates are subject to such syntactic constraints.

However, all other join operators introduced in Section 2.2 are more limited in their reorderability. For example, changing the relative order of an inner join and a left outerjoin can change the result of the underlying query. In order to support queries containing non-inner joins, a plan generator needs to take these so-called *reordering conflicts* into account to avoid incorrect join orders.

As an example, consider the relations shown in Figure 3.6 that contain information about departments, employees and company cars used by the employees. Figure 3.7 shows an SQL query against this schema that returns an overview of the departments and the employees using a company car in each department. In order to list all departments in the result, including those that do not have employees with a company car, a left outerjoin is used. The table below the query displays the query result. There, a dash denotes a null value.

As can be seen in Figure 3.8, changing the join order such that the left outerjoin is applied before the inner join changes the result of the query. Therefore, this reordering is invalid and has to be avoided during plan generation.

Since the plan generators from the previous section have no means for preventing such invalid reorderings, they are only suited for optimizing queries containing inner joins. To remove this limitation, some form of conflict detec-

```
select d.name, e.name, c.c_id
from
departments d left outer join
 (employees e join cars c on c.e_id = e.e_id)
on e.d_id = d.d_id;
```

d.name	e.name	c.c_id
Sales	Doe	0
R&D	-	-

Figure 3.7: Query containing left outerjoin and query result

```
select d.name, e.name, c.c_id
from
(departments d left outer join employees e
 on e.d_id = d.d_id)
join cars c on c.e_id = e.e_id;
```

d.name	e.name	c.c_id
Sales	Doe	0

Figure 3.8: Query containing left outerjoin and query result

tion preventing incorrect join orders has to be implemented. Figure 3.9 shows a version of DPhyp with conflict detection.

As input the plan generator still expects a query graph and, in addition to this, the set of join operators contained in the query. This additional information is necessary, since possible reordering conflicts depend on the type of an operator used to join a ccp. When the enumerator emits a ccp (S_1, S_2), we iterate through the operator set and for each operator $\circ$ with attached join predicate p call the new subroutine APPLICABLE, passing the ccp and $\circ_p$ as arguments. The routine returns true if $\circ_p$ is suitable for joining S_1 and S_2, taking the syntactic constraints imposed by p and possible reordering conflicts present in the new plan candidate into account. Before building a plan joining S_1 and S_2 in reversed order, we need to make sure that $\circ$ is commutative.

Since the conflict detector is encapsulated in a separate routine and thereby independent of the other parts of the plan generator, namely the enumerator and the plan builder, it fits nicely in our modular design approach. Conflict detection can therefore be incorporated in all of the plan generators discussed in Section 3.1. However, we can make it more efficient by representing reordering conflicts by hyperedges in the query graph. This gives DPhyp a clear advantage over the other algorithms, since it is the only plan generator that can handle hypergraphs.

In Chapter 4 we will evaluate several possible implementations of APPLI-

```
DPhyp
   // Input: a set of relations R = {R_0, ..., R_{n-1}},
             a set of operators O with associated predicates,
             a query graph H
   // Output: an optimal bushy operator tree
1  for all R_i ∈ R
2        DPTable[{R_i}] = R_i // initial access paths
3  for all ccps (S_1, S_2) of H
4    for all ∘_p ∈ O
5      if Applicable(S_1, S_2, ∘_p)
6        BuildPlan(S_1, S_2, ∘_p)
7        if ∘_p is commutative
8        BuildPlan(S_2, S_1, ∘_p)
9  return DPTable[R]
```

Figure 3.9: Plan generator DPhyp with conflict detection

cable and demonstrate how reordering conflicts can be encoded in the query graph. There, we also systematize the different reordering transformations by defining what we call the *core search space*.

3.3 Exploiting Plan Properties

The concept of logical and physical plan properties has first been introduced in Section 2.4. Until this point, we have only considered the cost of a (sub-)plan to decide whether it should be stored in the DP table, or not. However, it can be beneficial to take other plan properties into account. Selinger et al. illustrated this by introducing the concept of interesting orders [40]. Interesting orders can be identified in two ways: they are either explicitly specified in an *order by* or *group by* clause in the input query, or they are derived from the predicates in the query in order to facilitate the application of merge joins. Since plans in the same class can produce different tuple orders depending on the way the base relations are accessed, the sort order is a physical plan property.

If a certain subplan p produces tuples in an interesting order (e.g., by applying an index scan instead of a full table scan), it can be beneficial to use this subplan as part of the overall solution, even if it is more expensive than another plan p' in the same plan class. That is because the tuple ordering provided by p can save sort operations that would otherwise be necessary further up in the operator tree to achieve an order or grouping specified in the input query. If no specific order or grouping is desired, using p as part of the final plan can enable the application of merge joins without the need to sort one of the join arguments beforehand, again leading to possible cost savings.

The downside of this approach is that subplans for the same relation set can become essentially incomparable. If, in the scenario described above, p is

```
struct Plan {
  Plan* next; // pointer to next plan for this plan class
  Plan* left; // pointer to left or only argument
  Plan* right; // pointer to right argument
  Properties properties; // struct storing plan properties
}
```

Figure 3.10: Plan node with pointer to next plan

more expensive than p', but on the other hand produces tuples in an intersting order, it is not immediately clear which of the two plans is better because the cost savings enabled by p's tuple order only manifest themselves further up in the tree. Since plans are built bottom-up, both plans have to be kept in the DP table. One way of doing this is to store in each plan a pointer to the next plan of the same plan class, thus effectively storing a linked list of plans. Such a plan structure is illustrated in Figure 3.10. Clearly, it is the task of the implementor of a plan generator to identify useful properties. Their choice strongly depends on the capabilities of the system and the plan generator. For example, maintaining the tuple order of every plan only makes sense if the system implements sort-based operators that can take advantage of this. The properties can be stored in a separate data structure that is associated with every plan, as proposed in Figure 3.10.

Storing all plans with different physical properties in the solution table leads to a massive increase of the search space size. Therefore, criteria for pruning down the search space while still guaranteeing an optimal solution are required. In the abovementioned situation we could safely discard p if p' did not only provide an interesting order, but was also cheaper than p. In this case we call p *dominated* by p'.

The example of interesting orders gives a first impression of how exploiting plan properties can lead to better query plans at the price of increasing the complexity of the plan generator. In Chapter 5 we will see more examples of interesting plan properties and how they can be used to extend the functionality of a plan generator. Identifying effective optimality-preserving pruning criteria is a central problem investigated in this chapter.

3.4 Summary

To summarize this section, Figure 3.11 provides an overview of the different elements of a state-of-the-art generative plan generator and how they are related. Although the focus of this work lies on DP-based plan generators, the shown design approach applies to memoization-based plan generators as well, since the two only differ in how the enumerator works.

If a hypergraph-aware enumerator such as DPhyp is in place and the join operators specified in the input query are not freely reorderable, a suitable conflict detector turns the query graph into a hypergraph. Thereby, reordering conflicts are encoded in the query graph as far as possible. However, this step

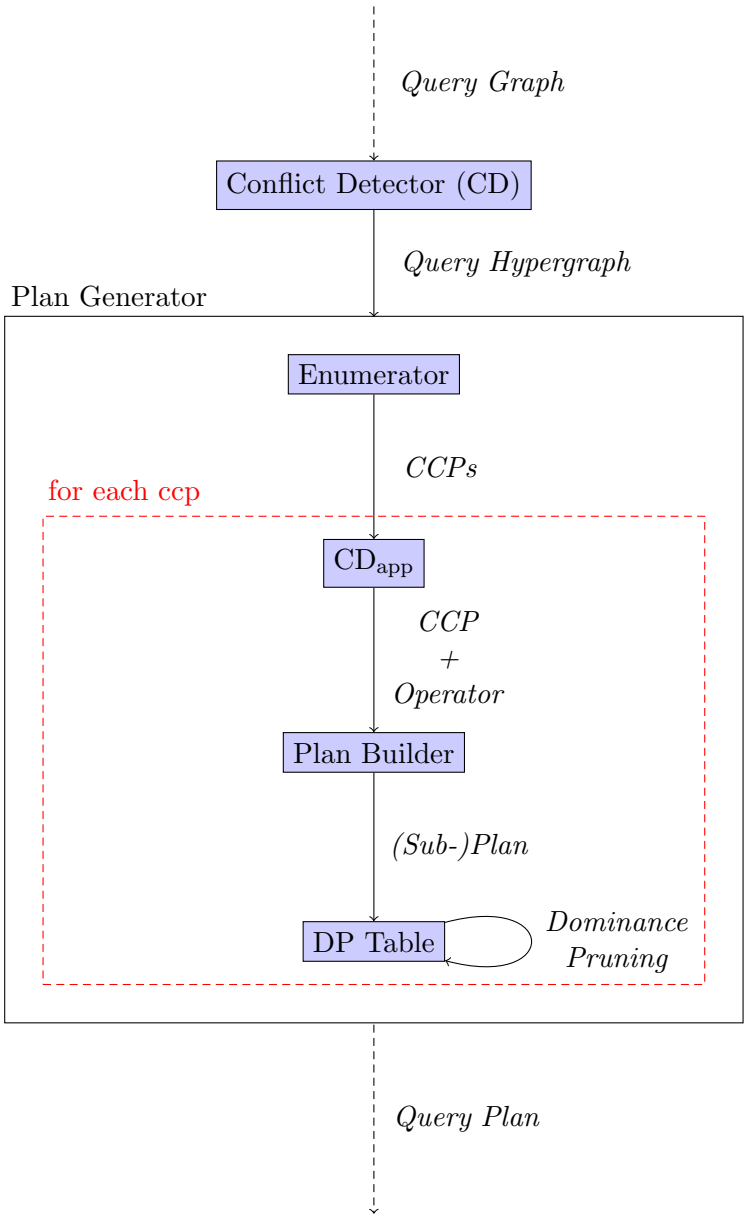


Figure 3.11: Components of a DP-based plan generator

is not necessary, because conflicts can also be detected later on by conducting an applicability test every time a new plan is built. The resulting hypergraph is passed to the plan generator.

The first step of the actual plan generation is the enumeration of ccps. For each ccp produced by the enumerator, the conflict detector is needed again to check if a suitable join operator can be found. That is because not all reordering conflicts can be represented by hyperedges in the query graph. If the applicability test is successful, the ccp with its associated join operator is passed to the plan builder, which builds the plan and inserts it into the DP table, if it is

the best plan for this relation set found so far. If plan properties compromising Bellman's Principle of Optimality are taken into account, dominance pruning can be applied at this point. Otherwise, the best plan is determined solely based on the plan cost.

Once all ccps have been considered, an optimal query plan can be retrieved from the DP table and is passed on to the next step of query compilation, which can either be a second rewrite phase or code generation.

4 Reordering Non-Inner Joins

This chapter covers the problem of reordering join operators that are not freely reorderable in a generative plan generator. First, we formalize the notion of reordering conflicts and introduce the so-called *core search space* in Section 4.2. In Section 4.3 we analyze existing approaches for dealing with reordering conflicts and show that they produce invalid plans. Therefore, three new approaches are described that are all correct, meaning that they all produce only valid plans. They can be found in Section 4.4. While they all guarantee correctness, there are significant differences between the three when it comes to their completeness. By completeness we mean the ability to only prevent invalid plans and no valid plans. Only one of the three is both complete and correct. An experimental evaluation of the different approaches can be found in Section 4.6. The content of this chapter was published in [31].

4.1 Introduction

In Section 3.2 the conflict detector is introduced as one of the basic components of a plan generator. In our design it serves two purposes: the encoding of reordering conflicts in the query graph in the form of hyperedges and, since this is not always possible for all conflicts occurring in a query, an explicit applicability test that is used to detect all remaining conflicts. Clearly, encoding conflicts in the query graph requires a hypergraph-aware enumeration algorithm. If the plan generator cannot handle hypergraphs, all conflicts have to be detected by an applicability test, meaning that invalid ccps are enumerated and then rejected. On the other hand, encoding conflicts in the hypergraph enables the plan generator to avoid the enumeration of invalid pairs altogether. See Figures 3.9 and 3.11 again for an idea of the conflict detector's role in our design.

In the literature we find two ways of preventing invalid plans in a DP-based plan generator. The first approach (NEL/EEL) is by Rao et al. Their conflict detector can deal with joins, left outerjoins and antijoins [36, 37]. The second approach (SES/TES) is by Moerkotte and Neumann and handles all operators in LOP [30]. As we will show in Sections 4.3 and 4.6, both approaches generate invalid plans. This means that their application in a real system is out of the question.

While correctness is the minimal requirement for a usable conflict detector, we also strive for completeness. What this means is that ideally, the conflict detector should reject only invalid plans. This enables the plan generator to generate all valid plans and choose the best among them. Other desirable properties are the flexibility to support a wide range of join predicates and the extensibility to allow for an easy addition of new operators with corresponding

reordering properties. The conflict detector CD-C, which is discussed in Section 4.4, fulfills all these requirements.

4.2 The Core Search Space

In this section the core search space is introduced. It is defined by a set of transformation rules exploring all valid alternatives to a given initial plan. Section 4.2.1 introduces these transformation rules and Section 4.2.2 defines the core search space.

4.2.1 Reorderability

Traditional join ordering approaches are intended to reorder inner joins and no other binary operators. Since the inner join is *commutative* and *associative*, all plans are valid and there is no danger of generating invalid plans. But in general, plan generators must reorder other join operators as well (e.g., $\ltimes$, $\triangleright$, $\bowtie$, $\mathbb{M}$, $\bowtie\!\!\!\!\!\!\Gamma$). Hence, we need to carry over the notions of commutativity and associativity to (pairs of) these operators. It is easy to see that some of the aforementioned operators are commutative, while others are not (see Table 4.1). If a binary operator $\circ$ is commutative, we denote this by comm($\circ$). If comm($\circ$) holds, the corresponding cell in the table contains a plus sign. Otherwise, it contains a minus sign.

$\circ$	$\times$	$\bowtie$	$\ltimes$	$\triangleright$	$\bowtie$	$\mathbb{M}$	$\bowtie\!\!\!\!\!\!\Gamma$
comm($\circ$)	+	+	-	-	-	+	-

Table 4.1: The comm($\circ$)-property

Associativity is a little more complex. We say that two not necessarily distinct operators $\circ^a$ and $\circ^b$ are associative if the following equivalence holds:

$$(e_1 \circ^a_{12} e_2) \circ^b_{23} e_3 \equiv e_1 \circ^a_{12} (e_2 \circ^b_{23} e_3). \tag{4.1}$$

Here, we adhere to the following convention. If an operator has a predicate, then the subscript ij indicates that it references attributes (and, thus, relations) from at most e_i and e_j. Hence, for $1 \leq i, j \leq 3$, $i \neq j$ this also indicates that $\mathcal{F}(e_{ij}) \cap \mathcal{F}(e_k) = \emptyset$ for $1 \leq k \leq 3$ and $k \notin \{i, j\}$. This ensures that the equivalence is correctly typed on both sides of the equivalence sign. For example, the predicate of $\circ^a_{12}$ accesses tables from e_1 and e_2, but not e_3. Note that $\circ^a_{12}$ may carry a complex predicate referencing more than two tables from e_1 and e_2. We will see an example in the next subsection. If some $\circ^a_{123}$ referenced tables in all three expressions e_1, e_2 and e_3, the expression on the left-hand side of Equivalence 4.1 would be invalid and the right-hand side would be valid, but could not be transformed into the left-hand side. For the purpose of conflict detection, complex predicates accessing more than two relations are no challenge. They just enlarge the set of tables that must be present before the complex predicate can be evaluated. The real challenge with complex predicates lies in the efficient

enumeration of the now more restricted search space (more on this in Section 4.5.1).

If Equivalence 4.1 holds for two operators $\circ^a$ and $\circ^b$, we denote this by $\text{assoc}(\circ^a, \circ^b)$. It is important to note that assoc is not symmetric. This means that the order of the operators (i.e., $(\circ^a, \circ^b)$ vs. $(\circ^b, \circ^a)$) is important. We tie the order in assoc to the syntactic pattern of Equivalence 4.1. It has to be the same order as on the left-hand side of the equivalence. This means that the left association has to be on the left-hand side and, consequently, the right association on the right-hand side of the equivalence.

If $\text{comm}(\circ^a)$ and $\text{comm}(\circ^b)$ hold, then $\text{assoc}(\circ^a, \circ^b)$ implies $\text{assoc}(\circ^b, \circ^a)$ and vice versa, as can be seen from

$$\begin{aligned}
(e_1 \circ^a_{12} e_2) \circ^b_{23} e_3 &\equiv e_1 \circ^a_{12} (e_2 \circ^b_{23} e_3) && \text{assoc}(\circ^a, \circ^b)\\
&\equiv (e_2 \circ^b_{23} e_3) \circ^a_{12} e_1 && \text{comm}(\circ^a)\\
&\equiv (e_3 \circ^b_{23} e_2) \circ^a_{12} e_1 && \text{comm}(\circ^b)\\
&\equiv e_3 \circ^b_{23} (e_2 \circ^a_{12} e_1) && \text{assoc}(\circ^b, \circ^a)\\
&\equiv (e_2 \circ^a_{12} e_1) \circ^b_{23} e_3 && \text{comm}(\circ^b)\\
&\equiv (e_1 \circ^a_{12} e_2) \circ^b_{23} e_3 && \text{comm}(\circ^a).
\end{aligned}$$

Table 4.2 summarizes the associativity properties. Since assoc is not symmetric, $\circ^a$ *must* be looked up within a row and $\circ^b$ within a column. For some operators $\text{assoc}(\circ^a, \circ^b)$ only holds if one or both of the predicates associated with $\circ^a$ and $\circ^b$ reject nulls (see Definition 1). For more details, see the corresponding footnotes at the bottom of the table.

$\circ^a$	$\circ^b$						
	$\times$	$\bowtie$	$\ltimes$	$\triangleright$	$\mathbb{M}$	$\mathbb{M}\!\!\times$	$\bowtie\!\!\!\!-$
$\times$	+	+	+	+	+	-	+
$\bowtie$	+	+	+	+	+	-	+
$\ltimes$	-	-	-	-	-	-	-
$\triangleright$	-	-	-	-	-	-	-
$\mathbb{M}$	-	-	-	-	+[1]	-	-
$\mathbb{M}\!\!\times$	-	-	-	-	+[1]	+[2]	-
$\bowtie\!\!\!\!-$	-	-	-	-	-	-	-

[1] if p_{23} rejects nulls on $\mathcal{A}(e_2)$ (Eqv. 4.1)
[2] if p_{12} and p_{23} reject nulls on $\mathcal{A}(e_2)$ (Eqv. 4.1)

Table 4.2: The $\text{assoc}(\circ^a, \circ^b)$-property

The following equivalence for the semijoin shows that commutativity and associativity do not cover all valid transformations:

$$(e_1 \ltimes_{12} e_2) \ltimes_{13} e_3 \equiv (e_1 \ltimes_{13} e_3) \ltimes_{12} e_2.$$

Clearly, we cannot derive the join order on the right-hand side from the one on the left-hand side using associativity and commutativity because neither of the two holds for the semijoin. Instead, we need a third property which we call the *left asscom property* (l-asscom for short). It is defined as follows:

$$(e_1 \circ^a_{12} e_2) \circ^b_{13} e_3 \equiv (e_1 \circ^b_{13} e_3) \circ^a_{12} e_2. \tag{4.2}$$

We denote by l-asscom($\circ^a, \circ^b$) the fact that Equivalence 4.2 holds for $\circ^a$ and $\circ^b$. Analogously, we can define a *right asscom property* (r-asscom):

$$e_1 \circ^a_{13} (e_2 \circ^b_{23} e_3) \equiv e_2 \circ^b_{23} (e_1 \circ^a_{13} e_3). \tag{4.3}$$

First, note that l-asscom and r-asscom are symmetric properties, i.e.,

$$\begin{aligned} \text{l-asscom}(\circ^a, \circ^b) &\Leftrightarrow \text{l-asscom}(\circ^b, \circ^a), \\ \text{r-asscom}(\circ^a, \circ^b) &\Leftrightarrow \text{r-asscom}(\circ^b, \circ^a). \end{aligned}$$

The following reasoning

$$\begin{aligned} (e_1 \circ^a_{12} e_2) \circ^b_{23} e_3 &\equiv (e_2 \circ^a_{12} e_1) \circ^b_{23} e_3 && \text{if comm}(\circ^a_{12}) \\ &\equiv (e_2 \circ^b_{23} e_3) \circ^a_{12} e_1 && \text{if l-asscom}(\circ^a_{12}, \circ^b_{23}) \\ &\equiv e_1 \circ^a_{12} (e_2 \circ^b_{23} e_3) && \text{if comm}(\circ^a_{12}) \\ &\equiv (e_1 \circ^a_{12} e_2) \circ^b_{23} e_3 && \text{if assoc}(\circ^a_{12}, \circ^b_{23}) \end{aligned}$$

implies that

$$\begin{aligned} \text{comm}(\circ^a_{12}), \text{assoc}(\circ^a_{12}, \circ^b_{23}) &\Rightarrow \text{l-asscom}(\circ^a_{12}, \circ^b_{23}), \\ \text{comm}(\circ^a_{12}), \text{l-asscom}(\circ^a_{12}, \circ^b_{23}) &\Rightarrow \text{assoc}(\circ^a_{12}, \circ^b_{23}). \end{aligned}$$

Thus, the l-asscom property is implied by associativity and commutativity, which explains its name. Quite similarly, the implications

$$\begin{aligned} \text{comm}(\circ^b_{23}), \text{assoc}(\circ^a_{12}, \circ^b_{23}) &\Rightarrow \text{r-asscom}(\circ^a_{12}, \circ^b_{23}), \\ \text{comm}(\circ^b_{23}), \text{r-asscom}(\circ^a_{12}, \circ^b_{23}) &\Rightarrow \text{assoc}(\circ^a_{12}, \circ^b_{23}) \end{aligned}$$

can be deduced.

Table 4.3 summarizes the l-/r-asscom properties. Again, entries with a footnote require that some predicates reject nulls. We assume that calls to assoc and l/r-asscom take care of this.

$\circ$	$\times$	$\bowtie$	$\ltimes$	$\triangleright$	$\fullouterjoin$	$\rightouterjoin$	$\leftouterjoin$
$\times$	+/+	+/+	+/-	+/-	+/-	-/-	+/-
$\bowtie$	+/+	+/+	+/-	+/-	+/-	-/-	+/-
$\ltimes$	+/-	+/-	+/-	+/-	+/-	-/-	+/-
$\triangleright$	+/-	+/-	+/-	+/-	+/-	-/-	+/-
$\fullouterjoin$	+/-	+/-	+/-	+/-	+/-	+[1]/-	+/-
$\rightouterjoin$	-/-	-/-	-/-	-/-	+[2]/-	+[3]/+[4]	-/-
$\leftouterjoin$	+/-	+/-	+/-	+/-	+/-	-/-	+/-

[1] if p_{12} rejects nulls on $\mathcal{A}(e_1)$ (Eqv. 4.2)
[2] if p_{13} rejects nulls on $\mathcal{A}(e_3)$ (Eqv. 4.2)
[3] if p_{12} and p_{13} rejects nulls on $\mathcal{A}(e_1)$ (Eqv. 4.2)
[4] if p_{13} and p_{23} reject nulls on $\mathcal{A}(e_3)$ (Eqv. 4.3)

Table 4.3: The l-/r-asscom($\circ^a, \circ^b$) property

If an entry in one of the Tables 4.1 to 4.3 is marked with − or its condition in the footnote is violated, we say that there is a *conflict* regarding this property. A conflict means that the application of the corresponding transformation rule results in an invalid plan.

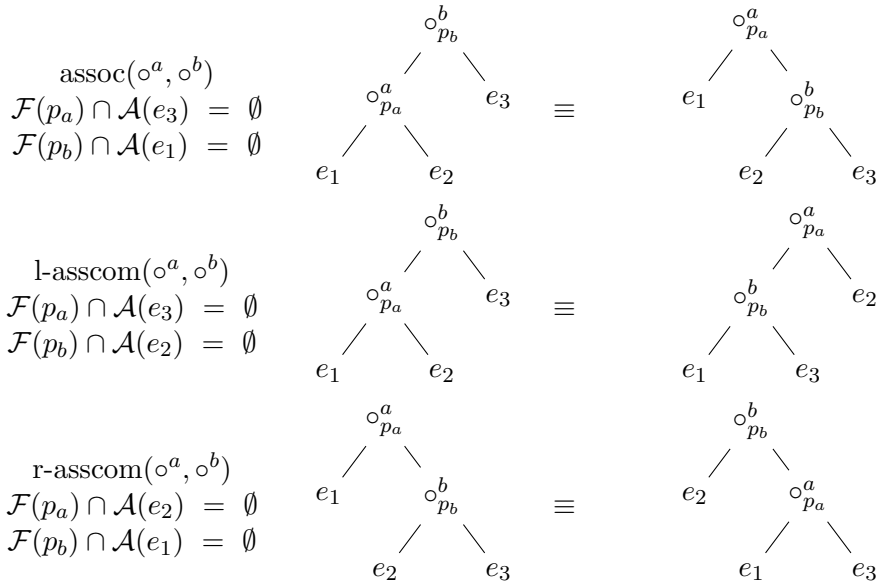


Figure 4.1: Transformation rules for assoc, l-asscom, and r-asscom

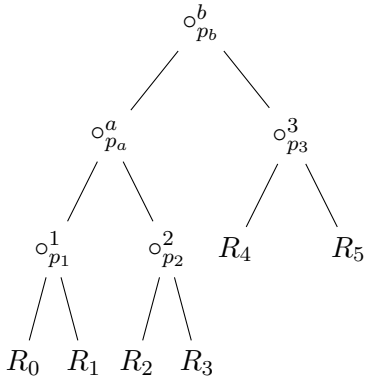


Figure 4.2: Example operator tree

4.2.2 Definition of the Core Search Space

Figure 4.1 shows an overview of the basic transformations that result from the reordering properties discussed so far, except for commutativity. All equivalences can be applied from left to right and from right to left. We define the *core search space* for a given initial plan to be the set of plans generated by exhaustively applying these four transformations to an initial plan.

Figure 4.2 shows a larger operator tree. Let us consider several possibilities for the predicate of the top-most operator $\circ^b$. If $p_b \equiv R_0.a + R_1.a + R_2.a + R_3.a = R_4.a * R_5.a$, then no reordering is possible, since all tables are referenced. If $p_b \equiv R_2.a + R_3.a = R_4.a * R_5.a$, then applying associativity is possible from a syntactic point of view, since in our example $\mathcal{F}_{\mathcal{T}}(p_b) \cap \mathcal{T}(e_1)$ becomes

$\{R_2, R_3, R_4, R_5\} \cap \{R_0, R_1\} = \emptyset$. In fact, although the predicate is complex, it references only tables below $\circ^2$ and $\circ^3$, whose subtrees correspond to e_2 and e_3 in Figure 4.1. Clearly, a binary predicate, e.g., $p_b \equiv R_0.a = R_5.a$, generates the largest search space and, thus, the most opportunities for generating invalid plans and missing valid plans.

Taking a closer look at the syntactic constraints shown in Figure 4.1, we observe that for non-degenerate predicates (see Definition 2) the following holds:

Observation 1. *The syntactic constraints for non-degenerate predicates imply that (1) either associativity or l-asscom can be applied for left nesting, but not both and (2) either associativity or r-asscom can be applied for right-nesting, but not both.*

Thus, non-degenerate predicates simplify the handling of conflicts, since we have to take care of either associativity or l/r-asscom and never both at the same time.

Figure 4.3 shows an example of the core search space for the expression $(e_1 \circ^a_{12} e_2) \circ^b_{13} e_3$. We observe that any expression in the core search space can be reached by a sequence of at most two applications of commutativity, at most one application of associativity, l-asscom, or r-asscom, finally followed by at most two applications of commutativity. The total number of applications of commutativity can be restricted to 2. More specifically, one application of commutativity to each operator in the plan suffices.

4.3 Existing Approaches

If an input query involves binary operators other than $\bowtie$ and $\times$, not all transformations as discussed in Section 4.2.2 are valid. Thus, any plan generator must be modified in such a way that it restricts its search to valid transformations only. Otherwise, the generated plan may not be equivalent to the input query and therefore the result may be wrong.

Several approaches to restrict the search space are described in existing work. First, the problem of outerjoin simplification and reordering was extensively studied by Galindo-Legaria and Rosenthal [16, 17, 38]. They identified a subclass of join and outerjoin queries where the query graph unambiguously determines the semantics of a query. For this type of queries, they proposed a procedure that analyzes paths in the query graph to detect conflicting reorderings. They enhanced a conventional dynamic programming algorithm to deal with these conflicts. Although very useful, their approach is restricted to joins and outerjoins and the query graph must exhibit some specific properties.

In order to handle complex predicates, Bhargava et al. extended this approach and presented a conflict detector, which analyzes paths in hypergraphs [1]. Their approach is also limited to joins and outerjoins. Rao et al. presented a method that is not restricted to joins and outerjoins. They additionally considered antijoins and proposed to use the initial operator tree instead of the query graph in order to maintain the semantics of the input query [36, 37]. Their idea is to calculate a set of relations associated with every predicate. This set of

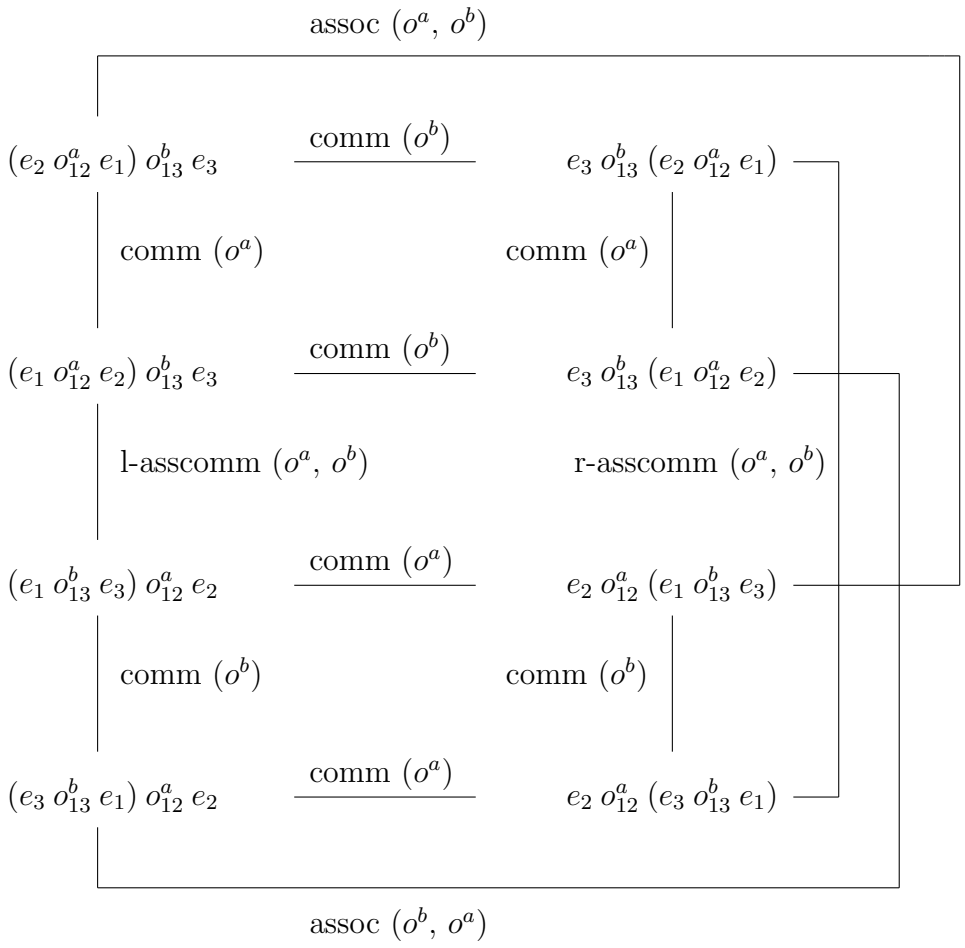


Figure 4.3: Core search space example

relations (called EEL, for *extended eligibility list*) must be available before the predicate can be evaluated.

To calculate the EEL($\circ$) for an operator $\circ$, they first determined the operator's so-called *normal eligibility list* (NEL($\circ$)), which contains all relations that must be present before $\circ$ can be applied. That is, a plan of the form $S_1 \circ S_2$ is valid only if SES($\circ$) $\subseteq S_1 \cup S_2$ holds. In general, the NEL is equal to the set of relations referenced in the operator's predicate. In the next subsection we discuss how the NEL is extended to become the EEL which can then be used to restrict the plan generator's search space to valid join orders.

Moerkotte and Neumann adopted the idea of NEL and EEL and called them SES and TES, respectively [30]. SES stands for *syntactic eligibility set* and TES stands for *total eligibility set*. Other than that, the two sets basically have the same function as before. However, they are calculated differently and support all join operators in LOP (see Section 2.2). Neither the approach by Rao et al., nor the one by Moerkotte and Neumann is correct. Both generate invalid plans. Subsequently, we will present examples demonstrating how they work and why they fail. We will also fix the algorithm by Rao et al.

4.3.1 Reordering Outerjoins and Antijoins with EELs

First, we explain the approach by Rao et al. in short [36, 37]. Then, we give a counter-example that shows the incorrectness of their method. After that we show how it can be repaired.

In their paper Rao et al. propose an algorithm called CALC_{EEL} to compute the EEL for each predicate carried by a join operator $\circ \in \{\bowtie, \mathbb{M}, \triangleright\}$ [36]. The pseudo code of CALC_{EEL} is shown in Figure 4.4. CALC_{EEL} computes the EELs in a single bottom-up traversal (Lines 4-20) of the initial operator tree. During the traversal, it maintains for each relation R an outerjoin set $outer_R$ and an antijoin set $anti_R$. Initially, both sets contain only the corresponding relations themselves (Lines 2 and 3). Thereby, outer_R stores all relations that are linked through either inner- or antijoin predicates (Lines 13-16) and anti_R keeps track of all relations $R \in \mathcal{T}(\text{left}(\circ)) \cap \text{NEL}(\circ)$ that are linked through $\mathbb{M}$ (Lines 17-20). Essentially, this means that R has to be on the preserving side of a one-sided outerjoin predicate. As the name implies, outer_R is used to compute the EEL for an outerjoin predicate (Lines 6-8). Similarly, anti_R is used to compute the EEL for an antijoin predicate (Lines 9-12). The test executed in $\text{APPLICABLE}(S_1, S_2, \circ_p)$ is $\text{EEL}(\circ_p) \subseteq S_1 \cup S_2$.

CALC_{EEL}

```
    // Input: T(∘), NEL(∘) where ∘ ∈ {⋈, ⟗, ▷}
    // Output: EEL(∘)
 1  for each R ∈ T(topmost ∘)
 2    outer_R = {R}
 3    anti_R = {R}
 4  for each operator ∘ during bottom-up traversal
 5    EEL(∘) = NEL(∘)
 6    if ∘ ∈ {⟗}
 7      W = ⋃_{R∈T(right(∘))∩NEL(∘)} outer_R
 8      EEL(∘) = EEL(∘) ∪ W
 9    elseif ∘ ∈ {▷}
10      V = ⋃_{R∈T(left(∘))∩NEL(∘)} anti_R
11      U = {R|R ∈ T(right(∘)) ∩ NEL(∘)}
12      EEL(∘) = EEL(∘) ∪ V ∪ U
13    if ∘ ∈ {⋈, ▷}
14      W = ⋃_{R∈NEL(∘)} outer_R
15      for each R ∈ W
16        outer_R = W
17    elseif ∘ ∈ {⟗}
18      V = ⋃_{R∈T(left(∘))∩NEL(∘)} anti_R
19      for each R ∈ T(right(∘)) ∩ NEL(∘)
20        anti_R = anti_R ∪ V
```

Figure 4.4: Pseudo code for Calc_{EEL}

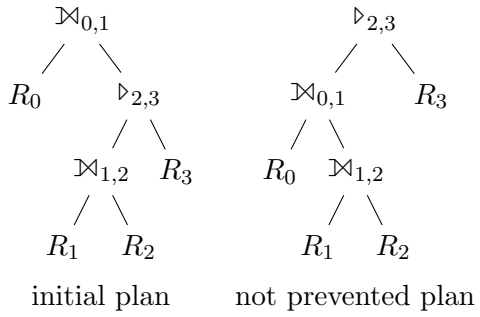


Figure 4.5: Example showing the incorrectness of Calc_{EEL}

Figure 4.5 shows an example in which EELs resulting from CALC_{EEL} do not prevent the generation of invalid plans. The initial plan is given on the left. The plan on the right can only be derived by applying $\text{assoc}(\mathbb{M}_{0,1}, \triangleright_{2,3})$. A look at Table 4.2 reveals that $\text{assoc}(\mathbb{M}_{0,1}, \triangleright_{2,3})$ is not valid. Thus, the two plans are not equivalent. We can verify this by using the relations in Table 4.4 as input for both plans. The result of the initial plan is given in Table 4.5. Clearly, it differs from the result of the invalid plan shown in Table 4.6.

Table 4.7 shows anti_{R} and outer_{R} after executing CALC_{EEL}. Table 4.8 displays the results of CALC_{EEL}. According to $\text{EEL}(\mathbb{M}_{0,1})$ and $\text{EEL}(\triangleright_{2,3})$, the antijoin $\triangleright_{2,3}$ can be applied on top of the outerjoin $\mathbb{M}_{0,1}$, which is wrong. $\text{EEL}(\mathbb{M}_{0,1})$ should contain $\{R_0, R_1, R_2, R_3\}$ in order to be correct because $\neg\text{assoc}(\mathbb{M}_{0,1}, \triangleright_{2,3})$ holds.

R_0	R_1		R_2		R_3
A	A	B	B	C	C
1	1	1	1	1	1

Table 4.4: Example relations

$R_0 \mathbb{M}_{R_0.A=R_1.A} ((R_1 \mathbb{M}_{R_1.B=R_2.B} R_2) \triangleright_{R_2.C=R_3.C})$

$R_0.A$	$R_1.A$	$R_1.B$	$R_2.B$	$R_2.C$	$R_3.C$
1	-	-	-	-	-

Table 4.5: Result of initial plan (Fig. 4.5)

We can easily fix CALC_{EEL} as follows: we only have to eliminate the intersection with $\text{NEL}(\circ)$ in Lines 7 and 18 as in

7 $\quad W = \bigcup_{R \in \mathcal{T}(right(\circ))} outer_R$

...

18 $\quad V = \bigcup_{R \in \mathcal{T}(left(\circ))} anti_R$

With this fix CALC_{EEL} prevents reordering conflicts, but is not complete any more. Hence, we traded in correctness for incompleteness, which still is a major improvement. The incompleteness of the fixed algorithm can be verified by

$(R_0 \bowtie_{R_0.A=R_1.A} (R_1 \bowtie_{R_1.B=R_2.B} R_2)) \triangleright_{R_2.C=R_3.C}$

$R_0.A$	$R_1.A$	$R_1.B$	$R_2.B$	$R_2.C$	$R_3.C$
$\emptyset$					

Table 4.6: Result of invalid plan (Fig. 4.5)

R	$outer_R$	$anti_R$
R_0	$\{R_0\}$	$\{R_0\}$
R_1	$\{R_1\}$	$\{R_0, R_1\}$
R_2	$\{R_2, R_3\}$	$\{R_1, R_2\}$
R_3	$\{R_2, R_3\}$	$\{R_3\}$

Table 4.7: anti_R and outer_R after executing Calc_EEL

using $R_0 \bowtie_{0,1} (R_1 \bowtie_{1,2} R_2)$ as input plan. The modified CalcEEL procedure returns $\text{EEL}(\bowtie_{0,1}) = \{R_0, R_1, R_2\}$, which prevents $(R_0 \bowtie_{0,1} R_1) \bowtie_{1,2} R_2$, although the latter is a valid plan because $\text{assoc}(\bowtie_{0,1}, \bowtie_{1,2})$ holds. Thus, $\text{EEL}(\bowtie_{0,1})$ should contain $\{R_0, R_1\}$ only.

4.3.2 Reordering Joins with TESs

Before a join operator $\circ_p$ can be applied, the plan generator needs to ensure that the producer/consumer constraints implied by p are met. The conventional test is to check if $\mathcal{F}_\mathcal{T}(p)$ is a subset of $\mathcal{T}(\circ)$. Moerkotte and Neumann extended this test to detect reordering conflicts [30]. Therefore, they introduced the notion of the TES. The TES is defined to be a set of relations that is attached to every binary operator $\circ_p$ in the query and is a superset of the SES, which captures the syntactic constraints imposed by the join predicate p. Before $\circ_p$ can be applied to join a ccp (S_1, S_2), it is ensured that all elements of $\text{TES}(\circ_p)$ are contained in $S_1 \cup S_2$.

Moerkotte and Neumann proposed an algorithm called CalcTES, which can be found in their paper [30]. It calculates the TES for every join operator in LOP. As it turns out, their approach is neither correct nor complete: it generates wrong plans and misses correct ones.

Figure 4.6 contains an example showing the incorrectness of the TES approach: the plan on the right is not equivalent to the initial plan on the left. Applying $\text{assoc}(\bowtie_{1,2}, \triangleright_{2,3})$ as a first step and $\text{assoc}(\bowtie_{0,1}, \triangleright_{2,3})$ thereafter transforms the initial plan into the plan on the right. To see that the plan on the right is invalid, consider the different results in Tables 4.9 and 4.10, which are

$\circ$	NEL	EEL
$\bowtie_{1,2}$	$\{R_1, R_2\}$	$\{R_1, R_2\}$
$\triangleright_{2,3}$	$\{R_2, R_3\}$	$\{R_1, R_2, R_3\}$
$\bowtie_{0,1}$	$\{R_0, R_1\}$	$\{R_0, R_1\}$

Table 4.8: NEL and EEL after executing Calc_EEL

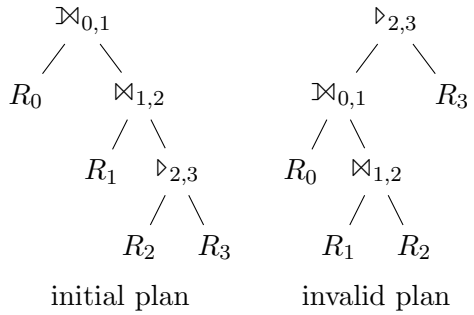


Figure 4.6: Example showing the incorrectness of Calc_{TES}

$R_0 \mathbb{M}_{R_0.A=R_1.A} (R_1 \bowtie_{R_1.B=R_2.B} (R_2 \triangleright_{R_2.C=R_3.C} R_3))$

$R_0.A$	$R_1.A$	$R_1.B$	$R_2.B$	$R_2.C$
1	NULL	NULL	NULL	NULL

Table 4.9: Result of initial plan (Fig. 4.6)

based on the same input relations as before (Table 4.4).

Table 4.11 shows the result of applying $\textsc{CalcTES}$ to the initial plan. Due to the values of $\text{TES}(\mathbb{M}_{0,1})$ and $\text{TES}(\triangleright_{2,3})$, the test $\text{TES}(\mathbb{M}_{0,1}) \subseteq \{R_0, R_1, R_2\}$ succeeds, allowing the antijoin $\triangleright_{2,3}$ to move on top of $\mathbb{M}_{0,1}$, which is invalid, since $\neg\text{assoc}(\mathbb{M}_{0,1}, \triangleright_{2,3})$ holds. In order to prevent the reordering, $\text{TES}(\mathbb{M}_{0,1})$ should contain $\{R_0, R_1, R_2, R_3\}$.

4.4 Conflict Detection

In this section we propose three new conflict detection algorithms named CD-A, CD-B and CD-C. All of them are correct and CD-C is also complete. We adopt the naming conventions proposed in previous work, meaning that our approaches all make use of two relation sets named SES and TES [30]. In addition to this, CD-B and CD-C also require a set of so-called *conflict rules*.

$(R_0 \mathbb{M}_{R_0.A=R_1.A} (R_1 \bowtie_{R_1.B=R_2.B} R_2)) \triangleright_{R_2.C=R_3.C} R_3$

$R_0.A$	$R_1.A$	$R_1.B$	$R_2.B$	$R_2.C$
$\emptyset$				

Table 4.10: Result of invalid plan (Fig. 4.6)

$\circ$	SES	TES
$\triangleright_{2,3}$	$\{R_2, R_3\}$	$\{R_2, R_3\}$
$\bowtie_{1,2}$	$\{R_1, R_2\}$	$\{R_1, R_2\}$
$\mathbb{M}_{0,1}$	$\{R_0, R_1\}$	$\{R_0, R_1, R_2\}$

Table 4.11: SES and TES after executing Calc_{TES}

```
CALC_SES(∘_p)
    // Input: binary operator ∘ ∈ LOP carrying predicate p
1   if ∘_p ∈ {⋈, ⟕, ⟖, ▷, ⟕}
2     return ⋃_{R∈F_T(p)} {R} ∩ T(∘_p)
3   elseif ∘_{p;a_1:e_1,...,a_n:e_n} ∈ {⟗}
4     return ⋃_{R∈F_T(p)∪F_T(e_i)} {R} ∩ T(∘_p)
5   else                          // cross product ×
6     return ∅
```

Figure 4.7: Pseudo code for Calc_{SES}

4.4.1 Outline

In order to open our conflict detectors to new algebraic operators, we use a table-driven approach. We use four tables containing the information from Tables 4.1, 4.2 and 4.3 (the latter includes two tables). Extending our approach only requires to extend these tables.

We develop our final approach in three steps. In each step we introduce one of our conflict detectors CD-A, CD-B, and CD-C. For each of these conflict detectors, we present a complete bundle consisting of three components:

1. a conflict representation,
2. an algorithm, which detects the conflicts in the initial operator tree and produces a conflict representation for each operator contained in it and
3. the implementation of APPLICABLE, which uses the conflict representation for an operator and then determines whether the operator can be applied in a given context.

Each of the bundles we will discuss subsequently is correct, but only the last one is complete.

The main idea in the following (the same as in previous work [30, 36, 37]) is to extend the producer/consumer constraints modeled through the SES by adding more tables to it. This is meant to restrict the explored search space to valid plans only, which is possible, since the SES is used to express syntactic constraints: all referenced attributes and tables must be present before an expression can be evaluated. Therefore, the explored search space will become smaller if we add more tables.

Let us now define the SES. First of all, the SES contains the tables referenced by a predicate. If some operator such as the groupjoin $⟕$ introduces new attributes, they will be treated as if they belong to a new table. This new table is present in the set of accessible tables after the groupjoin has been applied. Let R be a table and let $\circ_p$ be any of our binary operators other than a groupjoin. The pseudo code for the SES calculation is shown in Figure 4.7. In the case of non-degenerate predicates, $\text{CALC}_{\text{SES}}(\circ_p) = \mathcal{F}_{\mathcal{T}}(p)$.

We always initialize TES with SES. Furthermore, we assume that our conflict representation has two accessors *L-TES* and *R-TES*, that are defined as follows:

$$\begin{aligned} \text{L-TES}(\circ) &:= \text{TES}(\circ) \cap \mathcal{T}(\text{left}(\circ)) \text{ and} \\ \text{R-TES}(\circ) &:= \text{TES}(\circ) \cap \mathcal{T}(\text{right}(\circ)). \end{aligned}$$

This distinction is necessary, because we want to consider commutativity explicitly and in those cases where commutativity does not hold, we want to prevent operators which occurred on the left-hand side of an operator from moving to its right-hand side, or vice versa.

For a ccp (S_1, S_2) all our implementations of Applicable include the following test:

$$\text{L-TES} \subseteq S_1 \ \wedge \ \text{R-TES} \subseteq S_2.$$

4.4.2 Approach CD-A

Let us first consider a simple operator tree with only two operators. Take a look at the upper half of Figure 4.8. There, the application of associativity and l-asscom to a plan is illustrated. In case associativity does not hold, we add $\mathcal{T}(e_1)$ to $\text{TES}(\circ^b)$. This prevents the plan on the right-hand side of the arrow marked with assoc. It does not, however, prevent the plan on the right-hand side of the arrow marked with l-asscom. Similarly, adding $\mathcal{T}(e_2)$ to $\text{TES}(\circ^b)$ does prevent the plan resulting from l-asscom, but not the plan resulting from applying associativity. The lower part of Figure 4.8 shows the actions needed if an operator is nested in the right argument. Again, we can precisely prevent the invalid plans.

Only one problem remains to be solved. It occurs if a conflicting operator $\circ_a$ is not a direct child of $\circ_b$, but instead a descendant situated deeper in the operator tree. This is possible since in general, the e_i are trees themselves. Some reordering could possibly move a conflicting operator $\circ_a$ up to the top of an argument subtree.

Thus, we have to calculate the TESs bottom-up by applying CD-A to every operator $\circ^b$ in the operator tree. The pseudo code for CD-A is shown in Figure 4.9. The conflict representation comprises the TES for every operator. The pseudo code for Applicable is:

$\text{Applicable}_A(S_1, S_2, \circ)$

// **Input:** binary operator $\circ$, sets of tables S_1, S_2

1 **return** $\text{L-TES}(\circ) \subseteq S_1 \wedge \text{R-TES}(\circ) \subseteq S_2$

Let us now verify that Applicable_A is correct. We have to show that it prevents the generation of invalid plans. Take the ¬assoc case with nesting on the left. Let the original operator tree contain $(e_1 \circ^a_{12} e_2) \circ^b_{23} e_3$. Define the set of tables $R_2 := \mathcal{F}_\mathcal{T}(\circ^b_{23}) \cap \mathcal{T}(\text{left}(\circ^b_{23}))$ and $R_3 := \mathcal{F}_\mathcal{T}(\circ^b_{23}) \cap \mathcal{T}(\text{right}(\circ^b_{23}))$. Then, $\text{SES}(\circ^b_{23}) = R_2 \cup R_3$. Further, since $\neg\text{assoc}(\circ^a_{12}, \circ^b_{23})$, we have

$$\text{TES}(\circ^b_{23}) \supseteq \text{SES}(\circ^b_{23}) \cup \mathcal{T}(e_1).$$

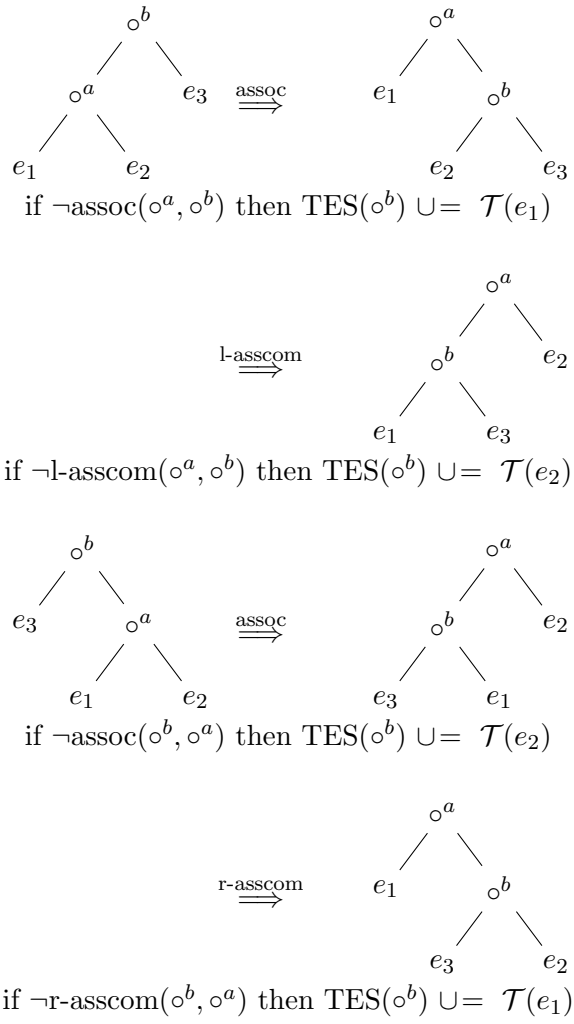


Figure 4.8: Calculating TES according to CD-A

Note that we used $\supseteq$ and not equality, since TES($\circ^b$) could be larger due to other conflicts. Next, we observe that

$$\begin{aligned}
\text{L-TES}(\circ^b_{23}) &\supseteq (\text{SES}(\circ^b_{23}) \cup \mathcal{T}(e_1)) \cap \mathcal{T}(\text{left}(\circ^b_{23})) \\
&\supseteq (\text{SES}(\circ^b_{23}) \cap \mathcal{T}(\text{left}(\circ^b_{23}))) \cup \\
&\quad\ (\mathcal{T}(e_1) \cap \mathcal{T}(\text{left}(\circ^b_{23}))) \\
&\supseteq ((R_2 \cup R_3) \cap \mathcal{T}(\text{left}(\circ^b_{23}))) \cup (\mathcal{T}(e_1)) \\
&\supseteq R_2 \cup \mathcal{T}(e_1)
\end{aligned}$$

and

$$\begin{aligned}
\text{R-TES}(\circ^b_{23}) &\supseteq (\text{SES}(\circ^b_{23}) \cup \mathcal{T}(e_1)) \cap \mathcal{T}(\text{right}(\circ^b_{23})) \\
&\supseteq \text{SES}(\circ^b_{23}) \cap \mathcal{T}(\text{right}(\circ^b_{23})) \\
&\supseteq R_3.
\end{aligned}$$

CD-A($\circ^b$)

```
    // Input: operator ∘^b
 1  TES(∘^b) = CALC_SES(∘^b)
 2  for ∀ ∘^a ∈ STO(left(∘^b))
 3    if ¬assoc(∘^a, ∘^b)
 4      TES(∘^b) = TES(∘^b) ∪ T(left(∘^a))
 5    if ¬l-asscom(∘^a, ∘^b)
 6      TES(∘^b) = TES(∘^b) ∪ T(right(∘^a))
 7  for ∀ ∘^a ∈ STO(right(∘^b))
 8    if ¬assoc(∘^b, ∘^a)
 9      TES(∘^b) = TES(∘^b) ∪ T(right(∘^a))
10    if ¬r-asscom(∘^b, ∘^a)
11      TES(∘^b) = TES(∘^b) ∪ T(left(∘^a))
```

Figure 4.9: Pseudo code for CD-A

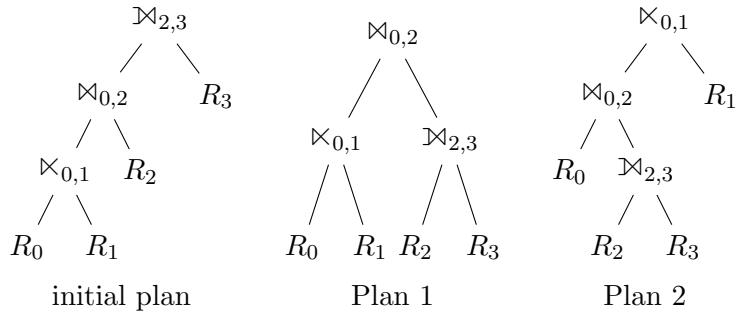


Figure 4.10: Example illustrating incompleteness of CD-A

Let (S_1, S_2) be a ccp generated by our plan generator with conflict detection. Then, the call APPLICABLE(S_1,S_2,$\circ^b$) checks

$$\begin{aligned} \text{L-TES}(\circ^b_{23}) &\subseteq S_1 \text{ and} \\ \text{R-TES}(\circ^b_{23}) &\subseteq S_2. \end{aligned}$$

This fails if $S_1 \not\supseteq \mathcal{T}(e_1)$. Hence, neither $e_2 \circ^b_{23} e_3$, nor $e_3 \circ^b_{23} e_2$ will be generated and, consequently, $e_1 \circ^a_{12} (e_2 \circ^b_{23} e_3)$ will not be generated either. Similarly, if $\neg$l-asscom($\circ^a, \circ^b$), L-TES($\circ^b$) will contain $\mathcal{T}(e_2)$ and the test prevents the generation of $e_1 \circ^b e_3$. The remaining two cases can be verified analogously.

From this discussion it follows that our plan generator generates only valid plans. However, it does not generate all valid plans. It is incomplete, as we can see from the example shown in Figure 4.10. Since $\neg$assoc($\ltimes_{0,1}, \rtimes_{2,3}$), TES($\rtimes_{2,3}$) contains R_0 (line 4 of CD-A($\rtimes_{2,3}$)) . Thus, neither the valid plans Plan 1 and Plan 2, nor any plan that can be derived from the two by applying join commutativity will be generated.

4.4.3 Approach CD-B

In order to reduce the number of valid plans missed by our plan generator, we introduce the more flexible mechanism of *conflict rules*. A conflict rule (CR) is a pair of table sets denoted by $T_1 \rightarrow T_2$. With every operator in the operator tree we associate a set of conflict rules, so our conflict representation now consists of a TES and a set of conflict rules for every operator.

Before we describe the construction of conflict rules, let us illustrate their role in APPLICABLE(S_1, S_2, $\circ$). To this end, we define *rule obedience* as follows:

Definition 8. *A conflict rule $T_1 \rightarrow T_2$ is* obeyed *for relation sets S_1 and S_2 if with $S = S_1 \cup S_2$ the following condition holds:*

$$T_1 \cap S \neq \emptyset \implies T_2 \subseteq S.$$

Thus, if T_1 contains a single table from S, S must contain all tables in T_2. Keeping this in mind, it is easy to see that invalid plans are indeed prevented by obeying the rules shown in Figure 4.11. As we will see, the TES is restricted to the SES in CD-B. Compared to the TES, conflict rules allow for more flexibility: while the TES containment test is unconditional, conflict rules represent a conditional containment test.

The pseudo code for the new conflict detector is given in Figure 4.12 with CD-B. As before, we apply CD-B bottom-up to every operator $\circ^b$ in the tree.

With the conflict rules we need a new test for applicability. Now, the test given in Figure 4.13 with APPLICABLE$_{B/C}(S_1, S_2, \circ)$ checks for two conditions:

1. L-TES $\subseteq S_1 \wedge$ R-TES $\subseteq S_2$ must hold (Line 1) and
2. all rules in the rule set of $\circ$ must be obeyed (Lines 2-6).

With CD-B all plans in Figure 4.10 can be generated.

Again, this implementation of APPLICABLE is correct, but not complete, as the example in Figure 4.14 shows. Since assoc($\bowtie_{0,1}, \ltimes_{1,3}$), assoc($\bowtie_{1,2}, \ltimes_{1,3}$) and l-asscom($\bowtie_{1,2}, \ltimes_{1,3}$), the only conflict occurs due to $\neg$r-asscom($\bowtie_{0,1}, \ltimes_{1,3}$). Consequently, CR($\bowtie_{0,1}$) contains the following rule:

$$\mathcal{T}(\{R_3\}) \rightarrow \mathcal{T}(\{R_1, R_2\})$$

This rule prevents the plan on the right-hand side of Figure 4.14, which is overly careful, since $R_2 \notin \mathcal{F}_\mathcal{T}(\ltimes_{1,3})$. In fact, r-asscom would never be applied in this example, since $\bowtie_{0,1}$ accesses table R_1, meaning that applying r-asscom would destroy the producer/consumer relationship ($\mathcal{F}_\mathcal{T}(\bowtie_{0,1}) \cap \{R_1, R_2\} \neq \emptyset$) already covered by SES($\bowtie_{0,1}$).

4.4.4 Approach CD-C

The approach CD-C differs from CD-B only in the way the conflict rules are calculated. The conflict representation and the procedure APPLICABLE remain

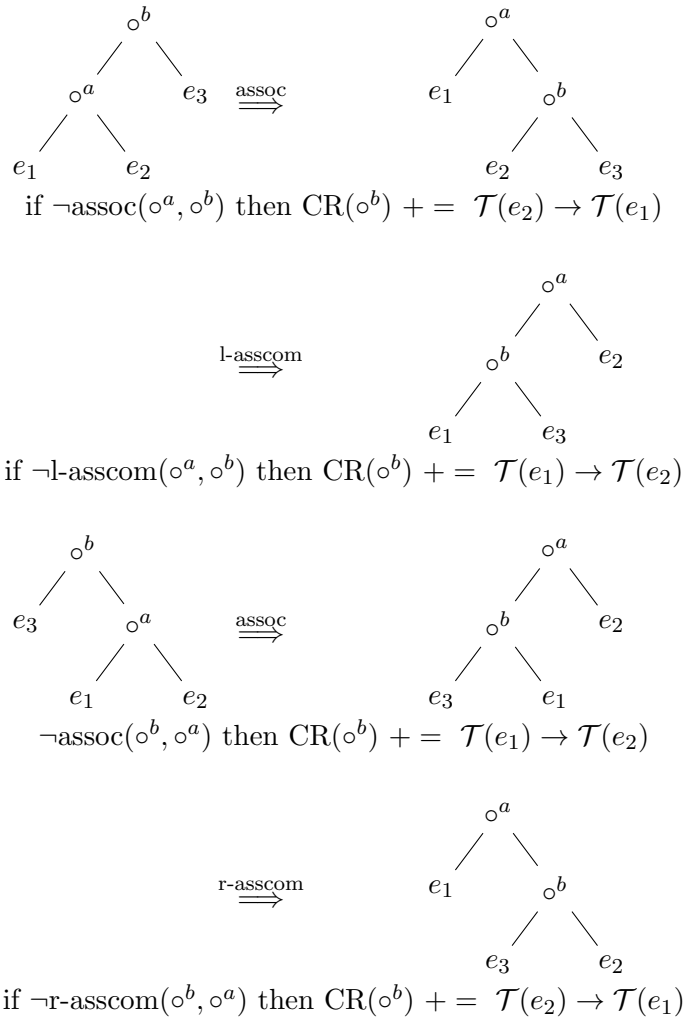


Figure 4.11: Calculating conflict rules according to CD-B

the same. The idea is to learn from the above example and include only those tables under operator $\circ^a$, which occur in its predicate. However, we have to be careful to include special cases for degenerate predicates and cross products. The pseudo code is given in Figure 4.15. Let us revisit the example from Section 4.4.3. Since in the example the only conflict occurs due to ¬r-asscom($\bowtie_{0,1}, \ltimes_{1,3}$), the rule set CR($\bowtie_{0,1}$) contains $\mathcal{T}(\{R_3\}) \rightarrow \mathcal{T}(\{R_1\})$ (Line 21 of CD-C). As a consequence, the plan on the right-hand side of Figure 4.14 will not be prevented anymore.

We show that APPLICABLE$_{B/C}$ for the ¬assoc case with nesting on the left is correct. The remaining cases can be proven similarly. Let the original operator tree contain $(e_1 \circ^a_{12} e_2) \circ^b_{23} e_3$. Since ¬assoc($\circ^a, \circ^b$), one of the following (Line 5

CD-B($\circ^b$)

```
    // Input: operator ∘^b
 1  TES(∘^b) = CALC_SES(∘^b)
 2  for ∀ ∘^a ∈ STO(left(∘^b))
 3    if ¬assoc(∘^a, ∘^b)
 4      CR(∘^b) += T(right(∘^a)) → T(left(∘^a))
 5    if ¬l-asscom(∘^a, ∘^b)
 6      CR(∘^b) += T(left(∘^a)) → T(right(∘^a))
 7  for ∀ ∘^a ∈ STO(right(∘^b))
 8    if ¬assoc(∘^b, ∘^a)
 9      CR(∘^b) += T(left(∘^a)) → T(right(∘^a))
10    if ¬r-asscom(∘^b, ∘^a)
11      CR(∘^b) += T(right(∘^a)) → T(left(∘^a))
```

Figure 4.12: Pseudo code for CD-B

$\text{APPLICABLE}_{B/C}(S_1, S_2, \circ)$

```
    // Input: binary operator ∘, sets of tables S1, S2
1   if L-TES(∘) ⊆ S1 ∧ R-TES(∘) ⊆ S2
2     for all (T1 → T2) ∈ CR(∘)
3       if T1 ∩ (S1 ∪ S2) ≠ ∅
4         if T2 ⊈ (S1 ∪ S2)
5           return FALSE
6     return TRUE
7   else
8     return FALSE
```

Figure 4.13: Pseudo code for $\text{Applicable}_{\text{B/C}}$

or Line 7) holds:

$$\begin{aligned} \text{CR}(\circ^b) \;&+= \; \mathcal{T}(e_2) \rightarrow \mathcal{T}(e_1) \text{ or} \\ &+= \; \mathcal{T}(e_2) \rightarrow \mathcal{T}(e_1') \text{ with } e_1' \subset e_1 \wedge e_1' \neq \emptyset. \end{aligned}$$

The second case subsumes the first case. Thus, it suffices to show the second case. To construct $e_1 \circ_{12}^a (e_2 \circ_{23}^b e_3)$ (right-hand side of Equivalence 4.1), the subtree $(e_2 \circ_{23}^b e_3)$ must be constructed first. We show that the routine $\text{APPLICABLE}_{B/C}(S_1, S_2, \circ^b)$ returns *false* with either (A) $\mathcal{T}(e_2) \subseteq S_1 \wedge \mathcal{T}(e_3) \subseteq S_2$ or (B) $\mathcal{T}(e_3) \subseteq S_1 \wedge \mathcal{T}(e_2) \subseteq S_2$. If the test in Line 1 fails, *false* is returned and we are done. Otherwise, L-TES($\circ$) $\subseteq S_1$ holds. Note that, since we are trying to construct $(e_2 \circ_{23}^b e_3)$, $\mathcal{T}(e_1) \cap (S_1 \cup S_2) = \emptyset$ must hold. On the other hand, the conflict rule $T_1 \rightarrow T_2$ with $T_1 = \mathcal{T}(e_2)$ and $T_2 \supseteq \mathcal{T}(e_1')$ is contained in CR($\circ^b$). Thus, for this rule $T_1 \rightarrow T_2$: $T_1 \cap (S_1 \cup S_2) \neq \emptyset$ and $T_2 \not\subseteq (S_1 \cup S_2)$ hold

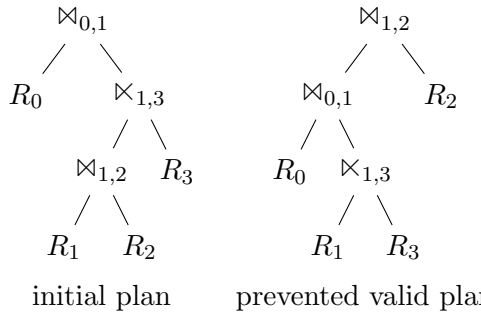


Figure 4.14: Example illustrating incompleteness of CD-B

in both cases (A) and (B). Accordingly, *false* is returned by $\textsc{Applicable}_{B/C}$. Hence, CD-C and, consequently, CD-B are correct.

4.4.5 Rule Simplification

Reducing the number of conflict rules slightly decreases plan generation time, because fewer tests for rule obedience have to be carried out. Thus, laws like

$$\begin{aligned} R_1 \to R_2, R_1 \to R_3 &\equiv R_1 \to R_2 \cup R_3 \\ R_1 \to R_2, R_3 \to R_2 &\equiv R_1 \cup R_3 \to R_2 \end{aligned}$$

should be used to rearrange the rule set for efficient evaluation.

However, we are much more interested in eliminating rules altogether by adding their right-hand side to the TES. This is due to the following argument: in Section 2.3 we stated that a higher number of hyperedges in the query graph leads to a smaller search space by reducing the number of ccps the plan generator needs to consider. Since the hyperedges are constructed from the TES (more on this in the following section), eliminating conflict rules and in turn adding some relations to one or more TESs has a positive impact on the runtime of hypergraph-aware plan generators like DPhyp or its memoization-based equivalent TDMcCHyp [12].

Consider a conflict rule $R_1 \to R_2$ for an operator $\circ$. If $R_1 \cap \text{TES}(\circ) \neq \emptyset$, we can add R_2 to TES due to the existential quantifier on the left-hand side of a rule in the definition of *obey* (see Definition 8). Moreover, if $R_2 \subseteq \text{TES}(\circ)$, we can safely eliminate the rule. Applying these rearrangements is often possible, since both $(\mathcal{T}(\text{left}(\circ^a)) \cap \mathcal{F}_\mathcal{T}(\circ))$ and $(\mathcal{T}(\text{right}(\circ^a)) \cap \mathcal{F}_\mathcal{T}(\circ))$ will be non-empty.

4.5 Minor Issues

In this section we investigate some smaller issues that have not been covered so far. One of them is how the TES can be used to add hyperedges to the query graph, thereby allowing the plan generator to avoid invalid reorderings without the need for an explicit applicability test. The second topic of this section is how crossproducts or, in general, degenerate predicates can be handled by the conflict detector.

```
CD-C(∘^b)
     // Input: operator ∘^b
 1   TES(∘^b) = CALC_SES(∘^b)
 2   for ∀ ∘^a ∈ STO(left(∘^b))
 3     if ¬assoc(∘^a, ∘^b)
 4       if T(left(∘^a)) ∩ F_T(∘^a) ≠ ∅
 5         CR(∘^b) += T(right(∘^a)) → T(left(∘^a)) ∩ F_T(∘^a)
 6       else
 7         CR(∘^b) += T(right(∘^a)) → T(left(∘^a))
 8     if ¬l-asscom(∘^a, ∘^b)
 9       if T(right(∘^a)) ∩ F_T(∘^a) ≠ ∅
10         CR(∘^b) += T(left(∘^a)) → T(right(∘^a)) ∩ F_T(∘^a)
11       else
12         CR(∘^b) += T(left(∘^a)) → T(right(∘^a))
13   for ∀ ∘^a ∈ STO(right(∘^b))
14     if ¬assoc(∘^b, ∘^a)
15       if T(right(∘^a)) ∩ F_T(∘^a) ≠ ∅
16         CR(∘^b) += T(left(∘^a)) → T(right(∘^a)) ∩ F_T(∘^a)
17       else
18         CR(∘^b) += T(left(∘^a)) → T(right(∘^a))
19     if ¬r-asscom(∘^b, ∘^a)
20       if T(left(∘^a)) ∩ F_T(∘^a) ≠ ∅
21         CR(∘^b) += T(right(∘^a)) → T(left(∘^a)) ∩ F_T(∘^a)
22       else
23         CR(∘^b) += T(right(∘^a)) → T(left(∘^a))
```

Figure 4.15: Pseudo code for CD-C

4.5.1 Larger TES, Faster Plan Generation

Simple plan generators like DPsub (Figure 3.3), DPsize (Figure 3.1), and MemoizationBasic [10] generate various pairs of relation sets (S_1, S_2) and then possibly reject some of them later on if they turn out to be invalid. This can be due to one of two reasons: either the pair is not a ccp, i.e., one of the connection tests applied by these algorithms fail, or a conflict detector is in place and there is a reordering conflict that prohibits joining (S_1, S_2), i.e., APPLICABLE fails. In both situations the effort for enumerating the join pair and conducting the tests that lead to its rejection is wasted. In Section 3.1 we have already stated that the connection tests can be avoided by using the query graph to enumerate only valid ccps. This is the approach followed by DPccp (Figure 3.4) and DPhyp (Figure 3.5) and the reason for their superior efficiency.

With a hypergraph-aware plan generator like DPhyp, we can oftentimes also avoid the second issue stated above, i.e., failing the applicability test. This can be achieved by using the TES (which are contained in CD-A and CD-C, where

they are possibly enlarged by rule elimination) to generate hyperedges instead of using them only within Applicable. Hence, the hyperedges can directly cover most of the possible conflicts, if not all (see Section 4.6 for precise numbers). The construction of the hyperedges proceeds as follows.

For every operator $\circ$, we construct a hyperedge (l, r) such that $r = \text{TES}(\circ) \cap \mathcal{T}(\text{right}(\circ)) = \text{R-TES}(\circ)$ and $l = \text{TES}(\circ) \setminus r = \text{L-TES}(\circ)$. Together with the nodes given by the relations in the input query, these hyperedges define the query graph that is the input to DPhyp. Two things are important to observe. First, in case of non-empty rule sets, the applicability test must still be executed. Second, since SES $\subseteq$ TES, no other hyperedges have to be constructed.

Let us now come to the question why larger TESs result in higher efficiency. The efficiency of an advanced plan generator is directly correlated to the number of ccps. Obviously, larger TESs result in larger hypernodes in the hyperedges (l, r). Potentially, a hyperedge (l, r) gives rise to a ccp (l, r) if both l and r induce connected subgraphs. Further, every (S_1, S_2) with $S_1 \supseteq l$, $S_2 \supseteq r$, $S_1 \cap S_2 = \emptyset$ is a potential ccp. Thus, enlarging (l, r) decreases the number of ccps.

4.5.2 Cross Products and Degenerate Predicates

Cross products and degenerate predicates require some special attention. Consider the example $(R_1 \times R_2) \bowtie_{1,3} (R_3 \ltimes_{3,4} R_4)$. So far, nothing prevents our plan generator from producing invalid plans such as $R_1 \bowtie_{1,3} (R_3 \ltimes_{3,4} (R_2 \times R_4))$. Note that in order to prevent this plan, we would have to detect conflicts on the "other side" of the plan. Since cross products and degenerate predicates should be rare in real queries, it suffices to produce correct plans. We have no ambition to explore the complete search space. Thus, we just want to make sure that in these abnormal cases, the plan generator still produces a correct plan. This can be achieved by conjunctively adding the check

$$\mathcal{T}(\text{left}(\circ)) \cap S_1 \neq \emptyset \wedge \mathcal{T}(\text{right}(\circ)) \cap S_2 \neq \emptyset$$

to the test for Applicable(S_1, S_2, $\circ$).

In the example given above, the test will fail when trying to apply the crossproduct joining R_2 and R_4, since

$$\mathcal{T}(\text{left}(\times)) \cap S_1 = \{R_1\} \cap \{R_2\} = \emptyset.$$

The additional condition prevents invalid plans, but a significant portion of the valid search space will not be explored if cross products are present in the initial operator tree. However, if the initial plan does not contain cross products or degenerate predicates, this test will always succeed, so in this case the core search space will still be explored completely. Moreover, the portion of the core search space explored by this approach is still larger than with Rao et al.'s approach, which consists of performing two separate runs of the plan generator for each of the arguments of a cross product [36, 37]. This hinders any reordering of cross products with other operators. The approach proposed by Moerkotte and Neumann cannot handle cross products at all [30].

There is a second issue concerning cross products. In some rare cases it might be beneficial to introduce cross products, even if the initial plan does not demand them. In these cases, we can proceed as proposed by Rao et al. [36, 37]: for each relation R, a *companion set* is calculated, which contains all relations that are connected to R only by inner join predicates. Within a companion set all join orders are valid, including those that require crossproducts.

4.6 Evaluation

When comparing the various conflict detectors discussed in the previous sections, the two properties that are of particular interest are correctness and completeness. We have already sketched out correctness proofs for all approaches, but we have also observed that not all of them are complete. This section provides some experimental results that serve to measure the number of valid plans missed by each conflict detector, if any. Aside from that, we also show experimentally that none of them produces incorrect plans.

In order to do so, we implemented a transformation-based plan generator. It exhaustively applies the transformation rules defined in Section 4.2 until no new plan can be generated. Additionally, we implemented all known conflict detectors and used them within DPsub (see Figure 3.3). To this end, we simply added a call to Applicable in Line 15 of the algorithm.

Moreover, we modified DPsub such that it does not prune dominated plans but instead keeps all generated plans. This set of plans is then compared to the set of plans generated by the transformation-based plan generator. This way, we found (1) invalid plans and (2) valid plans not generated by our DP-based plan generator equipped with a specific conflict detector. Since NEL/EEL only handles joins, antijoins and left outerjoins, but our conflict detectors allow for more operators, we ran our experiments for two distinct operator sets ($\{\bowtie, \triangleright, \fullouterjoin\}$ and $\{\bowtie, \ltimes, \triangleright, \fullouterjoin, \fullouterjoin\}$).

For any given set of operators, we generated all possible initial plans for a given number of relations (varied between 3 and 7). For each initial plan the different plan generators were executed. The generation of all initial plans for n relations proceeded in three steps. In a first step all integers from 1 to $\mathcal{C}(n-1)$ are unranked. $\mathcal{C}$ denotes the Catalan numbers. For the unranking we used the method proposed by Liebehenschel, which returns raw binary trees [26]. In a second step an operator from the respective operator set is attached to every inner node, making sure that every combination is generated exactly once. Finally, binary predicates are generated by exploring all possibilities to reference one relation from the operator's left subtree and one from its right subtree. We did not generate complex predicates, since they only simplify the enumeration of the core search space (see Section 4.2.2).

Tables 4.12 and 4.13 show the results. The columns contain the number of relations (n), the number of distinct queries (initial operator trees), the number of plans the transformation-based plan generator generates for these queries, and for each conflict detector the number of invalid plans (I) and the number of missing valid plans (M). The conflict detector EEL-F is the fixed version of

n	#Queries	#Plans	EEL		EEL-F		TES	
			I	M	I	M	I	M
3	26	88	0	0 %	0	1.14 %	0	0 %
4	344	4059	2	0 %	0	2.02 %	23	2.24 %
5	5834	301898	296	0 %	0	2.51 %	3964	6.47 %
6	117604	32175460	41108	0 %	0	2.70 %	605914	12.23 %
7	2708892	4598129499	6349126	0 %	0	2.71 %	99179293	19.05 %

n	#Queries	#Plans	CD-A		CD-B		CD-C	
			I	M	I	M	I	M
3	26	88	0	0 %	0	0 %	0	0 %
4	344	4059	0	3.30 %	0	2.02 %	0	0 %
5	5834	301898	0	8.54 %	0	5.38 %	0	0 %
6	117604	32175460	0	14.66 %	0	9.77 %	0	0 %
7	2708892	4598129499	0	21.06 %	0	15.04 %	0	0 %

Table 4.12: Small operator set: join, left outerjoin, antijoin

the original NEL/EEL approach. Additionally, Table 4.13 contains for CD-C the number of rule sets which are empty after applying rule simplifications as well as the number of non-empty rule sets.

From Table 4.12 we see that both the EEL/NEL approach and the SES/TES approach produce invalid plans. From Table 4.13 we see that CD-A and CD-B lose large fractions of the valid search space, but CD-C does not. We also see that about 70% of all rule sets are empty if rule simplification is applied. This means that most reordering conflicts can be covered with the TESs alone and we can encode these conflicts in the query graph to avoid the call to Applicable with a hypergraph-aware enumerator.

n	#Queries	#Plans	CD-A I	CD-A M
3	62	203	0	0
4	1114	11148	0	473 (4.24 %)
5	25056	934229	0	102019 (10.92 %)
6	661811	108294798	0	20113801 (18.57 %)
7	19846278	16448441514	0	4329578881 (26.32 %)

n	#Queries	#Plans	CD-B I	CD-B M
3	62	203	0	0
4	1114	11148	0	246 (2.21 %)
5	25056	934229	0	55725 (5.96 %)
6	661811	108294798	0	11868102 (10.96 %)
7	19846278	16448441514	0	2793701760 (16.98 %)

n	#Queries	#Plans	CD-C I	CD-C M	Rule Sets $\emptyset$	Rule Sets $\neg\emptyset$
3	62	203	0	0	107	17
4	1114	11148	0	0	2725	617
5	25056	934229	0	0	77484	22740
6	661811	108294798	0	0	2432717	876338
7	19846278	16448441514	0	0	83560096	35517572

Table 4.13: Large operator set: join, left/full outerjoin, semijoin, antijoin

5 Reordering Join and Grouping

The findings from the previous chapter constitute all the tools necessary to build a plan generator that is capable of correctly reordering join operators of any type. With efficient enumeration algorithms like DPhyp already existing, we consider the classic join ordering problem solved. This allows us to look into other aspects of plan generation. One of them is optimizing the placement of grouping operators. Instead of applying a single grouping after all joins have been applied, grouping operators can be pushed down in the operator tree to apply them before a join. This transformation can have a significant impact on the runtime of a large class of queries, namely analytical queries as they are often seen in the context of data warehouses. These queries usually contain a number of joins and a grouping in order to produce an aggregated result.

This chapter covers the problem of reordering join and grouping by first providing the necessary algebraic equivalences. Subsequently, different approaches for exploiting these equivalences in a DP-based plan generator are presented. They can be classified into heuristic approaches that do not guarantee an optimal solution, but add only little overhead to the plan generator and approaches that guarantee an optimal solution, but require sophisticated pruning strategies to keep the plan generator's runtime in check. Both approaches have in common that they can easily be added to any plan generator adhering to the component-based architecture described in Chapter 3.

Moreover, the pruning approaches described in Section 5.7 serve as a representative example of how plan properties can be used to extend the functionality of a DP-based plan generator.

5.1 Motivation

For a motivating example, consider the query against the TPC-H schema [5] shown in Figure 5.1. It counts the customers and suppliers of each nation. Since nations that have no suppliers or customers should also be included, a full outerjoin is applied. Figure 5.2 shows the corresponding operator tree.

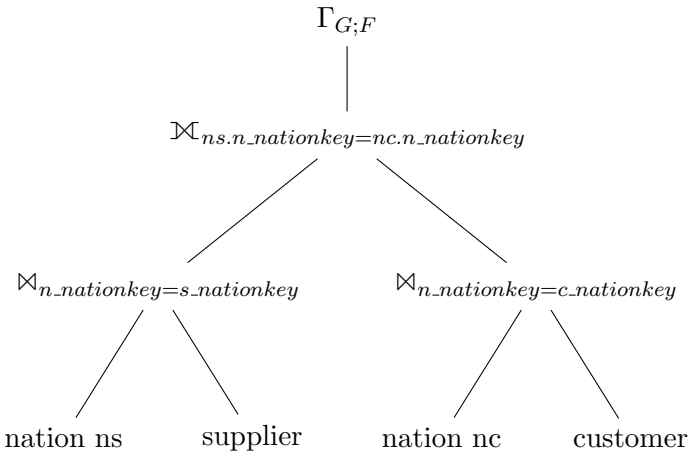
Manually rewriting the query to apply the grouping before the full outerjoin yields the query shown in Figure 5.3 and the corresponding operator tree in Figure 5.4

In HyPer [23] the execution time of the first query is 12,700 ms, whereas the second query takes only 14.5 ms. Similar results were obtained on a commercial disk-based system: 68,237 ms vs. 62 ms. All numbers were obtained with a TPC-H instance of scale factor 1 on commodity hardware. None of the two systems were optimized for benchmarking purposes.

The effect of this rewrite comes as no surprise, since pushing down groupings is a well-established query optimization technique. The main point here is that

```
select ns.n_name, nc.n_name,
  count(distinct c_custkey), count(distinct s_suppkey)
  from
(nation ns inner join supplier s
  on (ns.n_nationkey = s.s_nationkey))
full outer join
(nation nc inner join customer c
  on (nc.n_nationkey = c.c_nationkey))
on (ns.n_nationkey = nc.n_nationkey)
group by ns.n_name, nc.n_name
order by ns.n_name, nc.n_name
```

Figure 5.1: Query containing full outerjoin and group-by



with

$$\begin{aligned} G &= \{ns.n\_name, nc.n\_name\}, \\ F &= (count(distinct\ c\_custkey), count(distinct\ s\_suppkey)) \end{aligned}$$

Figure 5.2: Operator tree for query in Figure 5.1

only reorderings between grouping and inner join are known [2, 44, 45, 46, 47, 48]. Thus, the outerjoin typically constitutes a barrier to any reordering with grouping. Since in general, reordering grouping and outerjoins is not a correct rewrite, eliminating the barrier requires generalizing the definition of outerjoins. Section 5.3 provides all the details.

A quick complexity analysis shows that the free placement of grouping extends the search space of a plan generator substantially: a binary operator tree with n relations contains $2n - 2$ edges, and we can attach a grouping to each of these edges and on top of the root, resulting in $2n - 1$ possible positions

```
select ns.n_name, nc.n_name, scnt as suppliers,
  ccnt as customers
from
(nation ns inner join
  (select s_nationkey, count(distinct s_suppkey) as scnt
        from supplier group by s_nationkey) as x
on (ns.n_nationkey = x.s_nationkey))
full outer join
(nation nc inner join
  (select c_nationkey, count(distinct c_custkey) as ccnt
        from customer group by c_nationkey) as y
on (nc.n_nationkey = y.c_nationkey))
on ns.n_nationkey =  nc.n_nationkey
order by ns.n_name, nc.n_name
```

Figure 5.3: Rewritten query with pushed-down group-by

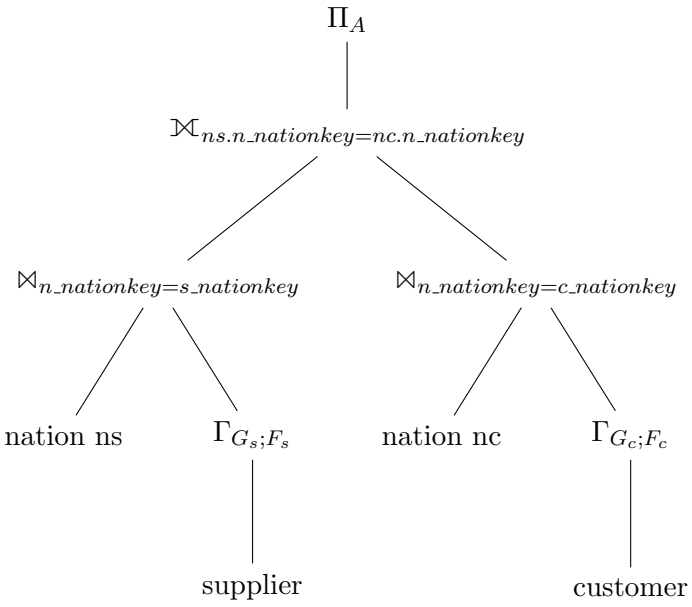
for a grouping. If one considers all valid combinations of these positions for every tree, the additional overhead caused by the optimal placement of grouping operators is a factor of $O(2^{2n-1})$. On the other hand, if one can infer at a certain position in the operator tree that the grouping attributes constitute a superkey, then a grouping at this position does not need to be considered because grouping by a key has no effect.

One possible strategy for dealing with the large search space is to abandon optimality and work with heuristics. Alternatively, one can derive optimality-preserving pruning criteria that allow for reducing the search space size and at the same time guarantee an optimal solution. In this chapter we examine both options. We also extend the plan generator further by incorporating another powerful transformation. It consists of replacing a sequence of join and grouping by a single groupjoin [33].

5.2 Properties of Aggregate Functions

Aggregate functions are applied to a group of tuples to aggregate their values in one common attribute to a single value. Some standard aggregate functions supported by SQL are *sum*, *count*, *min*, *max* and *avg*. Additionally, it is possible to specify how duplicates are treated by these functions using the *distinct* keyword as in sum(distinct), count(distinct) and so on. Since several aggregate functions are allowed in the *select* clause of a SQL query, we deal with vectors of aggregate functions, such as $F = (b_1 : sum(a), b_2 : count(*))$. Here, a denotes an attribute which is aggregated via sum and b_1, b_2 are attribute names for the aggregation results. If F_1 and F_2 are two vectors of aggregate functions, we denote their concatenation by $F_1 \circ F_2$.

The set of attributes provided by an expression e (e.g., a base relation) is denoted by $\mathcal{A}(e)$ and the set of attributes that occur freely in a predicate p or



with

$$\begin{aligned} A &= \{ns.n\_name, nc.n\_name, ccnt, scnt\}, \\ G_c &= \{c\_nationkey\}, \\ F_c &= (ccnt : count(distinct\ c\_custkey)), \\ G_s &= \{s\_nationkey\}, \\ F_s &= (scnt : count(distinct\ s\_suppkey)) \end{aligned}$$

Figure 5.4: Operator tree for query in Figure 5.3

aggregation vector F is denoted by $\mathcal{F}(p)$ or $\mathcal{F}(F)$, respectively. We introduce some properties of aggregate functions below.

5.2.1 Splittability

The following definition captures the intuition that we can split a vector of aggregate functions into two parts if each aggregate function accesses only attributes from one of two given alternative expressions.

Definition 9. *An aggregation vector F is splittable into F_1 and F_2 with respect to expressions e_1 and e_2 if*

1. $F = F_1 \circ F_2$,
2. $\mathcal{F}(F_1) \cap \mathcal{A}(e_2) = \emptyset$ *and*
3. $\mathcal{F}(F_2) \cap \mathcal{A}(e_1) = \emptyset$.

In this case we can evaluate F_1 on e_1 and F_2 on e_2. A special case occurs for *count(*)*, which accesses no attributes and can thus be added to both F_1 and F_2.

5.2.2 Decomposability

We define *decomposability* of an aggregate function as follows [3]:

Definition 10. *An aggregate function agg is decomposable if there exist aggregate functions* agg^1 *and* agg^2 *such that* $agg(Z) = agg^2(agg^1(X), agg^1(Y))$ *for bags of values* X, Y *and* Z *where* $Z = X \cup Y$.

In other words, if agg is decomposable, $agg(Z)$ can be computed independently on arbitrary subbags of Z and the partial results can be aggregated to yield the correct total result. For some aggregate functions, decomposability can easily be seen:

$$\begin{aligned} min(X \cup Y) &= min(min(X), min(Y)) \\ max(X \cup Y) &= max(max(X), max(Y)) \\ count(X \cup Y) &= sum(count(X), count(Y)) \\ sum(X \cup Y) &= sum(sum(X), sum(Y)) \end{aligned}$$

In contrast to the functions above, sum(distinct) and count(distinct) are not decomposable.

The treatment of avg is only slightly more complicated. If there are no null values present, SQL's avg is equivalent to $avg(X) = sum(X)/count(X)$. Since both sum and count are decomposable, we can decompose avg as follows:

$$avg(X \cup Y) = \frac{sum(sum(X), sum(Y))}{sum(count(X), count(Y))}.$$

If there exist null values, we need a slightly modified version of count that only counts tuples in which the aggregated attribute is not null. We denote this function by $count^{NN}$ and use it to decompose avg as follows:

$$avg(X \cup Y) = \frac{sum(sum(X), sum(Y))}{sum(count^{NN}(X), count^{NN}(Y))}.$$

5.2.3 Treatment of Duplicates

We have already seen that duplicates play a central role in correct aggregate processing. Thus, we define the following. An aggregate function f is called *duplicate-agnostic* if its result does *not* depend on whether there are duplicates in its argument, or not. Otherwise, it is called *duplicate-sensitive*. Yan and Larson use the terms *Class C* for duplicate-sensitive functions and *Class D* for duplicate-agnostic functions [44].

For SQL's aggregate functions, we have that

- min, max, sum(distinct), count(distinct), avg(distinct) are duplicate-agnostic and

- sum, count, avg are duplicate-sensitive.

If we want to decompose an aggregate function that is duplicate-sensitive, some care has to be taken. We express this through an operator *prime* ($'$) as follows. Let $F = (b_1 : agg_1(a_1), \ldots, b_m : agg_m(a_m))$ be an aggregation vector. Further, let c be some other attribute. In our case, c is an attribute holding the result of $count(*)$. Then, we define $F \otimes c$ as

$$F \otimes c := (b_1 : agg_1'(e_1), \ldots, b_m : agg_m'(e_m))$$

with

$$agg_i'(e_i) = \begin{cases} agg_i(e_i) & \text{if } agg_i \text{ is duplicate-agnostic} \\ agg_i(e_i * c) & \text{if } agg_i \text{ is } sum \\ sum(c) & \text{if } agg_i(e_i) = count(*) \end{cases}$$

and if $agg_i(e_i)$ is $count(e_i)$, then $agg_i'(e_i) := sum(e_i = null\,?\,0 : c)$.

5.3 Equivalences for Join and Grouping

This section is organized into two parts. The first part shows how to push down or pull up a grouping operator, whereas the second part shows how to eliminate an unnecessary top grouping operator. These findings have been published in [7]. The equivalences presented in this section are generally valid and form the basis for reordering join and grouping operators in any type of plan generator. In Section 5.5 we are going to see how they can be implemented in a DP-based plan generator.

5.3.1 The Outerjoin with Default Values

In general, pushing a grouping past a full or left outerjoin is not a valid transformation. This is why we generalize both operators in such a way that for tuples not finding a join partner, default values can be provided instead of null padding. More specifically, let $D^i = d_1^i : c_1^i, \ldots, d_k^i : c_k^i$, for $i = (1, 2)$ be two vectors assigning constants c_j to attributes d_j^i. With this, the left outerjoin with defaults is defined as follows:

$$e_1 \mathbin{⟕}_p^{D^2} e_2 := (e_1 \bowtie_p e_2) \cup ((e_1 \triangleright_p e_2) \times \{\bot_{\mathcal{A}(e_2)\setminus\mathcal{A}(D^2)} \circ [D^2]\} \tag{5.1}$$

The full outerjoin with defaults is defined analogously:

$$\begin{aligned} e_1 \mathbin{⟗}_p^{D^1;D^2} e_2 \quad := \quad & (e_1 \bowtie_p e_2) \\ & \cup ((e_1 \triangleright_p e_2) \times \{\bot_{\mathcal{A}(e_2)\setminus\mathcal{A}(D^2)} \circ [D^2]\} \\ & \cup (\{\bot_{\mathcal{A}(e_1)\setminus\mathcal{A}(D^1)} \circ [D^1]\} \times (e_2 \triangleright_p e_1)) \end{aligned} \tag{5.2}$$

As before, we denote tuple concatenation by $\circ$ and the tuple containing null in all attributes from attribute set A by $\bot_A$. Implementing these operators in a system that already supports outerjoins is straightforward. Figure 5.5 provides examples for both operators, where the value 7 is used as the default value for attributes b and e.

R_1

a	b	c
0	0	1
1	0	1
2	1	3
3	2	3

R_2

d	e	f
0	0	1
1	1	1
2	2	1
3	4	2

$R_1 \bowtie^{e:7}_{R_1.a=R_2.e} R_2$

a	b	c	d	e	f
0	0	1	0	0	1
1	0	1	1	1	1
2	1	3	2	2	1
3	2	3	-	7	-

$R_1 \mathbb{K}^{b:7;e:7}_{R_1.a=R_2.e} R_2$

a	b	c	d	e	f
0	0	1	0	0	1
1	0	1	1	1	1
2	1	3	2	2	1
3	2	3	-	7	-
-	7	-	3	4	2

Figure 5.5: Example for the left and full outerjoin with default values

5.3.2 Pushing Group-By

Since the work by Yan and Larson [44, 45, 46, 47, 48] is the most general, we take it as the basis for our work. In doing so, we also adopt their naming conventions: we use the term *eager aggregation* for pushing down a grouping operator in the operator tree and *lazy aggregation* for the inverse transformation.

Figures 5.6, 5.7 and 5.8 show all known and new equivalences. The nine equivalences already known from Yan and Larson can be recognized by the inner join on their left-hand sides. The different section headings within the figures were also proposed by Yan and Larson (except for *Others*). A special case of Equivalence 5.13 is already known from previous work [15].

Within the equivalences a couple of simple abbreviations as well as some conventions occur. We introduce them in this short paragraph. By G we denote the set of grouping attributes, by F a vector of aggregate functions, and by p a join predicate. The grouping attributes coming from expression e_i are denoted by G_i, i.e., $G_i = \mathcal{A}(e_i) \cap G$. The join attributes from expression e_i are denoted by J_i, i.e., $J_i = \bigcup_p \mathcal{F}(p) \cap \mathcal{A}(e_i)$, with p being a join predicate contained in the input query. The union of the grouping and join attributes from e_i is denoted by $G_i^+ = G_i \cup J_i$. If F_1 and F_2 occur in an equivalence, then the equivalence is based on the assumption that F is splittable into F_1 and F_2. If F_1 or F_2 does not occur in an equivalence, it is assumed to be empty. If for some $i \in \{1, 2\}$, F_i^1 and F_i^2 occur in some equivalence, the equivalence requires that F_i is decomposable into F_i^1 and F_i^2. Last but not least, $\perp$ abbreviates a special tuple that returns the null value for every attribute.

Example 1: Join

Figure 5.9 shows two relations e_1 and e_2, which will be used to illustrate Equivalence 5.3 as well as Equivalence 5.5.

Eager and Lazy Groupby-Count

$$\Gamma_{G;F}(e_1 \bowtie_p e_2) \equiv \Gamma_{G;(F_2 \otimes c_1) \circ F_1^2}(\Gamma_{G_1^+;F_1^1 \circ (c_1:count(*))}(e_1) \bowtie_p e_2) \quad (5.3)$$

$$\Gamma_{G;F}(e_1 \bowtie_p e_2) \equiv \Gamma_{G;(F_2 \otimes c_1) \circ F_1^2}(\Gamma_{G_1^+;F_1^1 \circ (c_1:count(*))}(e_1) \bowtie_p e_2) \quad (5.4)$$

$$\Gamma_{G;F}(e_1 \bowtie_p e_2) \equiv \Gamma_{G;(F_2 \otimes c_1) \circ F_1^2}(\Gamma_{G_1^+;F_1^1 \circ (c_1:count(*))}(e_1) \bowtie_q^{F_1^1(\{\bot\}),c_1:1;-} e_2) \quad (5.5)$$

$$\Gamma_{G;F}(e_1 \bowtie_p e_2) \equiv \Gamma_{G;(F_1 \otimes c_2) \circ F_2^2}(e_1 \bowtie_p \Gamma_{G_2^+;F_2^1 \circ (c_2:count(*))}(e_2)) \quad (5.6)$$

$$\Gamma_{G;F}(e_1 \bowtie_p e_2) \equiv \Gamma_{G;(F_1 \otimes c_2) \circ F_2^2}(e_1 \bowtie_p^{F_2^1(\{\bot\}),c_2:1} \Gamma_{G_2^+;F_2^1 \circ (c_2:count(*))}(e_2)) \quad (5.7)$$

$$\Gamma_{G;F}(e_1 \bowtie_p e_2) \equiv \Gamma_{G;(F_1 \otimes c_2) \circ F_2^2}(e_1 \bowtie_q^{-;F_2^1(\{\bot\}),c_2:1} \Gamma_{G_2^+;F_2^1 \circ (c_2:count(*))}(e_2)) \quad (5.8)$$

Eager and Lazy Group-by

$$\Gamma_{G;F}(e_1 \bowtie_p e_2) \equiv \Gamma_{G;F_1^2}(\Gamma_{G_1^+;F_1^1}(e_1) \bowtie_p e_2) \quad (5.9)$$

$$\Gamma_{G;F}(e_1 \bowtie_p e_2) \equiv \Gamma_{G;F_1^2}(\Gamma_{G_1^+;F_1^1}(e_1) \bowtie_p e_2) \quad (5.10)$$

$$\Gamma_{G;F}(e_1 \bowtie_p e_2) \equiv \Gamma_{G;F_1^2}(\Gamma_{G_1^+;F_1^1}(e_1) \bowtie_q^{F_1^1(\{\bot\});-} e_2) \quad (5.11)$$

$$\Gamma_{G;F}(e_1 \bowtie_p e_2) \equiv \Gamma_{G;F_2^2}(e_1 \bowtie_p \Gamma_{G_2^+;F_2^1}(e_2)) \quad (5.12)$$

$$\Gamma_{G;F}(e_1 \bowtie_p e_2) \equiv \Gamma_{G;F_2^2}(e_1 \bowtie_p^{F_2^1(\{\bot\})} \Gamma_{G_2^+;F_2^1}(e_2)) \quad (5.13)$$

$$\Gamma_{G;F}(e_1 \bowtie_p e_2) \equiv \Gamma_{G;F_2^2}(e_1 \bowtie_q^{-;F_2^1(\{\bot\})} \Gamma_{G_2^+;F_2^1}(e_2)) \quad (5.14)$$

Eager and Lazy Count

$$\Gamma_{G;F}(e_1 \bowtie_p e_2) \equiv \Gamma_{G;(F_2 \otimes c_1)}(\Gamma_{G_1^+;c_1:count(*)}(e_1) \bowtie_p e_2) \quad (5.15)$$

$$\Gamma_{G;F}(e_1 \bowtie_p e_2) \equiv \Gamma_{G;(F_2 \otimes c_1)}(\Gamma_{G_1^+;(c_1:count(*))}(e_1) \bowtie_p e_2) \quad (5.16)$$

$$\Gamma_{G;F}(e_1 \bowtie_p e_2) \equiv \Gamma_{G;(F_2 \otimes c_1)}(\Gamma_{G_1^+;(c_1:count(*))}(e_1) \bowtie_q^{c_1:1;-} e_2) \quad (5.17)$$

$$\Gamma_{G;F}(e_1 \bowtie_p e_2) \equiv \Gamma_{G;(F_1 \otimes c_2)}(e_1 \bowtie_p \Gamma_{G_2^+;c_2:count(*)}(e_2)) \quad (5.18)$$

$$\Gamma_{G;F}(e_1 \bowtie_p e_2) \equiv \Gamma_{G;(F_1 \otimes c_2)}(e_1 \bowtie_p^{c_2:1} \Gamma_{G_2^+;c_2:count(*)}(e_2)) \quad (5.19)$$

$$\Gamma_{G;F}(e_1 \bowtie_p e_2) \equiv \Gamma_{G;(F_1 \otimes c_2)}(e_1 \bowtie_q^{-;c_2:1} \Gamma_{G_2^+;(c_2:count(*))}(e_2)) \quad (5.20)$$

Figure 5.6: Equivalences for join and grouping (1/3)

Let us start with Equivalence 5.3. For now, we only look at the top equivalences above each relation and ignore the tuples below the separating horizontal line. Relations e_1 and e_2 at the top of Figure 5.9 serve as input. The calculation of the result of the left-hand side of Equivalence 5.3 is rather straightforward. Relation e_3 gives the result of the join $e_1 \bowtie_{j_1=j_2} e_2$. The result is then grouped by $\Gamma_{g_1,g_2;F}(e_3)$ for the aggregation vector $F = k : count(*), b_1 : sum(a_1), b_2 : sum(a_2)$. The result is given as e_4. In our join example it consists of a single tuple. We have intentionally chosen an example with a single group, since multiple groups make the example longer but do not give more insights.

Before we start the calculation of the right-hand side of Equivalence 5.3, we take apart the grouping attributes and the aggregation vector F. Among the grouping attributes $G = \{g_1, g_2\}$, only g_1 occurs in e_1. The only join attribute in the join predicate $j_1 = j_2$ from e_1 is j_1. Thus, $G_1^+ = \{g_1, j_1\}$. The aggregation vector F can be split into F_1, which references only attributes in e_1, and F_2, which references only attributes in e_2. This results in $F_1 = (k : count(*), b_1 : sum(a_1))$ and it does not matter whether we add k to F_1 or F_2, since it does

Double Eager and Double Lazy

$$\Gamma_{G;F}(e_1 \bowtie_p e_2) \equiv \Gamma_{G;(F_1^2 \otimes c_2)}(\Gamma_{G_1^+;F_1^1}(e_1) \bowtie_p \Gamma_{G_2^+;c_2:count(*)}(e_2)) \quad (5.21)$$

$$\Gamma_{G;F}(e_1 \bowtie_p e_2) \equiv \Gamma_{G;(F_1^2 \otimes c_2)}(\Gamma_{G_1^+;F_1^1}(e_1) \bowtie_p^{c_2:1} \Gamma_{G_2^+;c_2:count(*)}(e_2)) \quad (5.22)$$

$$\Gamma_{G;F}(e_1 \bowtie_p e_2) \equiv \Gamma_{G;(F_1^2 \otimes c_2)}(\Gamma_{G_1^+;F_1^1}(e_1) \bowtie_q^{F_1^1(\{\bot\});c_2:1} \Gamma_{G_2^+;(c_2:count(*))}(e_2)) \quad (5.23)$$

$$\Gamma_{G;F}(e_1 \bowtie_p e_2) \equiv \Gamma_{G;(F_2^2 \otimes c_1)}(\Gamma_{G_1^+;c_1:count(*)}(e_1) \bowtie_p \Gamma_{G_2^+;F_2^1}(e_2)) \quad (5.24)$$

$$\Gamma_{G;F}(e_1 \bowtie_p e_2) \equiv \Gamma_{G;(F_2^2 \otimes c_1)}(\Gamma_{G_1^+;c_1:count(*)}(e_1) \bowtie_p^{F_2^1(\{\bot\})} \Gamma_{G_2^+;F_2^1}(e_2)) \quad (5.25)$$

$$\Gamma_{G;F}(e_1 \bowtie_p e_2) \equiv \Gamma_{G;(F_2^2 \otimes c_1)}(\Gamma_{G_1^+;(c_1:count(*))}(e_1) \bowtie_q^{c_1:1;F_2^1(\{\bot\})} \Gamma_{G_2^+;F_2^1}(e_2)) \quad (5.26)$$

Eager and Lazy Split (with $\Gamma^2 := \Gamma_{G;(F_1^2 \otimes c_2) \circ (F_2^2 \otimes c_1)}$):

$$\Gamma_{G;F}(e_1 \bowtie_p e_2) \equiv \Gamma_{G;(F_1^2 \otimes c_2) \circ (F_2^2 \otimes c_1)}(\Gamma_{G_1^+;F_1^1 \circ (c_1:count(*))}(e_1) \bowtie_p \Gamma_{G_2^+;F_2^1 \circ (c_2:count(*))}(e_2)) \quad (5.27)$$

$$\Gamma_{G;F}(e_1 \bowtie_p e_2) \equiv \Gamma_{G;(F_1^2 \otimes c_2) \circ (F_2^2 \otimes c_1)}(\Gamma_{G_1^+;F_1^1 \circ (c_1:count(*))}(e_1) \bowtie_p^{F_2^1(\{\bot\}),c_2:1} \Gamma_{G_2^+;F_2^1 \circ (c_2:count(*))}(e_2)) \quad (5.28)$$

$$\Gamma_{G;F}(e_1 \bowtie_p e_2) \equiv \Gamma^2(\Gamma_{G_1^+;F_1^{1,1} \circ (c_1:count(*))}(e_1) \bowtie_q^{F_1^{1,1}(\{\bot\}),c_1:1;F_2^{1,1}(\{\bot\}),c_2:1} \Gamma_{G_2^+;F_2^{1,1} \circ (c_2:count(*))}(e_2)) \quad (5.29)$$

Figure 5.7: Equivalences for join and grouping (2/3)

Others

$$\Gamma_{G;F}(e_1 \ltimes_p e_2) \equiv \Gamma_{G;F}(e_1) \ltimes_p e_2 \quad (\mathcal{F}(q) \cap \mathcal{A}(e_1)) \subseteq G \quad (5.30)$$

$$\Gamma_{G;F}(e_1 \rhd_p e_2) \equiv \Gamma_{G;F}(e_1) \rhd_p e_2 \quad (\mathcal{F}(q) \cap \mathcal{A}(e_1)) \subseteq G \quad (5.31)$$

$$\Gamma_{G;F}(e_1 \bowtie^{\sqcap}_{J_1 \theta J_2;\overline{F}} e_2) \equiv \Gamma_{G;(F_2 \otimes c_1) \circ F_1^2}(\Gamma_{G_1^+;F_1^1 \circ (c_1:count(*))}(e_1) \bowtie^{\sqcap}_{J_1 \theta J_2;\overline{F}} e_2)) \quad (5.32)$$

$$\Gamma_{G;F}(e_1 \bowtie^{\sqcap}_{J_1 \theta J_2;\overline{F}} e_2) \equiv \Gamma_{G;F_1^2}(\Gamma_{G_1^+;F_1^1}(e_1) \bowtie^{\sqcap}_{J_1 \theta J_2;\overline{F}} e_2)) \quad (5.33)$$

$$\Gamma_{G;F}(e_1 \bowtie^{\sqcap}_{J_1 \theta J_2;\overline{F}} e_2) \equiv \Gamma_{G;(F_2 \otimes c_1)}(\Gamma_{G_1^+;(c_1:count(*))}(e_1) \bowtie^{\sqcap}_{J_1 \theta J_2;\overline{F}} e_2)) \quad (5.34)$$

Figure 5.8: Equivalences for join and grouping (3/3)

not reference any attributes. Next, we need to decompose F_1 into F_1^1 and F_1^2 according to the insights of Section 5.2. This results in $F_1^1 = (k' : count(*), b_1' : sum(a_1))$ and $F_1^2 = (k : \text{sum}(k'), b_1 : \text{sum}(b_1'))$. The inner grouping operator of Equivalence 5.3 requires us to add an attribute $c_1 : count(*)$ to F_1^1, which we abbreviate by F_X. Since there already exists one $count(*)$, the result of which is stored in k', we keep only one of them in Figure 5.9 and call the corresponding attribute k'/c_1. This finishes our preprocessing on the aggregate functions of the inner grouping operator. Its result, consisting of two tuples, is given as relation e_5 in Figure 5.9. The next step consists of calculating the join $e_5 \bowtie_{j_1=j_2} e_2$. As this is rather straightforward, we just give the result (relation e_6). The final step is again a little more complex. Equivalence 5.3 requires us to calculate $F_2 \otimes c_1$. Looking back at the end of Section 5.2, we see that sum is duplicate sensitive and that $F_2 \otimes c_1 = b_2 : sum(c_1 * a_2)$. Concatenating this

aggregation vector with F_1^2, as demanded by Equivalence 5.3, gives us F_Y as specified in Figure 5.9. The final result of the right-hand side of Equivalence 5.3, calculated as $e_7 = \Gamma_{g_1,g_2;F_Y}(e_6)$, is given in Figure 5.9. Note that it is equal to the result of the left-hand side (e_4).

Example 2: Full Outerjoin

The second example reuses the relations e_1 and e_2 given in Figure 5.9. But this time we calculate the full outerjoin instead of the inner join and we apply Equivalence 5.5. The corresponding expressions are given in the lower header line of each relation. Now all tuples in each e_i are relevant, including those below the separating horizontal line. The result of $e_1 \mathbin{⟗}_{j_1=j_2} e_2$ is given in e_3', whereby we denote null by '-'. We can reuse all the different aggregation vectors derived in the previous example. The only new calculation that needs to be done is the one for the default values for the full outerjoin on the right-hand side of Equivalence 5.5. The equivalence defines default values in case a tuple t from e_2 does not find a join partner from the other side. All c_1 values of orphaned e_2 tuples become 1. Further, $F_1^1(\{\bot\})$ evaluates to 1 for k (count(*) on a relation with a single element), and null for a_2, since SQL's sum returns null for sets which contain only null values. Thus prepared, we can calculate the right-hand side of Equivalence 5.5 via e_5 and e_6'. For the latter we use a full outerjoin with default. Finally, e_7' is calculated by grouping e_6', leading to the same result as e_4'.

Remarks

The main equivalences are those under the heading *Eager and Lazy Group-by–Count*. They fall into two classes depending on whether the grouping is pushed into the left or the right argument of the join. For commutative operators such as inner join and full outerjoin, deriving one from the other is simple. For non-commutative operators such as the left outerjoin, we can combine both equivalences to push the grouping into both arguments. The resulting equivalences are given under the heading *Eager and Lazy Split*. The equivalences between these two blocks are specializations in case an aggregation vector F accesses attributes from only one input. In this case either F_1 or F_2 is empty and the equivalences can be simplified. These simplifications are shown in the blocks *Eager and Lazy Group-By*, *Eager and Lazy Count* and *Double Eager and Double Lazy*. The block termed *Others* shows how to push the grouping operator into the left semijoin, left antijoin and the groupjoin. The latter requires another aggregation vector $\overline{F}$. They all have in common that after they have been applied, only the attributes from their left input are accessible. Thus, the grouping operator can only be pushed into their left argument.

5.3.3 Eliminating the Top Grouping

We wish to eliminate a top grouping from an expression of the form $\Gamma_{G,F}(e)$ with an aggregation vector $F = (b_1 : agg_1(a_1), \ldots, b_k : agg_k(a_k))$. Clearly, this is only possible if G is a superkey for e and e is duplicate-free, since in this case,

e_1

g_1	j_1	a_1
1	1	2
1	2	4
1	2	8
1	3	7

e_2

g_2	j_2	a_2
1	1	2
1	1	4
1	2	8
1	4	9

$e_3 := e_1 \bowtie_{j_1=j_2} e_2$

$e'_3 := e_1 \between_{j_1=j_2} e_2$

g_1	j_1	a_1	g_2	j_2	a_2
1	1	2	1	1	2
1	1	2	1	1	4
1	2	4	1	2	8
1	2	8	1	2	8
1	3	7	-	-	-
-	-	-	1	4	9

$e_4 := \Gamma_{g_1,g_2;F}(e_3)$

$e'_4 := \Gamma_{g_1,g_2;F}(e'_3)$

g_1	g_2	k	b_1	b_2
1	1	4	16	22
1	-	1	7	-
-	1	1	-	9

$e_5 := \Gamma_{g_1,j_1;F_X}(e_1)$

g_1	j_1	k'/c_1	b'_1
1	1	1	2
1	2	2	12
1	3	1	7

$e_7 := \Gamma_{g_1,g_2;F_Y}(e_6)$

$e'_7 := \Gamma_{g_1,g_2;F_Y}(e'_6)$

g_1	g_2	k	b_1	b_2
1	1	4	16	22
1	-	1	7	-
-	1	1	-	9

$e_6 := e_5 \bowtie_{j_1=j_2} e_2$

$e'_6 := e_5 \between^{F_1^1(\{\bot\}),c_1:1;-}_{j_1=j_2} e_2$

g_1	j_1	k'/c_1	b'_1	g_2	j_2	a_2
1	1	1	2	1	1	2
1	1	1	2	1	1	4
1	2	2	12	1	2	8
1	3	1	7	-	-	-
-	-	1^*	-	1	4	9

$$
\begin{aligned}
F &= k : \text{count}(*), b_1 : \text{sum}(a_1), b_2 : \text{sum}(a_2),\\
F_1 &= k : \text{count}(*), b_1 : \text{sum}(a_1),\\
F_2 &= b_2 : \text{sum}(a_2),\\
F_1^1 &= k' : \text{count}(*), b'_1 : \text{sum}(a_1),\\
F_1^2 &= k : \text{sum}(k'), b_1 : \text{sum}(b'_1),\\
F_X &= F_1^1 \circ (c_1 : \text{count}(*)),\\
F_Y &= (F_2 \otimes c_1) \circ F_1^2\\
&= b_2 : \text{sum}(c_1 * a_2), k : \text{sum}(k'), b_1 : \text{sum}(b'_1),\\
G &= \{g_1, g_2\},\\
G_1^+ &= \{g_1, j_1\}.
\end{aligned}
$$

Figure 5.9: Example for Equivalences 5.3 and 5.5

there exists exactly one tuple in e for each group. The only detail left is the application of the aggregation vector F. This can be done via a map operator as in

$$\Gamma_{G;F}(e) \equiv \Pi_C(\chi_{\hat{F}}(e)), \tag{5.35}$$

with $C = G \cup \{b_1, \ldots, b_k\}$ and $\hat{F}$ defined as an aggregation vector meant to calculate the results of some aggregate functions applied to single values:

$$\hat{F} := (b_1 : agg_1(\{a_1\}), \ldots, b_k : agg_k(\{a_k\})).$$

Remark. In general, K is a superkey for relation R if $K \to \mathcal{A}(R)$ holds. In SQL, a declaration of a primary key or a uniqueness constraint implies not only a superkey but also that the relation is duplicate-free.

5.4 Equivalences for the Groupjoin

This section covers the algebraic equivalences necessary to replace a sequence of left outerjoin and grouping or inner join and grouping by a groupjoin. The corresponding equivalences have already been thoroughly described in previous work [33]. However, each of these equivalences comes with a number of preconditions that are expressed with the help of functional dependencies. We aim to simplify these preconditions by reformulating them in terms of keys, which are easier to maintain during plan generation.

5.4.1 Replacing Group-By and Left Outerjoin by Groupjoin

We can replace a sequence of left outerjoin and grouping by a single groupjoin [33]:

$$\Gamma_{G;F}(e_1 \mathbin{⟕}_{A_1=A_2} e_2) \equiv \Pi_C(e_1 \mathbin{⟕\Gamma}_{A_1=A_2;F} e_2) \tag{5.36}$$

if

1. $G \to G_2^+$ holds in $e_1 \mathbin{⟕}_{A_1=A_2} e_2$,
2. $G_1, G_2^+ \to \mathrm{TID}(e_1)$ holds in $e_1 \mathbin{⟕}_{A_1=A_2} e_2$,
3. $A_2 \to G_2^+$ holds in e_2,
4. $\mathcal{F}(F) \subseteq \mathcal{A}(e_2)$ and
5. $F(\emptyset) = F(\{\bot\})$.

We denote by $\mathrm{TID}(e)$ the tuple identifier for e. All these requirements are mandatory [33].

The preconditions for Equivalence 5.36 refer to functional dependencies holding in the join result. As we are going to see subsequently, we may choose to avoid the complexity of computing functional dependencies and maintain only information about superkeys instead. Therefore, we provide a simplified set of requirements expressed in terms of superkeys and prove that they imply the requirements for Equivalence 5.36. Note that the two sets of requirements are not equivalent [9].

Theorem 1.

$$\Gamma_{G;F}(e_1 \bowtie_{A_1=A_2} e_2) \equiv \Pi_C(e_1 \ \bowtie\!\!\!\!\!\!\Gamma_{A_1=A_2;F} \ e_2) \tag{5.37}$$

if

1. $(A_2 \subseteq G) \vee (A_1 \subseteq G \wedge G_2 = \emptyset)$,
2. $\exists K \in \kappa(e_1), K \subseteq G$,
3. $(\exists K \in \kappa(e_2), K \subseteq A_2) \vee (G_2 = \emptyset)$,
4. $\mathcal{F}(F) \subseteq \mathcal{A}(e_2)$ *and*
5. $F(\emptyset) = F(\{\perp\})$.

We denote by $\kappa(e)$ the set of superkeys for a relation defined by expression e.

Proof. Since the last two constraints for Equivalences 5.36 and 5.37 are equal, we only have to prove that the first three constraints from Equivalence 5.37 imply the first three from Equivalence 5.36. We refer to the different requirements by the number of the respective equivalence followed by the number of the requirement. For example, 5.37::1 refers to the first requirement listed under Equivalence 5.37. We prove each implication in a separate paragraph.

5.37::1 ⇒ 5.36::1 This requirement can be fulfilled in two ways. The first case is if $A_2 \subseteq G$ holds, which follows from the argumentation below:

$$\begin{aligned} & A_2 \subseteq G_2 \\ \Rightarrow \quad & G_2 \rightarrow A_2 \\ \Rightarrow \quad & G_1, G_2 \rightarrow A_2 \\ \Leftrightarrow \quad & G_1, G_2 \rightarrow G_2, A_2 \\ \Leftrightarrow \quad & G \rightarrow G_2^+. \end{aligned}$$

The second case is if $G_2 = \emptyset$ and $A_1 \subseteq G$. Then, $G \rightarrow G_2^+$ becomes $G \rightarrow A_2$. Since $A_1 = A_2$ or $A_2 = \perp_{\mathcal{A}(A_2)}$ after applying the join, this is fulfilled if $A_1 \subseteq G$ holds. We thereby assume that two attributes are equal if they agree in value or they are both null, as suggested by Paulley [35].

5.37::2 ⇒ 5.36::2 The second requirement can be strengthened to $G \rightarrow \mathrm{TID}(e_1)$. In other words, G has to be a superkey for e_1. Again, we express this in terms of superkeys: $\exists K \in \kappa(e_1)$, $K \subseteq G$.

5.37::3 ⇒ 5.36::3 If $G_2 = \emptyset$, then $G_2^+ = A_2$ and the third requirement is clearly fulfilled. If $G_2 \neq \emptyset$, the third requirement is fulfilled if in addition A_2 is a superkey for e_2, i.e., $A_2 \rightarrow \mathcal{A}(e_2)$. This can be expressed as follows: $K \in \kappa(e_2), K \subseteq A_2$. □

5.4.2 Replacing Group-By and Inner Join by Groupjoin

We can replace a sequence of inner join and grouping by a single groupjoin [33]:

$$\Gamma_{G;F}(e_1 \bowtie_{A_1=A_2} e_2) \equiv \Pi_C(\sigma_{c_2>0}(e_1 \ \bowtie\!\!\!\!\!\Gamma_{A_1=A_2;F\circ(c_2:count(*))} \ e_2)) \tag{5.38}$$

if

1. $G \rightarrow G_2^+$ holds in $e_1 \bowtie_{A_1=A_2} e_2$,
2. $G_1, G_2^+ \rightarrow \mathrm{TID}(e_1)$ holds in $e_1 \bowtie_{A_1=A_2} e_2$,
3. $A_2 \rightarrow G_2^+$ holds in e_2 and
4. $\mathcal{F}(F) \subseteq \mathcal{A}(e_2)$.

Again, we aim to simplify these requirements such that only information about superkeys is necessary to check them during plan generation [9].

Theorem 2.

$$\Gamma_{G;F}(e_1 \bowtie_{A_1=A_2} e_2) \equiv \Pi_C(\sigma_{c_2>0}(e_1 \ \bowtie\!\!\!\!\!\Gamma_{A_1=A_2;F\circ(c_2:count(*))} \ e_2)) \tag{5.39}$$

if

1. $(A_2 \subseteq G) \vee (A_1 \subseteq G \wedge G_2 = \emptyset)$,
2. $\exists K \in \kappa(e_1), K \subseteq G$,
3. $(\exists K \in \kappa(e_2), K \subseteq A_2) \vee (G_2 = \emptyset)$,
4. $\mathcal{F}(F) \subseteq \mathcal{A}(e_2)$.

The proof is identical to the one for Theorem 1.

Corollary 1.

$$\Gamma_{G;F}(e_1 \bowtie_{A_1=A_2} e_2) \equiv \Pi_C(e_1 \ \bowtie\!\!\!\!\!\Gamma_{A_1=A_2;F\circ(c_2:count(*))} \ e_2) \tag{5.40}$$

if

1. $(A_2 \subseteq G) \vee (A_1 \subseteq G \wedge G_2 = \emptyset)$,
2. $\exists K \in \kappa(e_1), K \subseteq G$,
3. $(\exists K \in \kappa(e_2), K \subseteq A_2) \vee (G_2 = \emptyset)$,
4. $(e_1 \ltimes_{A_1=A_2} e_2) = e_1$ *and*
5. $\mathcal{F}(F) \subseteq \mathcal{A}(e_2)$.

Proof. The only difference between Theorem 2 and Corollary 1 is that the selection after the groupjoin is omitted in the latter. In Equivalence 5.39 the selection is needed to remove those tuples from the groupjoin result that result from a tuple from e_1 not having a join partner in e_2. These tuples are contained in the result of the groupjoin (see Definition 2.7), but not in the result of a join followed by a grouping. If the fourth condition stated above holds, i.e., $(e_1 \ltimes_{A_1=A_2} e_2) = e_1$, all tuples from e_1 find a join partner in e_2 and the selection can be omitted. This is the case if there is a foreign key constraint where A_1 references A_2 and no selection was applied to e_2 before the join. □

Some Remarks Concerning the Groupjoin

Information on how to implement the groupjoin can be found in previous work [33]. There, a straightforward implementation combining join and grouping is proposed. One system that currently supports the groupjoin is HyPer [23].

In general, the groupjoin is most effective if the result of the join is large compared to the result of the following grouping operator. In this case combining the two operators saves the construction of a large intermediate result.

5.5 Implementation in a Plan Generator

In this section we put the theoretical findings from the previous sections into practice by describing a DP-based plan generator that is capable of reordering join and grouping operators and introducing groupjoins. First, we take a look at two heuristic approaches that do not guarantee an optimal solution. Subsequently, we will see how optimality can be guaranteed and how the exponential overhead coming along with this can be mitigated. The algorithms in this chapter have been published in [7, 8].

5.5.1 Enumerating Join Trees with Pushed-Down Grouping

We begin by introducing the routine OpTrees (Fig. 5.10). Its arguments are two join trees T_1 and T_2 as well as a join operator $\circ_p$. The result consists of a set of at most four trees, which join T_1 and T_2 taking all possible variants of eager aggregation into account.

The relation sets S_1 and S_2 are obtained from T_1 and T_2, respectively, by extracting their leaf nodes. This is denoted by $\mathcal{T}(T)$ for a tree T. The first tree is the one joining T_1 and T_2 using $\circ_p$ without any grouping.

Line 7 is only executed if the current relation set contains all relations in the input query. In this case we have to add another grouping on top of $\circ_p$ if and only if the grouping attributes do not comprise a key (see Section 5.3.3). This is checked by calling NeedsGrouping, which is listed in Figure 5.11.

The next tree is the one that groups the left argument before the join. In order to do so, we have to make sure that the corresponding transformation is valid. This is accomplished by calling the subroutine Valid, which implements the equivalences from Section 5.3. According to the equivalences, a grouping can be pushed into the left and/or right argument of almost all join operators. However, the correct calculation of the aggregate functions has to be ensured by adequately decomposing and splitting the aggregation vector. Thus, Valid needs to take the aggregate functions' properties into consideration. If an aggregate function is not decomposable, but this is required for the respective transformation, or the aggregation vector is not splittable as required, the transformation is not applied.

Additionally, we have to avoid the case in which the grouping attributes G_i^+ form a key for the set S_i, with $i \in \{1, 2\}$, because then the grouping has no effect. And again, if necessary, we have to add a grouping on top. Once the

```
OpTrees(T1, T2, ∘p)
 1  S1 = T(T1)
 2  S2 = T(T2)
 3  S = S1 ∪ S2
 4  Trees = ∅
 5  NewTree = (T1 ∘p T2)
 6  if S == R ∧ NeedsGrouping(G, NewTree)
 7    NewTree = (ΓG(NewTree))
 8  Trees.Insert(NewTree)
 9  NewTree = Γ_{G1+}(T1) ∘p T2
10  if Valid(NewTree) ∧ NeedsGrouping(G1+, NewTree)
11    if S == R ∧ NeedsGrouping(G, NewTree)
12      NewTree = (ΓG(NewTree))
13    Trees.Insert(NewTree)
14  NewTree = T1 ∘p Γ_{G2+}(T2)
15  if Valid(NewTree) ∧ NeedsGrouping(G2+, NewTree)
16    if S == R ∧ NeedsGrouping(G, NewTree)
17      NewTree = (ΓG(NewTree))
18    Trees.insert(NewTree)
19  NewTree = Γ_{G1+}(T1) ∘p Γ_{G2+}(T2)
20  if Valid(NewTree)
        ∧ NeedsGrouping(G1+, NewTree)
        ∧ NeedsGrouping(G2+, NewTree))
21    if S == R ∧ NeedsGrouping(G, NewTree)
22      NewTree = (ΓG(NewTree))
23    Trees.insert(NewTree)
24  return Trees
```

Figure 5.10: Pseudo code for OpTrees

routine terminates, the set *Trees* contains up to four different join trees, which are depicted in Figure 5.12.

5.5.2 A First Heuristic

Incorporating the routine OpTrees from the previous section into the plan builder of our plan generator enables the latte to not only build a pure join tree for a given ccp, but also all join trees for this pair with pushed-down groupings. According to the common DP-based plan generation scheme described in Chapter 3, we can then compare the costs of the different trees and insert the cheapest one into the DP table. Unfortunately, we are not guaranteed to find the optimal solution with this approach, because Bellman's Principle of Optimality does no longer hold once the placement of grouping operators is taken into account.

```
NeedsGrouping(G, T)
1  if G → A(T) ∧ the result of T is duplicate-free
2    return False
3  else
4    return True
```

Figure 5.11: Pseudo code for NeedsGrouping

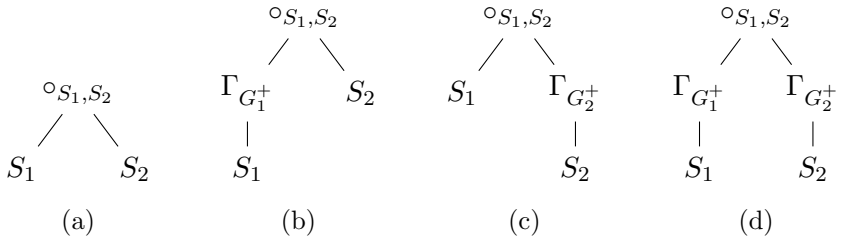


Figure 5.12: Operator trees for ccp $(S1, S2)$

Pushing a grouping operator into one or both arguments of a join operator can influence two properties of the respective subtree: the cardinality of the tree's result may be reduced and the functional dependencies and keys holding in the result may be altered. A reduced cardinality can reduce the cost of subsequent operations and thereby (more than) compensate the higher cost of a suboptimal tree. The functional dependencies, on the other hand, determine whether or not we need a final grouping on top to fix the query result. This final grouping causes an additional cost that can destroy the optimality of the plan. Consequently, we have to keep the more expensive subplans for each intermediate result because they might turn out to be a part of the optimal solution.

When optimizing the join order, plan classes are always defined through the result of the contained plans. That is, plans that are in the same class all produce the same result. This is no longer true in our case. Instead, we use the following equivalence relation for the definition of the plan classes: all plans in one class produce the same result if a grouping is added on top of each plan. The set of grouping attributes G^+ is unambiguously defined for each plan class. Following our definition of logical and physical plan properties, we thus consider the cardinality of a plan a physical property, since it can differ between plans belonging to the same class. The same is true for the key properties and functional dependencies holding in the result of a plan.

But there is still no need to calculate the cardinalities anew for every single plan. Since all plans belonging to one plan class produce the same result if they are grouped by G^+, the cardinality of this result can be calculated once for the corresponding plan class. This information can later be reused when joining any plan from the respective class with a grouping placed on top.

Figure 5.13 shows a modified version of the well-known routine BuildPlan

```
BuildPlanH1(S1, S2, ∘p)
1  for each T ∈ OpTrees(DPTable[S1], DPTable[S2], ∘p)
2    if Cost(T) < Cost(DPTable[S1 ∪ S2])
3      DPTable[S1 ∪ S2] = T
```

Figure 5.13: Pseudo code for BuildPlanH1

implementing the naive approach described above. We refer to the resulting plan generator as our first heuristic or H1. The modified routine is called BuildPlanH1. It serves to demonstrate the problems that arise from the violation of Bellman's Principle of Optimality. The only difference to the basic version of BuildPlan as presented in Chapter 3 is that the new routine calls OpTrees to find all possible trees for the current ccp. For each of them the cost function is called to compute the combined cost of the join and the groupings contained in the tree, if any. If the cost is lower than that of an existing plan or this is the first plan for the current set of relations, the plan is added to the DP table. In summary, H1 records only the single cheapest plan for every plan class.

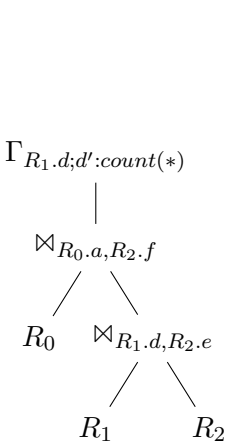
To clarify why this approach can lead to problems, Figure 5.14 provides an example. At the top of the figure there are two equivalent operator trees. Both of them involve a grouping operation. In the tree on the left there are no pushed-down groupings, so there is only one grouping at the top of the tree. In the tree on the right-hand side, a grouping operator has been pushed down into the left argument of $\bowtie_{R_1.d,R_2.e}$. Note how the aggregation vector of the original grouping operator at the top of the tree is adjusted according to our observations from Section 5.2. That is, we now have to sum up the values created by the other grouping operator to get the originally intended $count(*)$. Below the two operator trees there are instances of the three relations R_0, R_1 and R_2 as well as the intermediate results of both operator trees.

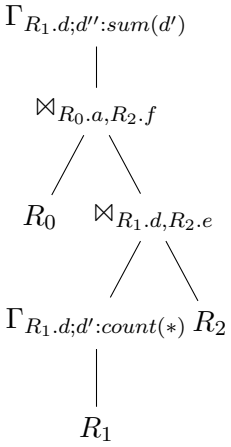
$C_{out}(R_0) = C_{out}(R_1) = C_{out}(R_2) = 0$	
$C_{out}(R_{1,2}) = 4$	$C_{out}(R'_1) = 3$
$C_{out}(R_{0,1,2}) = 8$	$C_{out}(R'_{1,2}) = 5$
$C_{out}(\Gamma(R_{0,1,2})) = 10$	$C_{out}(R'_{0,1,2}) = 7$
	$C_{out}(\Gamma(R'_{0,1,2}) = 9$

Table 5.1: Costs of intermediate results

Table 5.1 lists the costs of all subexpressions contained in both operator trees. For simplicity, we use a slightly extended version of the cost function C_{out} from Section 2.4 that is suitable for determining the cost of a grouping operation:

$$C_{out}(T) = \begin{cases} 0 & \text{if T is single relation} \\ |T| + C_{out}(T_1) + C_{out}(T_2) & \text{if } T = T_1 \circ T_2 \\ |T| + C_{out}(T_1) & \text{if } T = \Gamma(T_1) \end{cases}$$





R_0

a	b
0	0
1	0
2	1
3	1

R_1

c	d
0	1
1	0
2	1
3	1
4	4

R_2

e	f
0	0
1	1
2	3
3	4

$R_{1,2} := R_1 \bowtie_{R_1.a,R_2.e} R_2$

c	d	e	f
0	1	1	1
1	0	0	0
2	1	1	1
3	1	1	1

$R'_1 := \Gamma_{d;d':count(*)}(R_1)$

d	d'
0	1
1	3
4	1

$R_{0,1,2} := R_0 \bowtie_{R_0.a,R_2.f} R_{1,2}$

a	b	c	d	e	f
0	0	1	0	0	0
1	0	0	1	1	1
1	0	2	1	1	1
1	0	3	1	1	1

$R'_{1,2} := R'_1 \bowtie_{R_1.d,R_2.e} R_2$

d	d'	e	f
0	1	0	0
1	3	1	1

$\Gamma_{1.d;d':count(R_1.d)}(R_{0,1,2})$

d	d'
1	3
0	1

$R'_{0,1,2} := R_0 \bowtie_{R_0.a,R_2.f} R'_{1,2}$

a	b	d	d'	e	f
1	0	1	3	1	1
0	0	0	1	0	0

$\Gamma_{1.d;d'':sum(d')}(R'_{0,1,2})$

d	d''
1	3
0	1

Figure 5.14: Intermediate results of two equivalent operator trees

According to this cost function, scanning the base relations causes no cost at all, which is reflected in the first line of Table 5.1. Beginning in the second line, the left (right) column contains the cost of the intermediate results of the left (right) plan shown in Figure 5.14.

As per our heuristic, we decide against placing a grouping directly on top of relation R_1, because the combined cost of the grouping and the following join operation is higher than the cost of joining without prior grouping. Taking a closer look at the following lines in our table, we see that the cost of joining $R_{1,2}$ with R_0 amounts to 8, whereas the right column states a cost value of 7 for the join between $R'_{1,2}$ and R_0. For the total cost of the query, we notice the same difference: the total cost of the left tree amounts to 10, whereas the cost of the right one adds up to only 9. This means that in this case, our naive plan generator discards one tree in favor of a more expensive one.

The reason for this behavior is found in Figure 5.14. The early grouping of relation R_1 causes an additional cost of 3, but it also reduces the cardinalities of the following expressions $R'_{1,2}$ and $R'_{0,1,2}$ when compared to $R_{1,2}$ and $R_{0,1,2}$. The additional cost caused by the first grouping operation is therefore compensated by the reduced cardinalities and costs of the following expressions. Considering only the cost of expression $R'_{1,2}$, this is not obvious because it becomes visible only further up in the tree.

In the example above, the influence of an early grouping on the cardinalities of subsequent expressions is already enough to make eager aggregation beneficial. But there is also a second aspect to it that allows for even bigger cost savings: the introduction of new grouping operators also influences the functional dependencies holding in the intermediate results.

Looking back at the values for $R'_{0,1,2}$ in Figure 5.14, we can see that the final grouping is not necessary in order to produce the same result as the left join tree. Instead, a projection on the attribute set $\{R_1.d, d'\}$ suffices because the functional dependency $R_1.d \rightarrow \mathcal{A}(R'_{0,1,2})$ holds, i.e., $R_1.d$ is a key for $R'_{0,1,2}$ and the attribute d' already contains the correct value for the original aggregate function $count(*)$. We can therefore leave out the final grouping and replace it by a much cheaper duplicate-preserving projection $\Pi_{R_1.d,d'}$. Since our cost function does not take the projection cost into account, we end up with a cost value of 7 for the tree applying eager aggregation, in contrast to a value of 10 for the other tree.

These findings lead to the conclusion that it is not sufficient to "locally" assess the profitability of pushing down a grouping if an optimal plan is desired. Still, this approach can be used as a simple heuristic producing only a moderate overhead on top of join ordering and at the same time often producing better plans, as we will see in Section 5.8. Moreover, H1 never produces a plan that is worse than that without eager aggregation.

5.5.3 Improving the Heuristic

As we have seen in the example in the previous section, there are cases in which H1 discards a subplan that is required for the optimal overall plan. This is due to the conservative approach it follows by only pushing a grouping down if the

```
BuildPlanH2(S1, S2, ∘p)
1  for each T ∈ OpTrees(DPTable[S1], DPTable[S2], ∘p)
2    if CompareAdjustedCost(T, DPTable[S1 ∪ S2])
3      DPTable[S1 ∪ S2] = T

CompareAdjustedCost(T1, T2)
1  if T is top-level plan ∨ Eagerness(T1) == Eagerness(T2)
2    return Cost(T1) < Cost(T2)
3  if (Eagerness(T1) < Eagerness(T2))
4    return (F × Cost(T1)) < Cost(T2)
5  if (Eagerness(T1) > Eagerness(T2))
6    return Cost(T1) < (F × Cost(T2))
```

Figure 5.15: Pseudo code for BuildPlanH2 and CompareAdjustedCost

combined cost caused by the grouping and the following join is lower than the cost of the join alone. This way, the resulting plan can never be worse than the plan without eager aggregation.

One possible approach for improving H1 is to allow it to sometimes push a grouping even though the accumulated cost of the grouping and the subsequent join is higher than the join cost alone. Clearly, this bears the risk of making the resulting plan more expensive than the equivalent plan without any pushed-down groupings.

Inspired by the descriptive naming scheme established by Yan and Larson, we define the *eagerness* of a join tree T as the number of grouping operators that are direct children of the topmost join operator:

$$eagerness(T) = \begin{cases} 0 & \text{if } T = T_1 \circ T_2 \\ 1 & \text{if } T = \Gamma(T_1) \circ T_2 \text{ or } T = T_1 \circ \Gamma(T_2) \\ 2 & \text{if } T = \Gamma(T_1) \circ \Gamma(T_2) \end{cases}$$

Figure 5.15 shows the pseudo code for the routine BuildPlanH2, which favors more eager trees over less eager trees when deciding which tree to insert into the DP table.

The main difference to BuildPlanH1 is the new subroutine CompareAdjustedCost. It is called from Line 2. It takes two join trees and compares the costs of the two, whereby it adjusts the cost of the less eager tree using a constant factor F. The value of F determines the degree to which more eager plans are preferred when compared to less eager plans. If the eagerness of the two join trees passed to CompareAdjustedCost is equal, or the trees form a plan for the whole query, no cost adjustment will be made. In Section 5.8 we experiment with different values for F.

```
BuildAllPlans(S1, S2, ∘p)
1  S = S1 ∪ S2
2  for each T1 ∈ DPTable[S1]
3    for each T2 ∈ DPTable[S2]
4      for each T ∈ OpTrees(T1, T2, ∘p)
5        if S == R
6          InsertTopLevelPlan(S, T)
7        else
8          DPTable[S1 ∪ S2].Insert(T)

InsertTopLevelPlan(S, T)
1  if DPTable[S] == ∅ ∨ Cost(T) < Cost(DPTable[S])
2    DPTable[S] = {T}
```

Figure 5.16: Pseudo code for BuildAllPlans

5.5.4 Finding an Optimal Solution

According to the observations made in Section 5.5.2, we will have to keep all plans of a certain class enumerated by our plan generator in the DP table if the goal is to find the best possible query plan.

Figure 5.16 shows the routine BuildAllPlans, which resembles the well-known BuildPlan (see Figure 3.2). In contrast to the latter, it assumes a DP table storing a set of plans for each plan class. In Section 2.4 we proposed to store plans of the same plan class in the form of a linked list with pointers from each plan to the next one. This is a convenient way of implementing a DP table with more than one plan per plan class. We do not really need set semantics because no duplicate plans are enumerated and the ordering implied by the linked list is simply ignored.

As before, we enumerate all pairs of subsets (S_1, S_2) with $S = S_1 \cup S_2$ to find possible join trees for S. We then combine every tree for S_1 with every tree for S_2 using two loops. OpTrees is called for each pair of join trees, which results in up to four different trees for every combination. The newly created trees are added to the set for S.

Eventually, we face the situation where $S = R$ holds and we need to build a tree containing all input relations. At this point another subroutine named InsertTopLevelPlan is called. Inside this routine the join trees for S are compared to find the one with minimal cost, because there are no subsequent join operators that need to be taken into account. Before this, we have to decide whether or not a top-level grouping is needed by calling NeedsGrouping, which is shown in Figure 5.11. In contrast to the other relation sets, no set of trees has to be stored for R, but only the best tree found so far.

In summary, this algorithm enumerates and stores all possible plans for the input query, except for those in which a grouping would group by a key and

```
PruneDominatedPlans(S, T)
1  for T_old ∈ DPTable[S]
2    if T_old dominates T
3      return
4    if T dominates T_old
5      DPTable[S].Remove(T_old)
6  DPTable[S].Insert(T)
```

Figure 5.17: Pseudo code for PruneDominatedPlans

finally chooses the cheapest one among them. Clearly, this inefficient approach defeats the purpose of dynamic programming by negating its single largest advantage, namely that it allows for dividing a problem into smaller subproblems to solve them independently.

5.5.5 Pruning

Keeping all possible trees in the solution table guarantees an optimal solution, but causes such a big overhead that it is impractical for most queries. This leads us to the question whether we can find a way to reduce the number of DP table entries while preserving the optimality of the resulting solution. In other words, we are looking for an optimality-preserving pruning criterion.

To this end, we introduce the notion of *dominance*. Intuitively, if a tree is dominated by another tree, it will definitely not be contained in the optimal solution and can be discarded. The dominating tree, on the other hand, may be contained in the optimal solution, so we have to keep it.

Figure 5.17 shows the routine PruneDominatedPlans, which discards all trees that are dominated by another tree already stored in the respective tree set. The routine expects as arguments a set of relations S and a join tree T for this set. It is called from inside BuildAllPlans. To this end, we replace line 8 in BuildAllPlans with the following:

```
8   PruneDominatedPlans(S, T)
```

The loop in PruneDominatedPlans runs through the existing join trees for S taken from the DP table and compares each of them with the new tree T. If there is an existing tree T_{old}, which dominates the new tree T, then the latter is discarded. Therefore, the routine returns without adding T to the tree set for S.

If T dominates an existing tree T_{old}, we can safely delete the latter from the DP table. In this case, we continue to loop through the remaining trees, because more dominated trees may exist. Eventually, the loop ends and T is added to the set for S.

As long as Bellman's Principle of Optimality holds, dominance can be defined solely based on plan cost, as it was implicitly done in all the basic plan

```
GroupjoinTrees(S, Trees)
 1  GroupjoinTrees = ∅
    // Final grouping Γ_G is optional:
 2  for all T = (Γ_G(T_1 ∘_p T_2)) in Trees
 3    if S == R ∧ IsGrouped(T) ∧ GroupjoinApplicable(T)
 4      GroupjoinTrees.Insert(σ_{p_s}(T_1 ⋈_p T_2))
 5    if IsGrouped(T_1) ∧ GroupjoinApplicable(T_1)
 6      GroupjoinTrees.Insert((σ_{p_s1}(T_{1,1} ⋈_{p_1} T_{1,2})) ∘_p T_2)
 7      if S == R ∧ IsGrouped(T)
            ∧ GroupjoinApplicable((σ_{p_s1}(T_{1,1} ⋈_{p_1} T_{1,2})) ∘_p T_2)
 8        GroupjoinTrees.Insert(σ_{p_s}((σ_{p_s1}(T_{1,1} ⋈_{p_1} T_{1,2})) ⋈_p T_2))
 9    if IsGrouped(T_2) ∧ GroupjoinApplicable(T_2)
10      ... // Build tree with groupjoin in the right subtree
11    if IsGrouped(T_1) ∧ IsGrouped(T_2) ∧ GroupjoinApplicable(T_1)
        ∧ GroupjoinApplicable(T_2)
12      ... // Build tree with groupjoins in both subtrees
13  return Trees ∪ GroupjoinTrees
```

Figure 5.18: Pseudo code for GroupjoinTrees

generators described in Chapter 3. Clearly, defining dominance on the basis of only one plan property ensures that only one plan per plan class is stored in the DP table. That is because in this case it is impossible for a plan to dominate a plan in one aspect and be dominated in another. In Section 5.7 we discuss several possible notions of dominance based on different plan properties.

5.5.6 Introducing Groupjoins

The equivalences from Section 5.4 allow us to replace a sequence of grouping and join by a single groupjoin. Implementing this in a plan generator only requires changes to the subroutine OpTrees. Since the latter already produces all join trees with a grouping on top of one or both of the join arguments, we can simply extend it in such a way that it also produces all trees where a single groupjoin is applied instead. To this end, we introduce the new subroutine GroupjoinTrees shown in Figure 5.18 [9]. It is called in the last line of OpTrees and the resulting tree set is returned:

```
24  return GroupjoinTrees(S, Trees)
```

Clearly, the two subroutines could be combined into one.

Since GroupjoinTrees works by transforming join trees with pushed-down grouping, we pass as argument the set of join trees produced by OpTrees, which at this point contains up to four different join trees for S. The pseudo code for GroupjoinTrees is given in Figure 5.18. The four trees emitted by OpTrees are shown on the left-hand side of Figure 5.12. For each tree

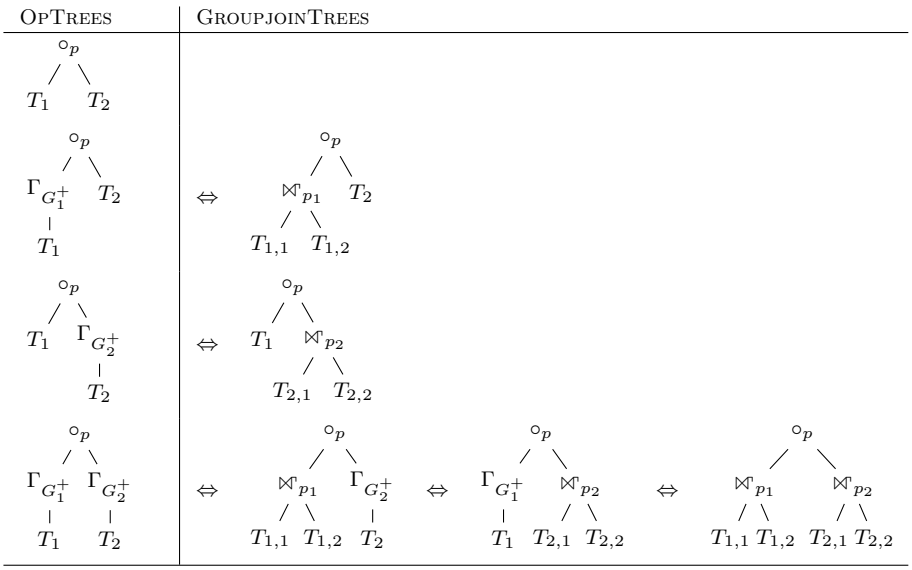


Figure 5.19: Trees enumerated by OpTrees and GroupjoinTrees

contained in the set, we consider introducing a groupjoin instead of a sequence of (left outer-) join and grouping. The existing trees can be top-level trees joining subtrees T_1 and T_2, possibly with a final grouping on top, or lower-level trees consisting only of a join between T_1 and T_2. For each tree we check if the left, right or both arguments of the join are grouped, i.e., if a grouping has been pushed down through the join. If this is the case, we check whether we can replace the join followed by a grouping by a groupjoin. That is, we test the requirements for Equivalences 5.36/5.38 or 5.37/5.39, depending on whether we have full information about functional dependencies or only keys available. This is achieved by a call to IsGroupjoinApplicable. If the routine returns *true*, we add the resulting tree to the set *GroupjoinTrees*.

In the pseudo code we include all selections that may be necessary according to the equivalences for the groupjoin. We refer to the left/right subtree of T_1 by $T_{1,1}/T_{1,2}$, respectively. T_2's subtrees are named accordingly. For each newly produced groupjoin tree, we also have to check if it is a top-level tree with a grouping on top. If this is the case, we might be able to replace the final join and grouping by a groupjoin. Again, we have to check the requirements before doing so. Finally, we return the union of *Trees* and *GroupjoinTrees*, i.e., the set of all possible trees including the ones with groupjoins.

The right-hand side of Figure 5.19 shows the five additional groupjoin trees that can be derived from the three original operator trees with pushed-down grouping operators. Together with the pure join tree without eager aggregation, we end up with a total number of nine possible operator trees for joining T_1 and T_2. We omit possibly necessary selection operators in the figure. The figure also does not show the special case where T_1 and T_2 contain all relations contained in the query. In that case, there may be even more trees in the set returned by GroupjoinTrees, because a grouping on top of the join may be necessary for some or all of the depicted trees. We may then be able to apply a top-level

groupjoin (see Figure 5.18).

The addition of GroupjoinTrees is the only modification necessary to enable the introduction of groupjoins in all plan generators discussed in this chapter. All other parts of the plan generator remain untouched. In Section 5.8 we will see that the runtime overhead caused by this extension is negligible. On the other hand, the introduction of groupjoins can significantly increase the quality of the resulting plans.

5.5.7 Summary

To summarize this section, Figure 5.20 provides an overview of the possible configurations of our plan generator. Every path in the tree leads to a valid plan generator with certain capabilities, such as reordering grouping and join (marked by Γ), introduction of groupjoins (marked by ⋈Γ) and dominance-pruning. For example, if we wish to implement a plan generator that is capable of reordering join and grouping and then apply some sort of optimality-preserving pruning without introducing groupjoins, we have to plug BuildAllPlans into the enumerator. The former always calls OpTrees to obtain operator trees with pushed-down groupings, but in our case we omit the call to GroupjoinTrees. For the trees returned by OpTrees we then call PruneDominatedPlans.

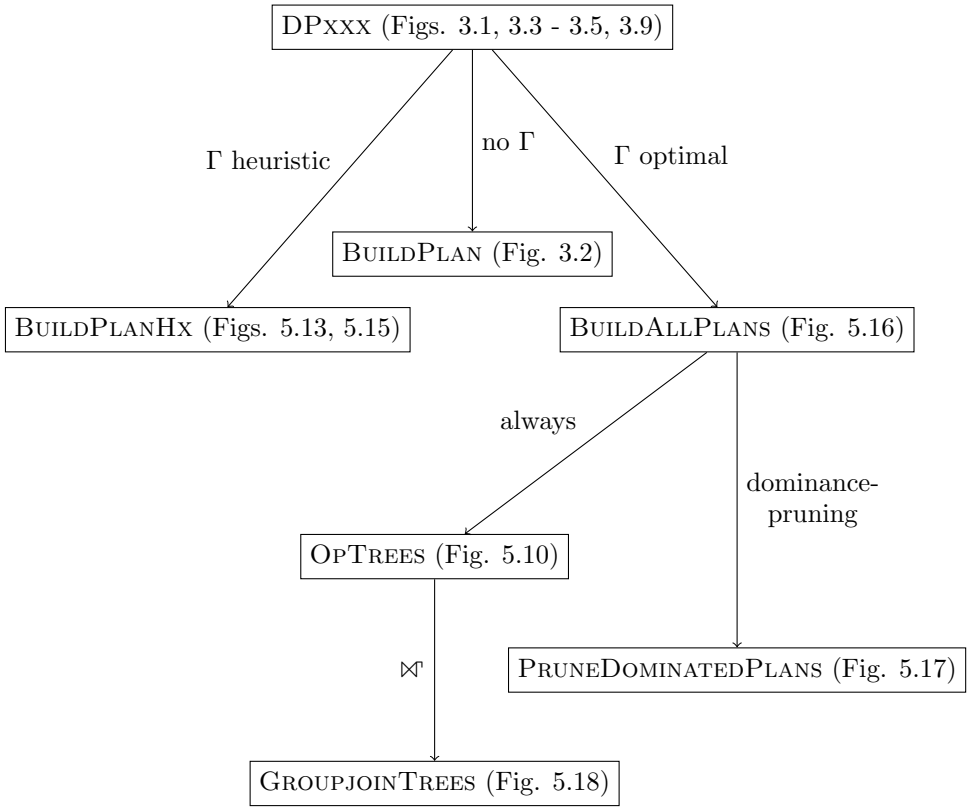


Figure 5.20: Configuration options for the plan generator

5.6 Interesting Plan Properties and their Derivation

This section provides rules for computing properties of query plans that we use to determine dominance. These properties need to be maintained for every plan, which can be done in the form of a data structure *properties* that is associated with every plan, as proposed in Figure 3.10. The following rules have first been presented in [8].

5.6.1 Interesting Properties

Keys: We denote by $\kappa(e)$ the set of keys for a relation defined by an expression e. Note that a single key is a set of attributes. Therefore, κ is a set of sets. Subsequently, we will use the term key for what is actually a superkey and only distinguish the two where it matters. The keys resulting from the full and left outerjoin contain null values. We therefore assume that null values are treated as suggested by Paulley, i.e., two attributes are equal if they agree in value or they are both null [35]. This reflects the semantics of SQL's *group by*. We assume that we know the keys of the base relations from the database schema.

Functional Dependencies: We denote by $FD(e)$ the set of functional dependencies (FDs) holding in expression e. Again, we adopt Paulley's definition of functional dependency, where two attributes with value null are treated as equal [35]. Initially, FDs for a base relation are deduced from the keys declared in the database schema. Hereinafter, we frequently use the closure of a given set of FDs, denoted by FD^+. By closure we mean the set of all dependencies derivable from a given set of dependencies, as the term is commonly understood.

Equality Constraints: We denote by $EC(e)$ the set of equality constraints holding in expression e. Equality constraints are captured in equivalence classes. An equivalence class is a set of attributes $\{a_1, a_2, \ldots, a_n\}$ where the attributes a_1 through a_n are known to have equal values. Note that this definition makes EC a set of sets. Below, we define a set of operations for accessing and modifying a given set of equality constraints.

We denote by $EC[a]$ the equivalence class containing attribute a:

$$EC[a] = \{c | c \in EC, a \in c\}.$$

We denote by $EC \leftarrow (a = b)$ the insertion of the equality constraint $a = b$ into EC, with a and b being two attributes:

$$EC \leftarrow (a = b) \equiv EC \setminus \{EC[a], EC[b]\} \cup \{EC[a] \cup EC[b]\}.$$

Initially, EC contains a singleton for each available attribute a_1 to a_n across relations:

$$EC = \{\{a_1\}, \{a_2\}, \ldots, \{a_n\}\}.$$

Definite Attributes: We denote by $NN(e)$ the set of definite attributes in an expression e. Definite attributes are attributes that do not contain the value null. If e is a base relation, $NN(e)$ contains the attributes that are declared as "not null" in the database schema.

5.6.2 Deriving Interesting Properties

We provide rules for computing the four sets bottom-up in an operator tree possibly containing all algebraic operators contained in LOP, as defined in Section 2.2. The rules concerning EC and FD are taken from Paulley [35]. For simplicity, we make some restrictions on the join predicates we consider. We assume (possibly) conjunctive predicates with each conjunct referencing exactly two relations. We do not claim that the presented rules are complete. A bigger set of rules may result in bigger property sets and thereby in more pruning opportunities. On the other hand, evaluating more rules leads to a higher overhead for computing the property sets. The rules presented here merely serve as examples and can easily be extended if necessary, for example to take additional operators into account.

Inner Join

Consider the join of two expressions e_1 and e_2 with join predicate p: $e_1 \bowtie_p e_2$.

Keys: We have to distinguish three cases [7]:

- In case $\{a_1\}$ is a key of e_1 and $\{a_2\}$ is a key of e_2, we have

  $$\kappa(e_1 \bowtie_{a_1=a_2} e_2) = \kappa(e_1) \cup \kappa(e_2).$$

  That is, each key from one of the input expressions is again a key for the join result.

- In case $\{a_1\}$ is a key but $\{a_2\}$ is not, we have

  $$\kappa(e_1 \bowtie_{a_1=a_2} e_2) = \kappa(e_2).$$

  The case where $\{a_2\}$ is a key and $\{a_1\}$ is not is handled analogously.

- Without any assumption on the a_i or the join predicate, we have

  $$\kappa(e_1 \bowtie_p e_2) = \bigcup_{K_1 \in \kappa(e_1), K_2 \in \kappa(e_2)} K_1 \cup K_2.$$

  In other words, every pair of keys from e_1 and e_2 forms a key for the join result.

Functional Dependencies: In the join result all FDs from the two input expressions still hold, resulting in the following equation:

$$FD(e_1 \bowtie_p e_2) = FD^+(e_1) \cup \ FD^+(e_2).$$

Equality Constraints: If p is an equality predicate of the form $a_1 = a_2$, with a_1 belonging to e_1 and a_2 belonging to e_2, we know that after the join a_1 and a_2 are equal.

We capture this information by defining an equivalence class containing the two attributes. The existing equality constraints holding in the join arguments remain valid after the join, i.e., the following equation holds for an equijoin:

$$EC(e_1 \bowtie_{a_1=a_2} e_2) = (EC(e_1) \cup EC(e_2)) \leftarrow (a_1 = a_2).$$

For all predicates other than equality conditions, we can state the following equation:

$$EC(e_1 \bowtie_p e_2) = EC(e_1) \cup EC(e_2).$$

Definite Attributes: All attributes that are known to be definite in the join arguments still have this property after the join. Additionally, all attributes that p rejects nulls on are definite after the join:

$$NN(e_1 \bowtie_p e_2) = NN(e_1) \cup NN(e_2) \cup NR(p).$$

Left Outerjoin

Consider the left outerjoin of expressions e_1 and e_2: $e_1 \mathbin{⟕}_p e_2$. Since the left outerjoin can introduce null values, we have to be careful when determining the dependencies and constraints holding in its result.

Keys: Here, we have only two possible cases. If $\{a_2\}$ is a key of e_2, then

$$\kappa(e_1 \mathbin{⟕}_{a_1=a_2} e_2) = \kappa(e_1).$$

Otherwise, we have to combine two keys from e_1 and e_2 to form a key:

$$\kappa(e_1 \mathbin{⟕}_p e_2) = \bigcup_{K_1 \in \kappa(e_1), K_2 \in \kappa(e_2)} K_1 \cup K_2,$$

where p is an arbitrary predicate.

Functional Dependencies: All FDs holding in e_1, the preserved side of the outerjoin, continue to hold in the join result. Dependencies from e_2, the null-supplying side of the outerjoin, only continue to hold if the left-hand side of the dependency contains an attribute that p rejects nulls on or a definite attribute. This gives rise to the following equation, where p is an arbitrary predicate:

$$\begin{aligned} FD(e_1 \mathbin{⟕}_p e_2) = {} & FD^+(e_1) \\ & \cup \{(\alpha \to \beta) \in FD^+(e_2) \mid (\alpha \cap (NN(e_2) \cup NR(p)) \neq \emptyset)\}. \end{aligned}$$

In the case of an equality predicate, we do not get a new equivalence class, as it was the case for the inner join. Instead, we get a new FD with the join attribute from the preserved join argument on the left-hand side and the one from the null-supplying argument on the right-hand side. Consider the following left

outerjoin of expressions e_1 and e_2, where a_1 belongs to e_1 and a_2 belongs to e_2: $e_1 \mathbin{⟕}_{a_1=a_2} e_2$. In this case the following equation holds:

$$\begin{aligned} FD(e_1 \mathbin{⟕}_{a_1=a_2} e_2) = {} & FD^+(e_1) \\ & \cup \{(\alpha \to \beta) \in FD^+(e_2) \mid (\alpha \cap (NN(e_2) \cup NR(p)) \neq \emptyset)\} \\ & \cup \{a_1 \to a_2\}. \end{aligned}$$

Equality Constraints: Equality constraints from both join arguments continue to hold in the join result, resulting in the following equation:

$$EC(e_1 \mathbin{⟕}_p e_2) = EC(e_1) \cup EC(e_2).$$

Definite Attributes: Since the left outerjoin can introduce null values in all attributes from the null-supplying relation (e_2 in our case), no attribute from e_2 is definite in the join result. The only definite attributes remaining are the ones from e_1, the preserved relation:

$$NN(e_1 \mathbin{⟕}_p e_2) = NN(e_1).$$

Full Outerjoin

Consider the full outerjoin of expressions e_1 and e_2: $e_1 \mathbin{⟗}_p e_2$.

Keys: Regardless of the join predicate, we have to combine two keys from e_1 and e_2 to form a key for the join expression:

$$\kappa(e_1 \mathbin{⟗}_p e_2) = \bigcup_{K_1 \in \kappa(e_1), K_2 \in \kappa(e_2)} K_1 \cup K_2,$$

where p is an arbitrary join predicate.

Functional Dependencies: Since in the full outerjoin both input relations are null-supplying, we have to apply the same rules to both join arguments that we used for the null-supplying argument of the left outerjoin. In other words, FDs from either e_1 or e_2 only continue to hold if the left-hand side of the dependency contains an attribute p rejects nulls on, or a definite attribute:

$$\begin{aligned} FD(e_1 \mathbin{⟗}_{a_1=a_2} e_2) = {} & \{(\alpha \to \beta) \in FD^+(e_1) \mid \\ & (\alpha \ \cap \ NN(e_1) \neq \emptyset) \vee (p \text{ is null-rejecting in } \mathcal{F}(p) \cap \mathcal{A}(e_1))\} \\ & \cup \ \{(\alpha \to \beta) \in FD^+(e_2) \mid (\alpha \cap NN(e_2) \neq \emptyset) \\ & \qquad \vee (p \text{ is null-rejecting in } \mathcal{F}(p) \cap \mathcal{A}(e_2))\}. \end{aligned}$$

Equality Constraints: As was the case for the left outerjoin, equality constraints from both join arguments remain valid in the result of a full outerjoin:

$$EC(e_1 \mathbin{⟗}_p e_2) = EC(e_1) \cup EC(e_2).$$

Definite Attributes: The full outerjoin can introduce null values in all attributes contained in the join result. This means that there are no definite attributes after the join:

$$NN(e_1 \mathbin{⟗}_p e_2) = \emptyset.$$

Left Semijoin/Left Antijoin/Left Groupjoin

Consider a left semijoin $(e_1 \ltimes_p e_2)$, left antijoin $(e_1 \triangleright_p e_2)$ or left groupjoin $(e_1 \bowtie\!\!\!\Gamma_p e_2)$ of expressions e_1 and e_2. According to our definitions from Section 2.2, none of these operators add new tuples to their result and none of them return tuples from the right argument. Therefore, the properties from the left argument generally remain valid in the join result and those from the right argument do not. Some exceptions occur in the case of the groupjoin.

Keys:

$$\kappa(e_1 \circ e_2) = \kappa(e_1), \text{ for } \circ \in \{\ltimes, \triangleright, \bowtie\!\!\!\Gamma_{G;A:F}\}.$$

Functional Dependencies:

$$FD(e_1 \circ_p e_2) = FD^+(e_1), \text{ for } \circ \in \{\ltimes, \triangleright\}.$$

In the left groupjoin, the attributes in G determine the ones in A:

$$FD(e_1 \bowtie\!\!\!\Gamma_{p;G;A:F} e_2) = FD^+(e_1) \cup \{G \rightarrow A\}.$$

Equality Constraints:

$$EC(e_1 \circ_p e_2) = EC(e_1), \text{ for } \circ \in \{\ltimes, \triangleright, \bowtie\!\!\!\Gamma\}.$$

Definite Attributes:

$$NN(e_1 \circ_p e_2) = NN(e_1), \text{ for } \circ \in \{\ltimes, \triangleright\}.$$

In the left groupjoin, an attribute $a \in A$ is definite if the aggregate function it results from does not return null. This depends on whether or not the argument of the aggregate function is definite and on the characteristics of the aggregate function. For example, *count(*)* never returns null, whereas min returns null if all input values are null.

If the former is the case for all $f \in F$, we can state the following equation:

$$NN(e_1 \bowtie\!\!\!\Gamma_{p;G;A:F} e_2) = NN(e_1) \cup A.$$

Grouping

The result of a grouping applied to an expression e consists of the attribute set A containing the aggregation results and those attributes from e that are contained in the grouping attributes G.

Keys: We assume a grouping applied to expression e: $\Gamma_{G;A:F}(e)$. The grouping attributes G can be a key of the grouping's argument e. In this case, all keys contained in G remain keys after applying the grouping:

$$\kappa(\Gamma_{G;A:F}(e)) = \{K \in \kappa(e) | K \subset G\}.$$

Otherwise, the only key of the resulting relation consists of the grouping attributes G:

$$\kappa(\Gamma_{G;A:F}(e)) = \{G\}.$$

Functional Dependencies: In the result of the grouping all FDs referring only to the grouping attributes or to a subset thereof remain valid. That is, we keep those dependencies where both sides are contained in the grouping attributes. Additionally, the grouping attributes determine the aggregation attributes:

$$FD(\Gamma_{G;A:F}(e)) = \{f : \alpha \to \beta \mid f \in FD^+(e) \wedge \alpha, \beta \subseteq G\} \cup \{G \to A\}.$$

Equality Constraints: Equality constraints referring only to the grouping attributes or a subset thereof still hold in the result of a grouping:

$$EC(\Gamma_G(e)) = \{c \cap G \mid c \in EC(e), c \cap G \neq \emptyset\}.$$

Definite Attributes: A grouping does not introduce new null values. The aggregation results in attribute set A may be definite under the same conditions as for the groupjoin. If this is the case for all attributes in A, the following holds:

$$NN(\Gamma_{G;A:F}(e)) = (NN(e) \cap G) \cup A.$$

5.6.3 Computing the Attribute Closure

During plan generation we need to compute the *attribute closure* of a set of attributes α, which we denote by $AC(\alpha)$. Since in the case of equijoins we do not store any FDs between the join attributes, but instead put them in an equivalence class, we have to make use of the equivalence classes to compute the attribute closure. For each FD $\alpha \to \beta$, we add all attributes to the result set that are in the same equivalence class as some attribute $B \in \beta$. Next, we have to go through the existing FDs and see if there is a dependency $\beta' \to \gamma$ with $\beta' \subseteq$ *result*, which gives the transitive dependency $\alpha \to \gamma$. In this case, we add γ to the result and repeat the whole process until there are no more changes.

The pseudo code for AttributeClosure is given in Figure 5.21. As arguments, the procedure expects the set of functional dependencies *FD*, the set of equivalence classes *EC* and the attribute set α for which the attribute closure is computed.

```
AttributeClosure(FD, EC, α)
 1  result = α
 2  repeat
 3    hasChanged = false
 4    for all e ∈ EC
 5      if (e ∩ result) ≠ ∅
 6        result = result ∪ e
 7    for all FDs β → γ in FD
 8      if β ⊆ result
 9        result = result ∪ γ
10        hasChanged = true
11  until hasChanged = false
12  return result
```

Figure 5.21: Pseudo code for AttributeClosure

5.6.4 Implementation Details

Computing and storing the aforementioned plan properties during plan generation causes some overhead, which can be mitigated by carefully choosing the data structures and algorithms used to represent and compute them. In our implementation we use bitvectors for all attribute sets, such as *NN* and equivalence classes in *EC*, making frequently needed set operations, such as inclusion tests, very fast. *EC* itself can be stored in a union-find data structure [4]. It is optimized for a fast lookup of equivalence classes with a single array access. This way, inserting new equivalence classes becomes more expensive, but we only need to compute equality constraints once for every plan class, whereas the lookup needs to be done much more often, namely whenever two plans are compared.

We also store in each plan the attribute closure for each attribute occurring on the left-hand side of some dependency. This way, we only need to update the closure when it changes instead of computing it from scratch, which can be done with a single iteration of the algorithm in Figure 5.21.

5.7 Optimality-Preserving Pruning

In Section 5.5.5, the concept of dominance was introduced without any specific information on how dominance can be determined. In this section five concrete notions of dominance are presented. All of them are based on the plan properties described in the previous section. They have been published in [7, 8].

5.7.1 Pruning with FDs

First, we define *f-dominance* [7]:

Definition 11. *A join tree T_1 f-dominates another join tree T_2 for the same set of relations if all of the following conditions hold:*

1. $Cost(T_1) \leq Cost(T_2)$
2. $|T_1| \leq |T_2|$
3. $FD^+(T_1) \supseteq FD^+(T_2)$.

We denote by $|T|$ the cardinality of operator tree T's result. It is important to note that the compared trees do not necessarily produce the same result due to the contained grouping operators. As discussed in Section 5.3, a grouping on top of the final join may be necessary to compensate this.

Theorem 3. *Let T_2 be any operator tree containing a subtree T_2^{sub}. Further, let T_1^{sub} be a tree f-dominating T_2^{sub}. Then, the following holds:*

$$\exists T_1 : T_1 \equiv T_2 \wedge Cost(T_1) \leq Cost(T_2),$$

where T_1^{sub} is a subtree of T_1.

By $T_1 \equiv T_2$ we mean the equivalence of T_1 and T_2 with respect to their result when evaluated as an algebraic expression.

Proof. Definition 11 implies that $\mathcal{T}(T_1^{sub}) = \mathcal{T}(T_2^{sub})$. In this case, we can replace T_2^{sub} in T_2 by T_1^{sub} and the resulting tree T_1 cannot be more expensive than T_2 due to the monotonicity of the cost function and the second condition of Definition 11. However, the two trees may not be equivalent, i.e., one or both of them may require an additional grouping on top to achieve the correct query result.

If T_2 requires an additional grouping, the monotonicity of the cost function guarantees that $Cost(T_1) \leq Cost(T_2)$ holds, regardless of whether or not T_1 requires an additional grouping as well. Assume that only T_1 requires an additional grouping, leading to $Cost(T_2) < Cost(T_1)$. Thus, the following must hold:

$$\exists f_2 : G \rightarrow \mathcal{A}(T_2) \in FD^+(T_2) \wedge \nexists f_1 : G \rightarrow \mathcal{A}(T_1) \in FD^+(T_1).$$

Let $O(T)$ be the set of operators contained in a tree T. Since $FD^+(T_1^{sub}) \supseteq FD^+(T_2^{sub})$ holds, the functional dependency f_2 must arise in one of the operators contained in $O(T_2) \setminus O(T_2^{sub})$. But since both trees are identical above T_1^{sub}/T_2^{sub}, respectively, the same functional dependency must arise in T_1. □

In order to avoid the overhead associated with computing FD^+, which is needed for f-dominance, one can use the key set κ to define a different form of dominance called *k-dominance* [7]:

Definition 12. *A join tree T_1 k-dominates another join tree T_2 for the same set of relations if all of the following conditions hold:*

1. $Cost(T_1) \leq Cost(T_2)$

2. $|T_1| \leq |T_2|$

3. $\kappa(T_1) \supseteq \kappa(T_2)$.

Theorem 4. *Let T_2 be any operator tree containing a subtree T_2^{sub}. Further, let T_1^{sub} be a tree k-dominating T_2^{sub}. Then, the following holds:*

$$\exists T_1 : T_1 \equiv T_2 \wedge \mathit{Cost}(T_1) \leq \mathit{Cost}(T_2),$$

where T_1^{sub} is a subtree of T_1.

Proof. Definition 12 implies that $\mathcal{T}(T_1^{sub}) = \mathcal{T}(T_2^{sub})$.

In this case, we can replace T_2^{sub} in T_2 by T_1^{sub}, and the resulting tree T_1 cannot be more expensive than T_2 due to the monotonicity of the cost function and the second condition of Definition 12. However, the two trees may not be equivalent, i.e., one or both of them may require an additional grouping on top to achieve the correct query result.

If T_2 requires an additional grouping, the monotonicity of the cost function guarantees that $\mathit{Cost}(T_1) \leq \mathit{Cost}(T_2)$ holds, regardless of whether or not T_1 requires an additional grouping as well. Assume that only T_1 requires an additional grouping, leading to $\mathit{Cost}(T_2) < \mathit{Cost}(T_1)$. Thus, the following must hold:

$$\exists k_2 \in \kappa(T_2) : G \subseteq k_2 \wedge \nexists k_1 :\in \kappa(T_1) : G \subseteq k_1.$$

Since $\kappa(T_1^{sub}) \supseteq \kappa(T_2^{sub})$ holds, the key k_2 must arise in one of the operators contained in $O(T_2) \backslash O(T_2^{sub})$. But since both trees are identical above T_1^{sub}/T_2^{sub}, respectively, the same key must arise in T_1. □

With the following example, we show that there are cases where one tree f-dominates another tree, but does not k-dominate it. In such cases it can be beneficial to use FDs instead of keys for the pruning. Figure 5.22 shows two operator trees for the same query on relations $R_0, \ldots, R_3$. We assume that each relation R_i has two attributes: one key attribute k_i and one non-key attribute n_i, with $i \in (0, \ldots, 3)$. In addition to the operators, the trees shown in Figure 5.22 contain special nodes displaying the keys valid at the respective point in the tree according to the key computation rules from Section 5.6. We assign numbers to the operators to make them easier to identify.

Assume that during plan generation we compare the subtrees for the relation set $\{R_0, R_1, R_2\}$ to decide if one of them can be discarded. To this end, we have to check if one of the trees dominates the other according to our definition of k-dominance (Def. 12). Assume further that the tree on the right has lower cost than the one on the left and equal cardinality. Therefore, the only condition for k-dominance remaining to be checked is the third one, i.e, we have to check if $\kappa(\bowtie^{2a}) \subseteq \kappa(\bowtie^{2b})$ holds. Here and in the following examples we write $\kappa(\circ)/FD^+(\circ)$ instead of $\kappa(T)/FD^+(T)$, respectively, where $\circ$ is the operator at the root of T. Obviously, the requirement from above is not met and we decide to keep the more expensive subtree. We will now use f-dominance as the pruning criterion.

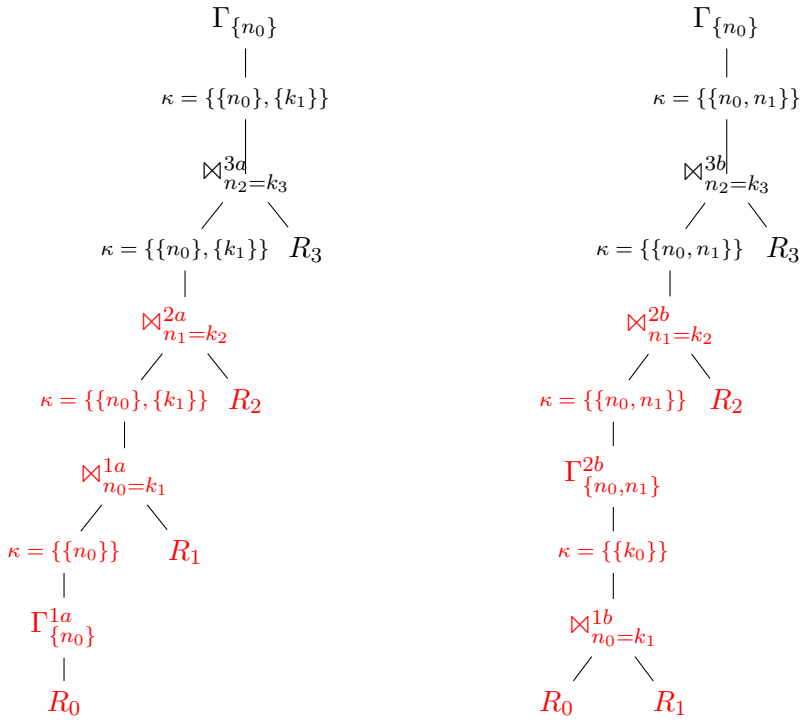


Figure 5.22: Two operator trees with keys

Table 5.2: Functional dependencies for Figure 5.22

	Figure 5.22a		**Figure 5.22b**	
	AC$^+$	**EC**	**AC$^+$**	**EC**
R_0	$\{k_0\} \to \{k_0, n_0\}$	$\emptyset$	$\{k_0\} \to \{k_0, n_0\}$	$\emptyset$
R_1	$\{k_1\} \to \{k_1, n_1\}$	$\emptyset$	$\{k_1\} \to \{k_1, n_1\}$	$\emptyset$
R_2	$\{k_2\} \to \{k_2, n_2\}$	$\emptyset$	$\{k_2\} \to \{k_2, n_2\}$	$\emptyset$
R_3	$\{k_3\} \to \{k_3, n_3\}$	$\emptyset$	$\{k_3\} \to \{k_3, n_3\}$	$\emptyset$
Γ^1	$\{n_0\} \to \{n_0\}$	$\emptyset$	-	$\emptyset$
$\bowtie^1$	$\{\{n_0, k_1\}\} \to \{\{n_0, k_1\}, n_1\}$	$\{n_0, k_1\}$	$\{\{n_0, k_1\}\} \to \{\{n_0, k_1\}, n_1\}$	$\{n_0, k_1\}$
Γ^2	-	$\{n_0, k_1\}$	$\{n_0, n_1\} \to \{n_0, n_1\}$ $\{n_0\} \to \{n_0, n_1\}$	$\emptyset$
$\bowtie^2$	$\{\{n_0, k_1\}\} \to \{\{n_0, k_1\}, \{n_1, k_2\}, n_2\}$ $\{\{n_1, k_2\}\} \to \{\{n_1, k_2\}, n_2\}$	$\{n_0, k_1\}$ $\{n_1, k_2\}$	$\{n_0\} \to \{n_0, \{n_1, k_2\}, n_2\}$ $\{n_0, n_1\} \to \{n_0, \{n_1, k_2\}, n_2\}$ $\{\{n_1, k_2\}\} \to \{\{n_1, k_2\}, n_2\}$	$\{n_1, k_2\}$

Table 5.2 shows the FDs and equivalence classes for each intermediate result of the join trees depicted in Figure 5.22. For each operator, the table gives the set of non-empty attribute closures AC^+ holding in the operator's result, computed according to the algorithm described in Section 5.6. We use AC^+ instead of FD^+, since the former is much smaller and provides all the information needed for our purposes.

For base relations, the only dependencies we have are given by the key constraints from the relations' schemas. Once the grouping on top of R_0 is applied in Figure 5.22a, we lose the key constraint of R_0 because the key is not part of

the grouping attributes. Instead, we get a new dependency from the grouping attribute n_0 to all other attributes in the result, namely the grouping attributes and the attributes containing the aggregation results. We omit the latter because they are of no importance for our observations.

The evaluation of the first join predicate results in an equivalence class containing the join attributes n_0 and k_1. Since the two attributes are equivalent, we can replace one by the other in all our FDs. We denote this by replacing all occurrences of an attribute by its equivalence class. This way, the FD $\{\{n_0, k_1\}\} \rightarrow \{\{n_0, k_1\}, n_1\}$ subsumes the following dependencies:

$$\begin{aligned} \{n_0\} &\rightarrow \{n_0, k_1, n_1\}, \\ \{k_1\} &\rightarrow \{n_0, k_1, n_1\}. \end{aligned}$$

Applying the closure computation algorithm from Section 5.6 and replacing attributes by their equivalence classes yields the dependencies and equivalence classes shown in the table.

We can now return to our original question: can we discard the more expensive tree from Figure 5.22a in favor of the one in Figure 5.22b by considering the FDs holding in both trees instead of the keys? That is, we need to check if the following relationship holds:

$$FD^+(\bowtie^{2a}) \subseteq FD^+(\bowtie^{2b}). \tag{5.41}$$

Instead of computing the closure for both trees, we can go through all FDs in $AC^+(\bowtie^{2a})$ and check if they hold in the right tree as well. This is where the equivalence classes come in handy. Consider the following dependency from the left join tree:

$$\{\{n_0, k_1\}\} \rightarrow \{\{n_0, k_1\}, \{n_1, k_2\}, n_2\}.$$

We do not have to find an exact match for this dependency in the right tree. Instead, we have to find one in which at least one member of each equivalence class contained in the above dependency occurs on the same side of the matching dependency. The following dependency from the right side of the table meets these requirements:

$$\{n_0\} \rightarrow \{n_0, \{n_1, k_2\}, n_2\}.$$

In our example we find a match for every dependency from the left side of the table, leading us to the conclusion that Equivalence 5.41 holds. We can therefore safely discard the more expensive tree.

Taking a closer look at Table 5.2, we also see that $\kappa(\bowtie^{2b}) = \{\{n_0\}\}$, since all attributes that are present in the tree are determined by n_0. The key resulting from the key computation shown in Figure 5.22 is therefore not minimal, i.e., it is a superkey only.

This example represents the situation where using f-dominance does allow the elimination of a subtree, while k-dominance does not. However, there are also cases where the opposite holds, especially in the presence of non-inner joins. We present an example in Figure 5.23.

We assume the same relation schemas as in the previous example, and we are again interested in discarding the tree in Figure 5.23a because it is more

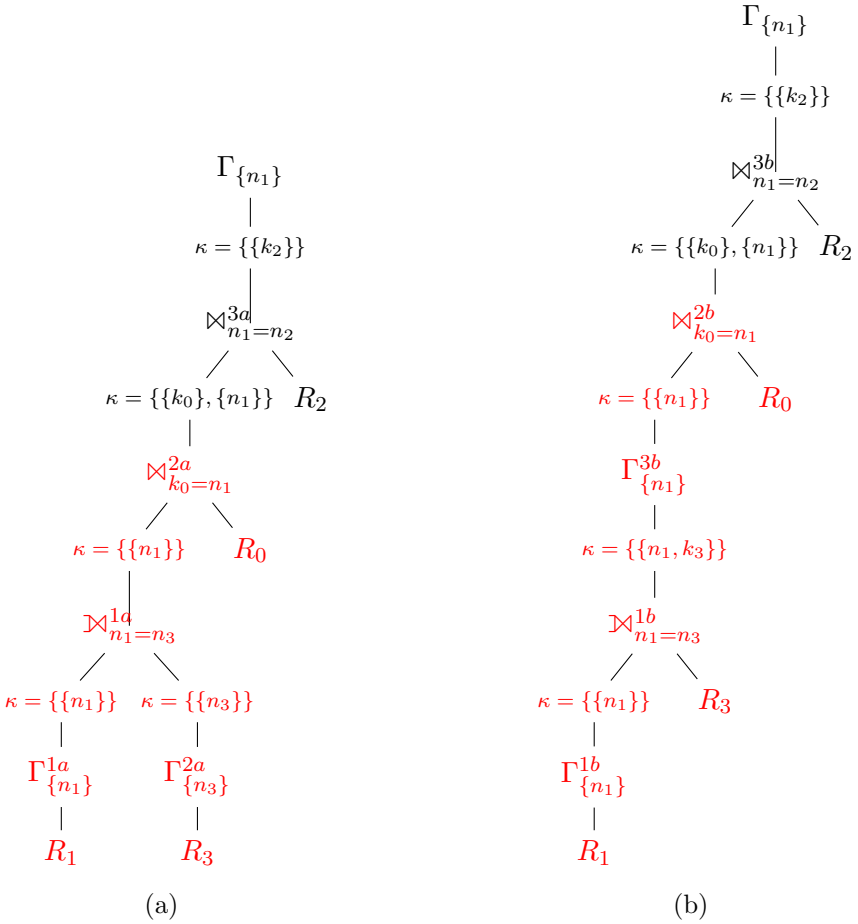


Figure 5.23: Two operator trees with keys

expensive with equal cardinality as the one in Figure 5.23b. Comparing the key sets of $\bowtie^{2a}$ and $\bowtie^{2b}$, we see that they are equal, i.e., the tree on the left-hand side can be discarded according to Definition 12. On the other hand, the requirements for f-dominance are not fulfilled, as can be seen in Table 5.3, which contains the FDs and equality constraints up to $\bowtie^2$, the root of the two subtrees we are comparing.

The dependency $\{\{k_0, n_1\}\} \rightarrow \{n_3\}$, which is contained in $AC^+(\bowtie^{2a})$, is not contained in $AC^+(\bowtie^{2b})$. This is because attribute n_3 is not available in the latter, since it is removed by Γ^3. To see that this is caused by the left outerjoin $\bowtie^1$, we replace it by an inner join. This results in an equivalence class $\{n_1, n_3\}$, which is later extended to $\{k_0, n_1, n_3\}$, turning the problematic dependency from above into $\{\{k_0, n_1, n_3\}\} \rightarrow \{\{k_0, n_1, n_3\}\}$. Since we only need to find one attribute from each equivalence class on the correct side of another dependency, the conditions for f-dominance are satisfied by the dependency $\{\{k_0, n_1\}\} \rightarrow \{\{k_0, n_1\}\}$ holding in $\bowtie^{2b}$.

Table 5.3: Functional dependencies for Figure 5.23

	Figure 5.23a		**Figure 5.23b**	
	AC$^+$	**EC**	**AC$^+$**	**EC**
R_0	$\{k_0\} \to \{k_0, n_0\}$	$\emptyset$	$\{k_0\} \to \{k_0, n_0\}$	$\emptyset$
R_1	$\{k_1\} \to \{k_1, n_1\}$	$\emptyset$	$\{k_1\} \to \{k_1, n_1\}$	$\emptyset$
R_3	$\{k_3\} \to \{k_3, n_3\}$	$\emptyset$	$\{k_3\} \to \{k_3, n_3\}$	$\emptyset$
Γ^1	$\{n_1\} \to \{n_1\}$	$\emptyset$	$\{n_1\} \to \{n_1\}$	$\emptyset$
Γ^2	$\{n_3\} \to \{n_3\}$	$\emptyset$	-	$\emptyset$
$\bowtie^1$	$\{n_1\} \to \{n_1, n_3\}$ $\{n_3\} \to \{n_3\}$	$\emptyset$	$\{n_1\} \to \{n_1, n_3\}$ $\{k_3\} \to \{k_3, n_3\}$	$\emptyset$
Γ^3	-	$\emptyset$	$\{n_1\} \to \{n_1\}$	$\emptyset$
$\bowtie^2$	$\{\{k_0, n_1\}\} \to \{\{k_0, n_1\}, n_0, n_3\}$ $\{n_3\} \to \{n_3\}$	$\{k_0, n_1\}$	$\{\{k_0, n_1\}\} \to \{\{k_0, n_1\}, n_0\}$	$\{k_0, n_1\}$

5.7.2 Pruning with Restricted Keys

In this section we propose a third pruning approach that makes use of keys and at the same time allows for a more effective pruning than k-dominance. Thereby, we combine the convenience of computing keys instead of functional dependencies with more opportunities for pruning. Again, we provide an example consisting of two alternative join trees for the same query. They are shown in Figure 5.24.

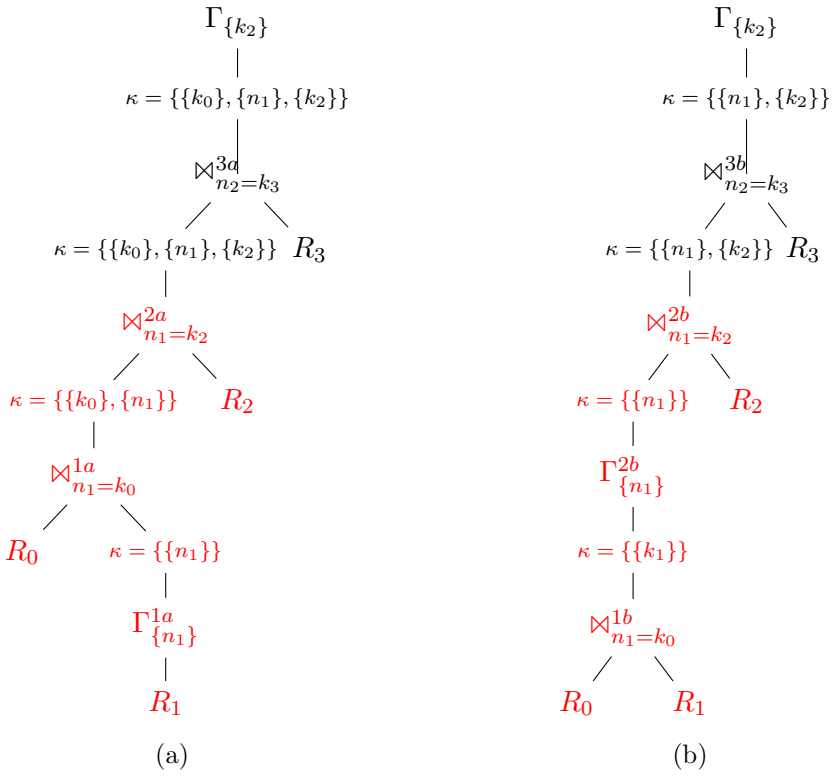


Figure 5.24: Two operator trees with keys

As before, we are comparing the subtrees for relation set $\{R_0, R_1, R_2\}$ and

we are interested in discarding the subtree in Figure 5.24a, assuming that it is more expensive than its counterpart on the right side and that both have equal cardinality. Using the key set as the pruning criterion, we notice that the tree on the left has a set containing three keys, whereas the one on the right only has two keys. Therefore, we decide to keep both trees since the third criterion for k-dominance is not met.

Going one level higher in the tree, we see that there is in fact no reason to keep the more expensive tree. In both trees the final grouping on $\{k_2\}$ has no effect because $\{k_2\}$ is a key of the tree rooted at $\bowtie^3$. Since the left tree contains a subtree that is more expensive than that contained in the tree on the right, the complete plan on the left can only be cheaper than the one on the right if it can omit the final grouping while the right plan cannot. This is not the case and, therefore, we could have removed the red subtree on the left, but k-dominance does not allow this.

We claim that the attribute set $\{k_0\}$ contained in $\kappa(\bowtie^{2a})$ but not in $\kappa(\bowtie^{2b})$, which inhibits the pruning, can be ignored since it is not referenced in any predicate further up in the tree or in the grouping attributes associated with the grouping at the top of the tree. Therefore, it does not influence the key constraints that hold in the following intermediate results, which in turn determine the necessity of the final grouping. The same argument implies that we can also ignore $\{n_1\}$. Removing these attributes from both $\kappa(\bowtie^{2a})$ and $\kappa(\bowtie^{2b})$, we see that the only remaining key in both sets is $\{k_2\}$. The sets are therefore equal and the third criterion for k-dominance is fulfilled, meaning that we can discard the more expensive subtree. This leads to a third notion of dominance. Before we define it, we define the *restricted key set* κ^- as follows:

$$\kappa^-(T) = \{K | K \in \kappa(T) \wedge K \subseteq G^+(T)\}.$$

Thus, the restricted key set of T contains only those keys of T that are subsets of $G^+(T)$. As stated in Section 5.3, the latter set contains all attributes that are referenced in a predicate or as part of a set of grouping attributes not belonging to T. We can now define *rk-dominance*:

Definition 13. *A join tree T_1 rk-dominates another join tree T_2 for the same set of relations if all of the following conditions hold:*

1. $Cost(T_1) \leq Cost(T_2)$
2. $|T_1| \leq |T_2|$
3. $\kappa^-(T_1) \supseteq \kappa^-(T_2)$.

Theorem 5. *Let T_2 be an arbitrary operator tree containing a subtree T_2^{sub}. Further, let T_1^{sub} be a tree rk-dominating T_2^{sub}. Then, the following must hold:*

$$\exists T_1 : T_1 \equiv T_2 \wedge Cost(T_1) \leq Cost(T_2),$$

where T_1^{sub} is a subtree of T_1.

Lemma 1. *Let T_{sub} be a subtree rooted at an inner node of a (sub)tree T. Then $G^+(T_{sub}) \supseteq G^+(T) \cap \mathcal{A}(T_{sub})$ follows.*

Proof. Let $O'(T_{sub})$ be the set of operators between the root of T_{sub} and the root of T:

$$O'(T_{sub}) = O(T) \setminus O(T_{sub}).$$

Further, let $P'(T_{sub})$ be the set of predicates associated with the operators in $O'(T_{sub})$:

$$P'(T_{sub}) = \bigcup_{\circ_p \in O'(T_{sub})} p.$$

Let $J(T_{sub})$ be the set of attributes provided by T_{sub} referenced in these predicates:

$$J(T_{sub}) = \{j | j \in (\bigcup_{p \in P'(T_{sub})} \mathcal{F}(p) \cap \mathcal{A}(T_{sub}))\}.$$

Then, the claim follows directly from

$$G^+(T) \cap \mathcal{A}(T_{sub}) = G^+(T_{sub}) \setminus (J(T_{sub}) \setminus G).$$

□

We prove the correctness of Theorem 5 by showing that substituting the condition $\kappa(T_1) \supseteq \kappa(T_2)$ from Definition 12 and the corresponding Theorem 4 with $\kappa^-(T_1) \supseteq \kappa^-(T_2)$ does not affect the optimality of the produced plan.

Proof of Theorem 5. The only modification we have made compared to Definition 12 is the additional condition $K \subseteq G^+(T)$ in the definition of $\kappa^-(T)$. Let T_{sub} be a subtree of T.

Our claim is that for any $K_{sub} \in \kappa(T_{sub})$ with $K_{sub} \nsubseteq G^+(T_{sub})$ the key K_{sub} can be disregarded. We need to show that this has no negative effect on $\kappa(T)$ in its ability to prevent unnecessary grouping operations.

Let $\kappa'(T_{sub}) = \{K | K \in \kappa(T_{sub}) \wedge K \nsubseteq G^+(T_{sub})\}$. Further, let $\kappa'(T)$ be the set of keys that cannot prevent unnecessary grouping operations on top of T with $\kappa'(T) = \{K | K \in \kappa(T) \wedge K \nsubseteq G^+(T)\}$. We show that through key propagation $K' \in \kappa'(T_{sub})$ can only have an impact on $\kappa'(T)$ but not on $\kappa(T) \setminus \kappa'(T)$. That is, all keys contained in $\kappa'(T_{sub})$ would have, according to our key propagation rules, ended up in $\kappa'(T)$, the set of "useless" keys in T.

Keys can be

1. extended($\times, \bowtie, \rtimes, \ltimes$),
2. eliminated ($\rtimes, \bowtie, \triangleright, \ltimes, \barwedge$),
3. left untouched ($\rtimes, \bowtie, \triangleright, \ltimes, \barwedge$) and
4. newly introduced (Γ).

Note that in none of these cases an existing key is made smaller. Let O' be the set of operators that are applied on the path in T between the root of T_{sub} and the root of T. If for one operator $\circ \in O'$ cases (2) or (4) apply to $\kappa(T_{sub})$, the proof is trivial since no key in $\kappa(T_{sub})$ propagates.

In general, $G^+(T_{sub}) \supseteq G^+(T) \cap \mathcal{A}(T_{sub})$ holds according to Lemma 1. Thus, we can deduce for every $K'_{sub} \in \kappa'(T_{sub})$ that $K'_{sub} \nsubseteq G^+(T) \cap \mathcal{A}(T_{sub})$ follows because of $K'_{sub} \nsubseteq G^+(T_{sub})$. Further, we know the relationship $K'_{sub} \subseteq \mathcal{A}(T_{sub})$ holds by definition. Hence, $K'_{sub} \nsubseteq G^+(T)$ follows. □

5.7.3 Pruning with Restricted FDs

So far, we have observed that we can often prune more subplans with FDs than with keys, but restricting the key set increases the effectiveness of key-based pruning. Applying the same principle to FDs by using a restricted set of FDs promises to further improve the pruning capabilities of our plan generator.

Again, we start by giving an example, consisting of two operator trees as shown in Figure 5.25, where we assume that the red subtree in Figure 5.25a is more expensive than the one in Figure 5.25b. Table 5.4 contains the FDs and

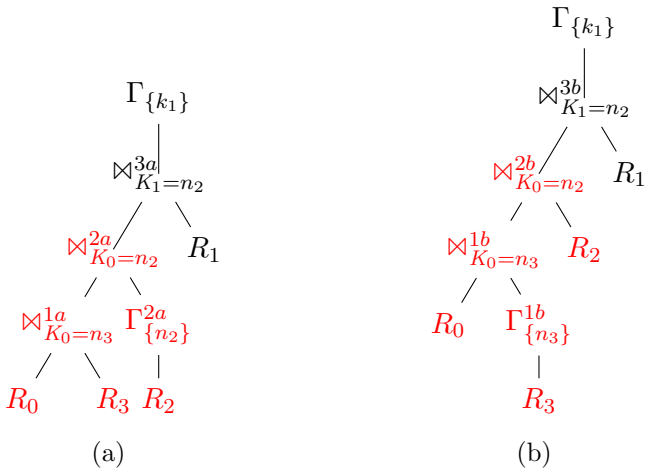


Figure 5.25: Two operator trees

equality constraints holding in each intermediate result up to the root nodes of the two subtrees. The FDs contained in $AC^+(\bowtie^{2a})$ are not contained in $AC^+(\bowtie^{2b})$, nor is $AC^+(\bowtie^{2b})$ a subset of $AC^+(\bowtie^{2a})$. Thus, we cannot discard either of the two trees based on f-dominance. More precisely, there are two dependencies that hinder the pruning: $\{k_3\} \rightarrow \{k_3, \{k_0, n_2, n_3\}, n_0\}$, which only holds in the left tree and $\{k_2\} \rightarrow \{k_2, \{k_0, n_2, n_3\}, n_0\}$, which only holds in the right tree.

The attributes k_2 and k_3 are not referenced in any of the join predicates above $\bowtie^2$, the root node of the two subtrees of interest. They are also not part of the grouping attributes at the topmost grouping operator. The only attribute from this subtree that is "still needed" further up in the tree is n_2. If we only consider those dependencies where the left-hand side contains n_2 for

Table 5.4: Functional dependencies for Figure 5.25

	Figure 5.25a	**Figure 5.25b**	**Figure 5.25**
	AC$^+$	**AC$^+$**	**EC**
R_0	$\{k_0\} \rightarrow \{k_0, n_0\}$	$\{k_0\} \rightarrow \{k_0, n_0\}$	$\emptyset$
R_2	$\{k_2\} \rightarrow \{k_2, n_2\}$	$\{k_2\} \rightarrow \{k_2, n_2\}$	$\emptyset$
R_3	$\{k_3\} \rightarrow \{k_3, n_3\}$	$\{k_3\} \rightarrow \{k_3, n_3\}$	$\emptyset$
Γ^1	-	$\{n_3\} \rightarrow \{n_3\}$	$\emptyset$
$\bowtie^1$	$\{\{k_0, n_3\}\} \rightarrow \{\{k_0, n_3\}, n_0\}$ $\{k_3\} \rightarrow \{k_3, \{k_0, n_3\}\}$	$\{\{k_0, n_3\}\} \rightarrow \{\{k_0, n_3\}, n_0\}$	$\{k_0, n_3\}$
Γ^2	$\{n_2\} \rightarrow \{n_2\}$	-	$\emptyset$
$\bowtie^2$	$\{\{k_0, n_2, n_3\}\} \rightarrow \{\{k_0, n_2, n_3\}, n_0\}$ $\{k_3\} \rightarrow \{k_3, \{k_0, n_2, n_3\}, n_0\}$	$\{\{k_0, n_2, n_3\}\} \rightarrow \{\{k_0, n_2, n_3\}, n_0\}$ $\{k_2\} \rightarrow \{k_2, \{k_0, n_2, n_3\}, n_0\}$	$\{k_0, n_2, n_3\}$

the comparison of the two trees, we can discard the subtree in Figure 5.25a. In analogy to the restricted key set κ^-, we define the *restricted FD set* FD^- as

$$FD^-(T) = \{f : \alpha \rightarrow \beta \mid f \in FD^+(T) \wedge \alpha \subseteq G^+(T)\}.$$

This leads to the definition of *rf-dominance*:

Definition 14. *A join tree T_1 rf-dominates another join tree T_2 for the same set of relations if all of the following conditions hold:*

1. $Cost(T_1) \leq Cost(T_2)$
2. $|T_1| \leq |T_2|$
3. $FD^-(T_1) \supseteq FD^-(T_2)$.

Theorem 6. *Let T_2 be an arbitrary operator tree containing a subtree T_2^{sub}. Further, let T_1^{sub} be a tree rf-dominating T_2^{sub}. Then, the following must hold:*

$$\exists T_1 : T_1 \equiv T_2 \wedge Cost(T_1) \leq Cost(T_2),$$

where T_1^{sub} is a subtree of T_1.

Proof. The only modification we have made compared to Definition 11 is the additional condition $\alpha \subseteq G^+(T)$ in the definition of $FD^-(T)$. Let T_{sub} be a subtree of T.

Our assumption is that for any dependency $f_{sub} : \alpha \rightarrow \beta \in FD^+(T_{sub})$ with $\alpha \nsubseteq G^+(T_{sub})$ the dependency f_{sub} can be disregarded. We need to show that this has no negative effect on $FD^+(T)$ in its ability to prevent unnecessary grouping operations.

Let $FD'(T_{sub}) = \{f : \alpha \rightarrow \beta | f \in FD^+(T_{sub}) \wedge \alpha \nsubseteq G^+(T_{sub})\}$. Further, let $FD'(T)$ be the set of dependencies that cannot prevent unnecessary grouping operations on top of T with $FD'(T) = \{f : \alpha \rightarrow \beta | f \in FD(T) \wedge \alpha \nsubseteq G^+(T)\}$. We show that, according to the rules for deriving functional dependencies, $f' \in FD'(T_{sub})$ can only have an impact on $FD'(T)$, but not on $FD(T) \setminus FD'(T)$. That is, all dependencies contained in $FD'(T_{sub})$ would have, according to the rules from Section 5.6, ended up in $FD'(T)$, the set of "useless" dependencies in T. Dependencies can be

1. eliminated ($\bowtie, \mathbb{M}, \triangleright, \ltimes, \overleftarrow{\Gamma}$),
2. left untouched ($\bowtie, \mathbb{M}, \mathbb{M}, \overleftarrow{\Gamma}$) and
3. newly introduced/extended ($\bowtie, \mathbb{M}, \mathbb{M}, \overleftarrow{\Gamma}, \Gamma$).

Note that in none of these variants the left side of a dependency is made smaller and that we compute the closures *before* considering to discard a dependency. This is important because we might discard a dependency f_{sub} that forms the transitive link between some attribute set $\gamma \subseteq G^+(T_{sub})$ and another dependency that arises further up in the tree.

Let O' be a set of operators that are applied on the path in T between the root of T_{sub} and the root of T. If for one operator $\circ \in O'$ case (1) applies to some $f_{sub} \in FD(T_{sub})$, the proof is trivial since then f_{sub} does not propagate.

In general, $G^+(T_{sub}) \supseteq G^+(T) \cap \mathcal{A}(T_{sub})$ holds according to Lemma 1. Thus, we can deduce for every $f'_{sub} : \alpha \rightarrow \beta \in FD'(T_{sub})$ that $\alpha \nsubseteq G^+(T) \cap \mathcal{A}(T_{sub})$ follows because of $\alpha \nsubseteq G^+(T_{sub})$. Further, we know the relationship $\alpha \subseteq \mathcal{A}(T_{sub})$ holds by definition. □

5.7.4 Pruning with Restricted Keys and Restricted FDs

Our observations from the previous sections suggest that we can benefit from using (r)f-dominance as the pruning criterion instead of (r)k-dominance, since it allows for the pruning of more subplans. On the other hand, there is also a cost associated with this approach, which lies in the higher complexity of computing and comparing the (restricted) closure instead of the (restricted) key set. This is why we propose a combination of rk-dominance and rf-dominance that maximizes the pruning capabilities of the plan generator and at the same time minimizes the overhead for evaluating the pruning criterion.

The idea is to always test rk-dominance first and only compute and compare the restricted closures of both plans if this test fails. Since in many cases rk-dominance is sufficient to discard a suboptimal plan, we only need to compute the closure for a fraction of all considered plans. We use the term *rkrf-dominance* when referring to this combined approach, even though it does not define a new form of dominance in the sense that it utilizes a new set of properties.

5.7.5 Summary

In this section five notions of dominance were presented. They can be distinguished along two dimensions, as shown in Table 5.5. On the one hand, we have those based on keys and on the other hand, the ones based on functional dependencies. Only rkrf-dominance makes use of both properties. Furthermore, the different pruning criteria can be categorized into those based on an unrestricted property set and those based on a restricted property set.

At this point it is impossible to make a sound statement on which of the different criteria is the best, because they all have their pros and cons. According to the observations from above, the restricted versions are likely to be superior to

Table 5.5: Different forms of dominance

	Non-Restricted	**Restricted**
Keys	K-Dominance	RK-Dominance, RKRF-Dominance
FDs	F-Dominance	RF-Dominance, RKRF-Dominance

their unrestricted counterparts. However, when it comes to the three restricted approaches, a tradeoff has to be made between their effectiveness as a pruning criterion and the overhead caused by maintaining the required property sets.

While keys are relatively cheap to compute and compare, the information they provide is incomplete and tends to leave some opportunities for pruning unused. Functional dependencies, on the other hand, are more expensive to maintain during plan generation, but promise to allow for more effective pruning.

In order to eliminate these uncertainties, a thorough experimental evaluation of the different approaches is provided in the next section.

5.8 Evaluation

Our evaluation consists of five major parts. We first measure the general impact of eager aggregation on the plan quality in terms of plan cost. The second part deals with the heuristics from Sections 5.5.2 and 5.5.3. There, we are mainly interested in how close to an optimal solution the heuristics come. After this, we turn our attention to the algorithms producing optimal plans by quantifying the impact of pruning. To this end, we measure the number of DP table entries with and without pruning. Having thus highlighted the importance of pruning, we then compare the numerous pruning approaches described in Section 5.7. Finally, we add groupjoins to the mix and examine their effect on the costs of the resulting plans. The results presented in this section have previously been published in [7, 8, 9].

5.8.1 Workload and Experimental Setup

All algorithms described in this chapter were implemented as an extension of the plan generator DPhyp (Figure 3.5). If not otherwise noted, the workload consists of randomly generated operator trees. Therefore, 10,000 trees are generated each for a certain parameter value, e.g., the number of relations contained in the tree. Duplicates are avoided by using the unranking procedure proposed by Liebehenschel to enumerate binary trees [27]. A grouping operator with a randomly chosen set of grouping attributes is placed at the top of the tree. For each input tree, a new subset of relations is randomly chosen from a set of 20 base relations, each with one key attribute and two non-key attributes. The relations differ in their cardinalities and attribute values, which are represented by randomly generated *logical profiles* [29]. Join operators are attached to the inner nodes of the tree. Either all join operators are inner joins, or the operator types are chosen randomly from the set $\{\bowtie, ⟕, ⟗\}$. A randomly generated binary equality predicate is attached to every join operator.

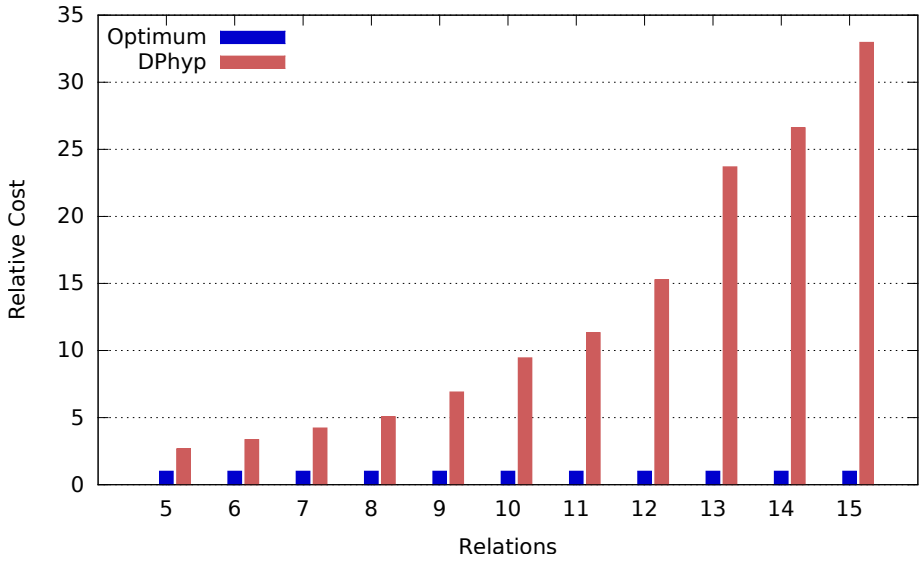


Figure 5.26: Relative plan costs DPhyp vs. optimum with inner joins

In most cases, the results are provided in the form of average values over the 10,000 input trees for a certain parameter value. We do not classify the workload by the shape of the query graph, as it is usually done when evaluating plan generators for pure join reordering. Instead, we are more interested in the complexity added by the extensions discussed in this chapter, which is strongly influenced by other factors, such as the number of foreign-key–key predicates.

Every run of the plan generator includes all of the steps shown in Figure 3.11. That is, the operator tree is first turned into a query hypergraph. The hyperedges are constructed as described in Section 4.5.1. Only then is the query graph passed to the plan generator. Moreover, cardinality estimation is conducted for every input tree by computing the profiles of all intermediate results during plan generation [29]. Costs are determined by the cost function C_{out}, as defined in Section 5.5.2.

Instead of the randomized workload, one experiment was carried out with selected queries from the TPC-H benchmark [5]. All experiments were run on an Intel Xeon E5-2690 V2 @ 3.00 GHz.

5.8.2 The Impact of Eager Aggregation

We are going to quantify the gain in plan quality enabled by eager aggregation. Figure 5.26 shows the average ratio between the plan cost achieved by the original DPhyp, which optimizes the join order and schedules a single grouping as the final step of query evaluation and the optimum that is achievable with eager aggregation.

The figure shows that the effect of applying eager aggregation on the costs of the resulting plans increases with increasing query size. The plans produced by DPhyp for queries with five relations are more than twice as expensive as the ones achievable with eager aggregation. For larger queries with ten relations,

the former are almost ten times as expensive as the latter. This trend continues, leading to a factor of 33 for queries with 15 relations.

Table 5.6 shows a comparison between DPhyp, our optimality-preserving plan generator applying k-dominance pruning, and the heuristics. We used a tolerance factor of 1.05 for H2. In this case, the workload is not randomly generated, but consists of the example query from Figure 5.1 (labeled "Ex") and three selected TPC-H queries (Q3, Q5, Q10) [5]. Query statistics are taken from a scale factor 1 instance of TPC-H.

The table contains the runtimes of the different plan generators and the resulting plan costs. Among the listed queries, Ex benefits most from eager aggregation, which is also reflected by the execution times we observed on different existing systems (see Section 5.1). Out of the shown queries, TPCH-Q5 profits least from eager aggregation.

Table 5.6: Optimization times and plan costs for TPC-H queries

	Ex	Q3	Q5	Q10
Time k-dominance [ms]	0.184	0.163	2.4	0.31
Time H1 [ms]	0.15	0.13	0.333	0.183
Time H2 [ms]	0.122	0.151	0.413	0.323
Time DPhyp [ms]	0.097	0.115	0.327	0.158
Rel. time k-dominance/DPhyp	1.9	1.42	7.34	1.96
Rel. time H1/DPhyp	1.55	1.13	1.02	1.16
Rel. time H2/DPhyp	1.26	1.31	1.26	2.04
Rel. cost k-dominance/DPhyp	6.1×10^{-4}	0.65	0.9	0.58
Rel. cost H1/DPhyp	6.1×10^{-4}	0.92	0.9	0.58
Rel. cost H2/DPhyp	6.1×10^{-4}	0.65	0.9	0.58

5.8.3 The Heuristics

Since the heuristics discussed in this chapter do not necessarily produce an optimal plan, we are first and foremost interested in how much the plans produced by the heuristics deviate from the optimum.

Figure 5.27 shows the plan cost achieved by H1 and H2 in relation to the optimal cost achievable with eager aggregation. For H2, four runs were made, each with a different tolerance factor F.

None of the heuristic plan generators produces optimal costs for every query, but all of them are significantly closer to optimality than DPhyp. H1 on average produces more expensive plans than H2 for all query sizes, regardless of the tolerance factor. Its average deviation from the optimum always lies between 6 and 14 percent. Note that H1 never produces a worse plan than the one without eager aggregation.

This is not the case with H2. Since it favors "more eager" subplans to a certain extent, even if they are more expensive than an equivalent subplan, the resulting plans can in theory be worse than those resulting from DPhyp. In our experiments, all of H2's plans were better or equally as good as those produced

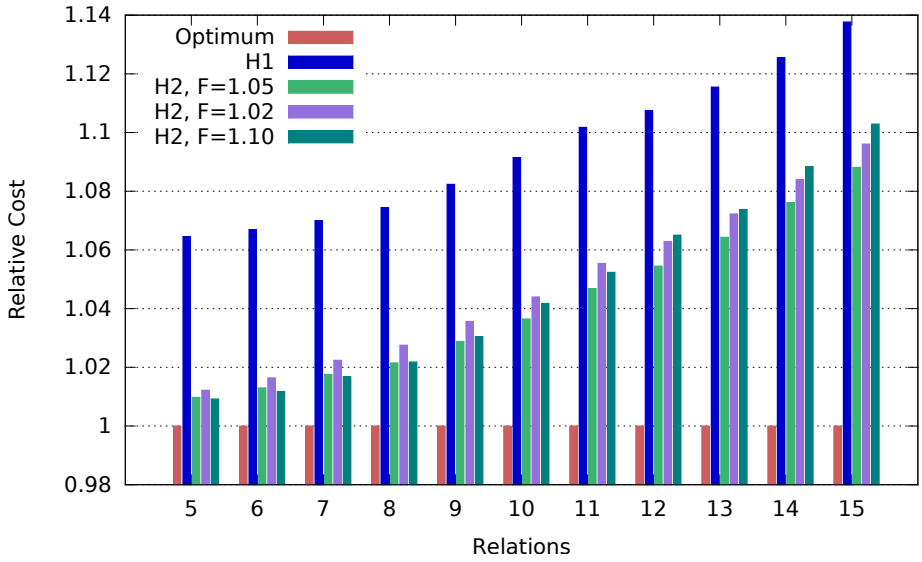


Figure 5.27: Relative plan costs with inner joins

by DPhyp. That is because the potential gain for pushing a grouping is larger than the potential harm caused by wrongfully pushing a grouping. That way, bad decisions at one point were always outweighed by cost savings at another point. Clearly, this is not generally the case since it strongly depends on the cost model and the characteristics of the query.

Compared to H1, there were cases where H2 performed worse. However, the plan resulting from H2 was never more than twice as expensive as the plan generated by H1 for the same query. On the other hand, there were cases where H2's plan caused only one third of the cost of H1's plan. Fittingly, the figure shows that on average the more aggressive strategy pays off in the sense that the plans generated by H2 are slightly cheaper than those produced by H1.

Clearly, the addition of eager aggregation to the plan generator causes some overhead, even if it is done in the form of heuristics. Figures 5.28 and 5.29 show the average runtimes for DPhyp and the two heuristics, either with inner joins only or with randomly selected operators from $\{\bowtie, ⟗, ⟕\}$. Since the runtimes of the heuristics are virtually identical, they are combined into a single bar.

Compared to DPhyp, the heuristics are slower by a factor of 8 to 15 for queries with inner joins. The slowdown factor slightly increases with increasing query size. With outerjoins, the heuristics are slower by an almost constant factor of five. In the latter case, the runtimes of the plan generators are significantly faster than with inner joins only, which is due to the reduced search space size that results from the limited reorderability of outerjoins.

5.8.4 The Impact of Dominance Pruning

The following experimental results highlight the importance of pruning when the goal is to generate an optimal plan with eager aggregation. Figures 5.30 and 5.31 show the average runtime of the plan generator without pruning de-

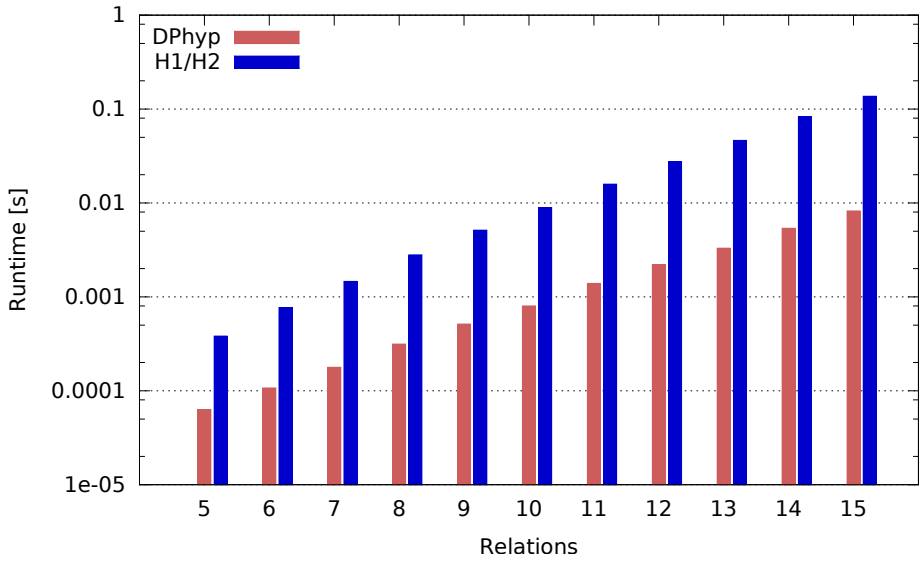


Figure 5.28: Runtimes with inner joins

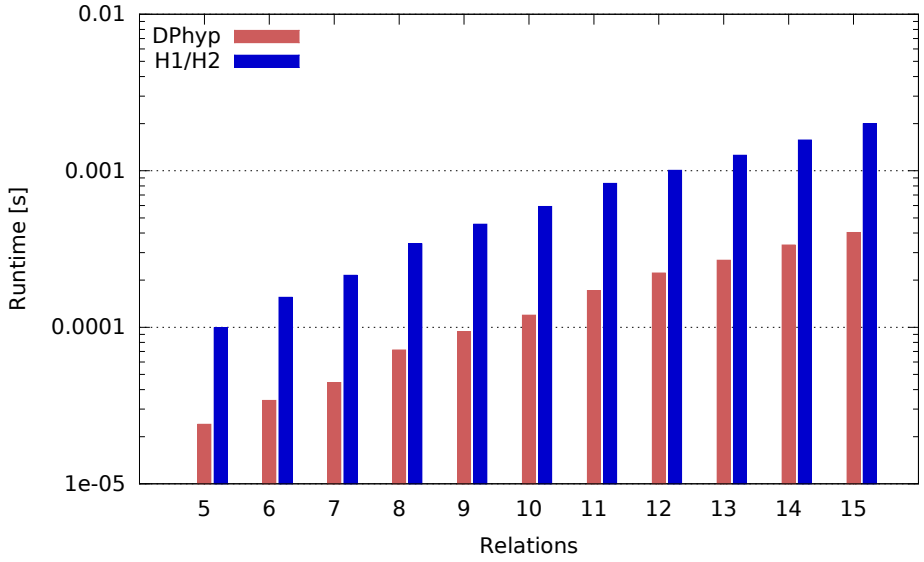


Figure 5.29: Runtimes with inner joins and outerjoins

scribed in Section 5.5.4 and the version implementing rk-dominance pruning compared to that of DPhyp. Again, runtimes were measured once for input queries containing only inner joins and once for queries containing outerjoins as well.

The figures show the impact of pruning on the runtime of the plan generator. While the plan generator without pruning takes 82 seconds for a query with eight relations and only inner joins, the one implementing rk-dominance pruning takes only 5 milliseconds. As expected, DPhyp is even faster and only takes 0.3 milliseconds for queries of this size. With outerjoins, the numbers for the

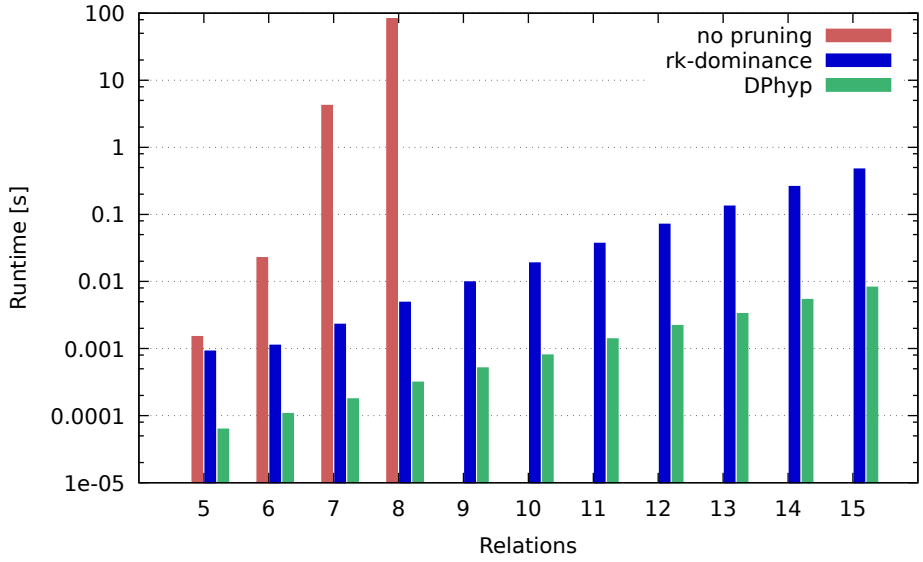


Figure 5.30: Runtimes with inner joins

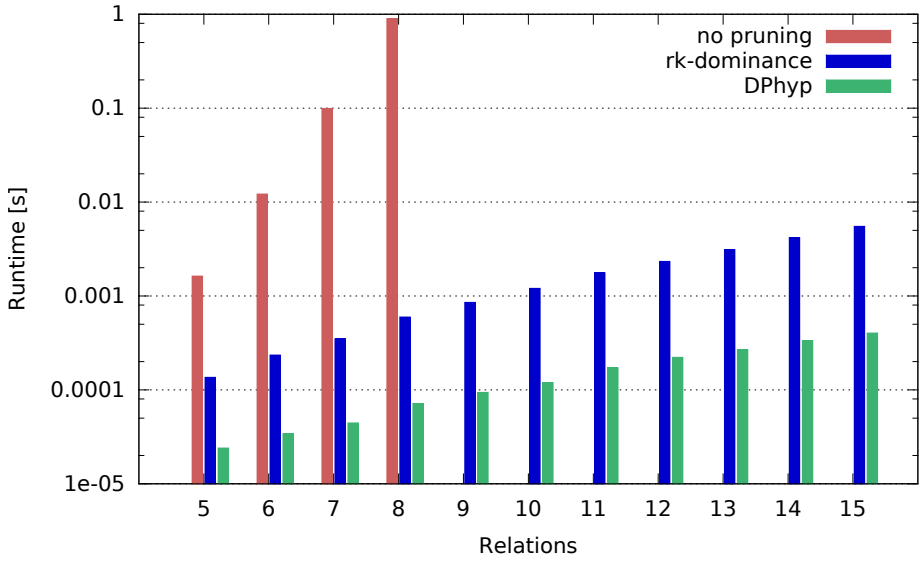


Figure 5.31: Runtimes with inner joins and outerjoins

same query size are again smaller: without pruning, query optimization takes 0.9 seconds on average and with rk-dominance pruning it takes 0.5 milliseconds. DPhyp takes 0.07 milliseconds on average to produce a plan with eight relations.

The reason for the excessive runtimes without pruning lies in the large number of subplans stored in the DP table. Figures 5.32 and 5.33 illustrate this by showing the number of entries stored in the DP table after successful plan generation for both algorithms. Without pruning, almost four million plans on average are stored in the DP table if the input query contains only inner joins. Pruning with rk-dominance reduces this number to 91. DPhyp and the

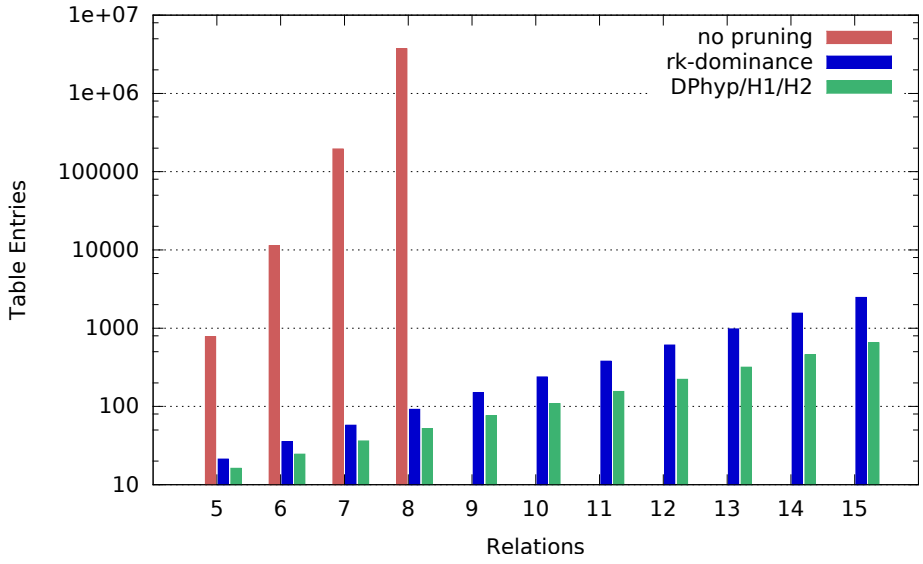


Figure 5.32: Number of table entries with inner joins

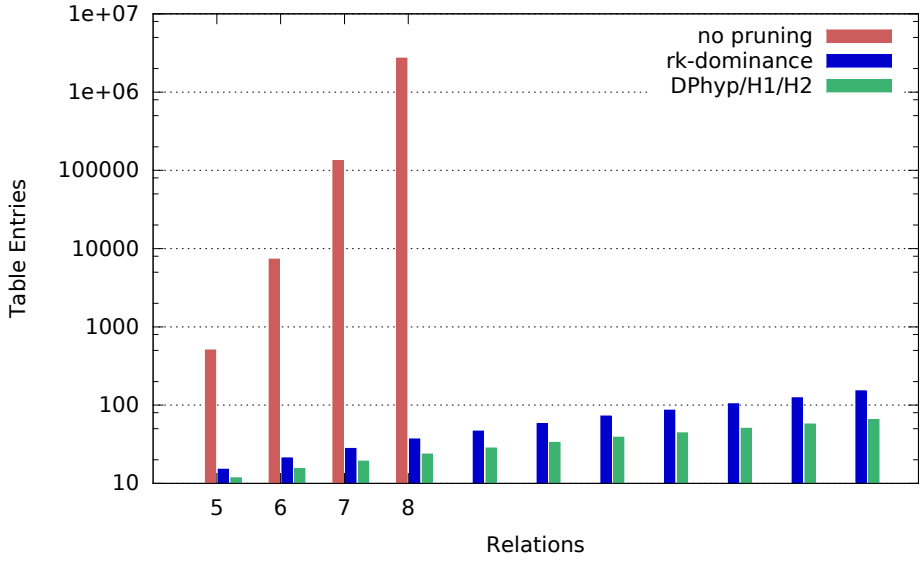


Figure 5.33: Number of table entries with inner joins and outerjoins

heuristics need to store only 51 plans on average. Adding outerjoins to the mix, these numbers amount to 2.7 million, 37 and 23, respectively.

5.8.5 Comparing the Pruning Approaches

We implemented the five pruning criteria presented in Section 5.7 in our extended version of DPhyp. We refer to the resulting plan generators by the name of the pruning criterion they implement. That is, the plan generator implementing f-dominance is labeled *f*, the one implementing rk-dominance is labeled *rk*

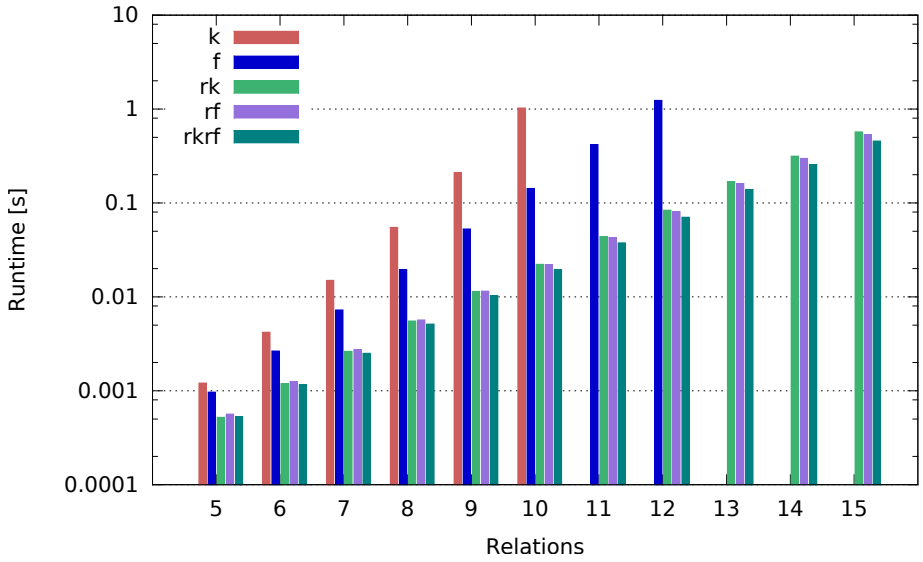


Figure 5.34: Runtimes with inner joins

and so on.

Runtime

Figures 5.34 and 5.35 show the runtimes of the five plan generators without groupjoins. The runtimes shown in Figure 5.34 result from queries containing only inner joins, while Figure 5.35 depicts queries containing inner, left outer and full outerjoins. Since the runtimes of k and f are so high, we did not run them for queries with more than eight or twelve relations, respectively.

In both cases we fix the proportion of foreign-key–key join predicates to eighty percent. The proportion of foreign-key–key joins has an impact on the runtime of the plan generators, especially the ones dealing with unrestricted property sets, since these predicates tend to keep the sets of keys and FDs small, making the comparison of said sets faster and increasing the chance of one plan dominating another. We consider eighty percent a rather cautious assumption and assume this number to be higher in most real queries.

Both figures confirm that a more effective pruning criterion generally results in faster plan generation. While the difference is marginal for small queries, it grows with an increasing number of relations. For queries with 15 relations and different join operators, k needs 1.4 seconds on average, while *rkrf* requires only 0.0015 seconds, making it almost three orders of magnitude faster. We can also see that the three algorithms working with restricted property sets have almost equal runtimes. However, *rk* and *rkrf* are faster than *rf*, which can be explained by the higher overhead for computing and comparing the closure, as demanded by rf-dominance.

To give an impression of how big this overhead is, we divided the runtimes of the different plan generators by the number of plan comparisons performed during plan generation. For queries with 15 relations containing inner joins and

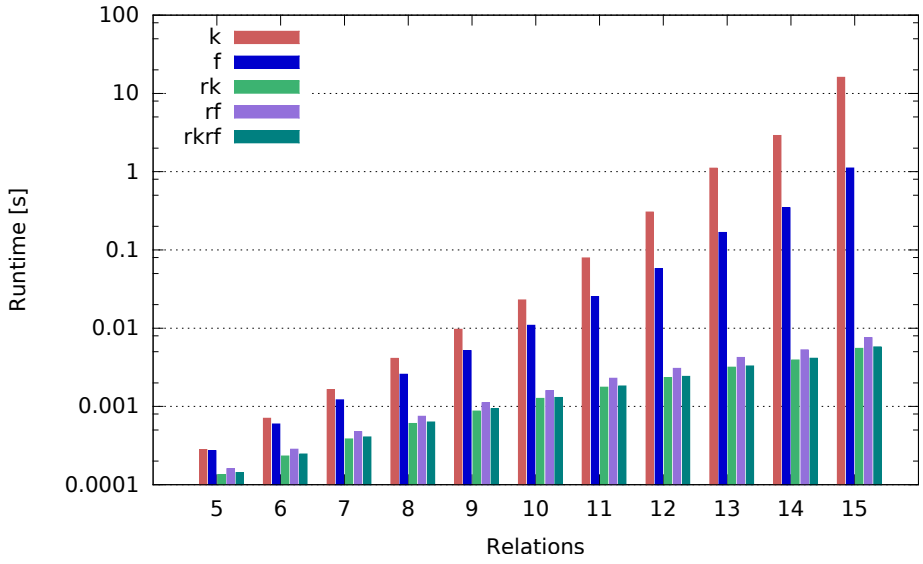


Figure 5.35: Runtimes with inner joins and outerjoins

outer joins, we measured the following numbers for "time per plan comparison": 23 / 306 / 2,073 / 3,705 / 3,077 nanoseconds for k- / f- / rk- / rf- / rkrf-dominance, respectively. Note that these numbers are based on the assumption that the plan comparisons are the dominating influence on the plan generator's runtime, which may not be true, especially for k-dominance.

When considering queries containing only inner joins, we observe the following trend for larger queries (see Figure 5.34): since the search space is so large for these queries, the search space restriction achieved by the pruning criterion becomes more critical, causing rk-dominance to become less and less efficient when compared to rf-dominance and rkrf-dominance.

As stated above, the proportion of foreign-key–key join predicates has a significant impact on the runtime of the different plan generators. Figures 5.36 and 5.37 show the runtimes for queries with ten relations and an increasing percentage of foreign-key–key joins from 0 to 100 with 10,000 input trees each.

Memory Usage

The reason for the runtime difference between k and the rest becomes obvious when we look at the number of DP table entries produced by the different algorithms, as depicted in Figures 5.38 and 5.39. As suggested by our observations in the previous sections, the least effective pruning criterion is k-dominance and the most effective is rf-dominance. Combining the latter with rk-dominance results in the same number of table entries, since they are equivalent in their pruning capability and differ only in the way they achieve it. With outerjoins, the average number of table entries is 22,300 / 146 for k / *rkrf* for 15 relations. Queries limited to inner joins have a much bigger search space, resulting in more table entries, which is reflected in the results of our experiments: here,

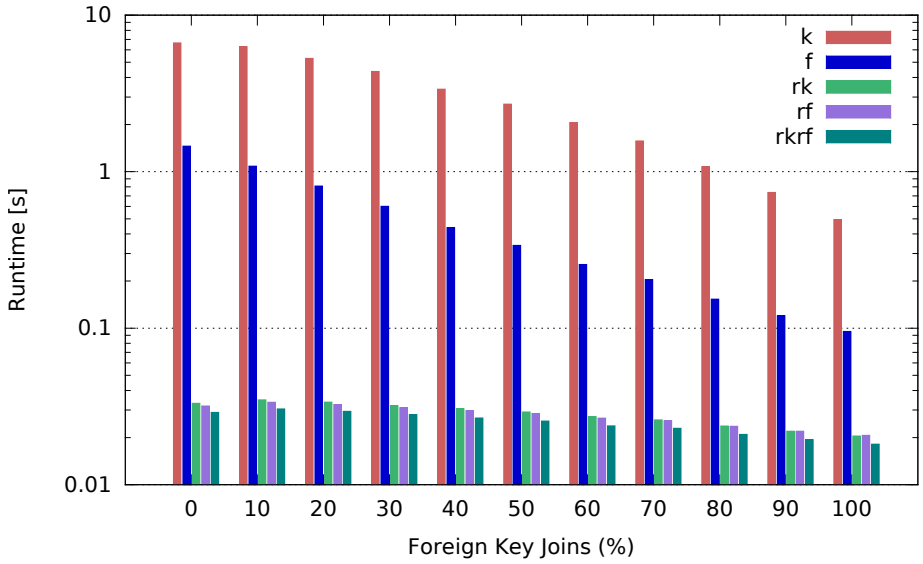


Figure 5.36: Runtimes for 10 relations with inner joins

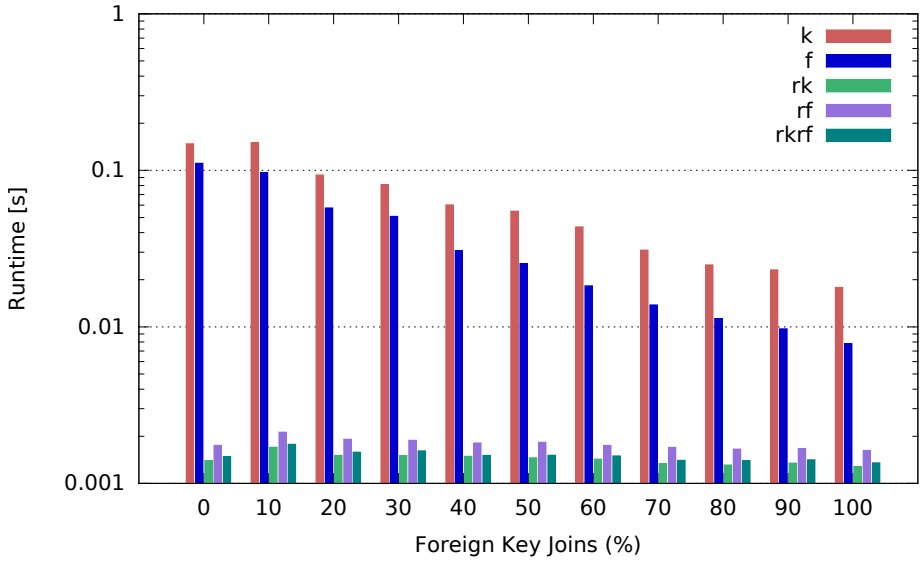


Figure 5.37: Runtimes for 10 relations with inner joins and outerjoins

we have 7,800 / 223 table entries on average for the same two plan generators and queries with ten relations.

5.8.6 The Impact of Groupjoins

This section deals with the plan generators applying eager aggregation and introducing groupjoins. We use the same labels for the different algorithms as above and add the prefix *gj* if the respective plan generator introduces groupjoins.

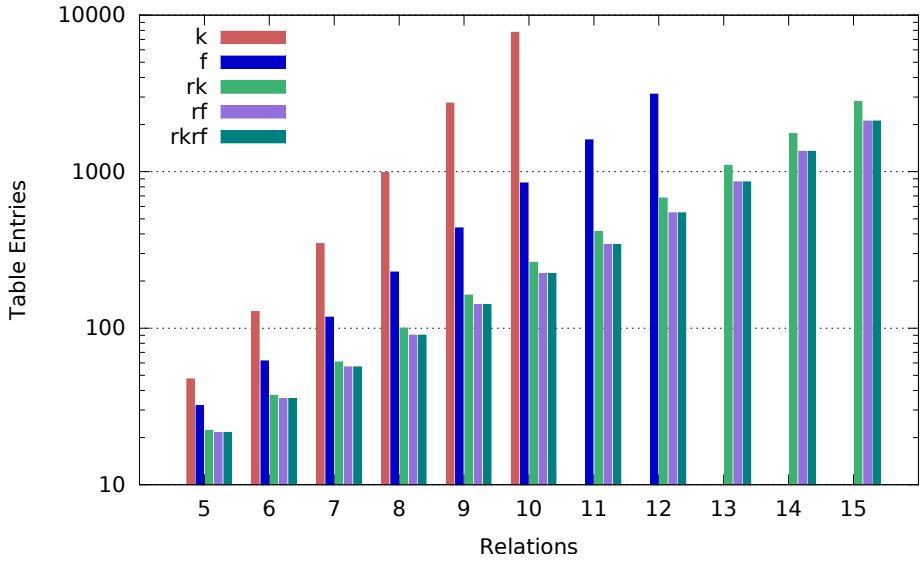


Figure 5.38: Number of table entries with inner joins

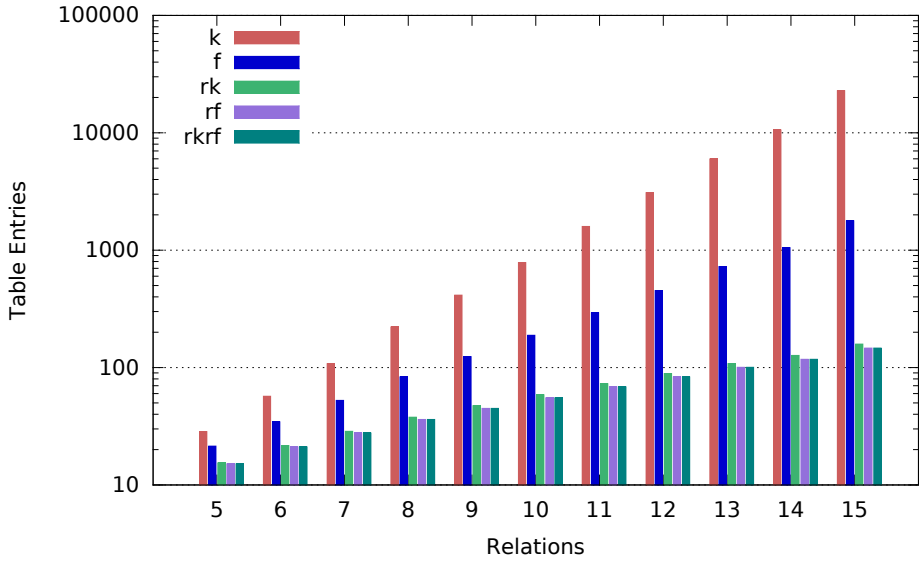


Figure 5.39: Number of table entries with inner joins and outerjoins

Plan Quality

Once we enable the introduction of groupjoins, we are sometimes able to achieve better plans than with "pure" eager aggregation. In this case, we also observe differences between the plans resulting from a key-based algorithm and the ones produced by an algorithm based on functional dependencies. This is because, as explained in Section 5.4, the requirements for applying a groupjoin cannot be precisely checked if only keys are known. Thus, the plan generator may fail to introduce a groupjoin if only keys are known, even though it would be possible.

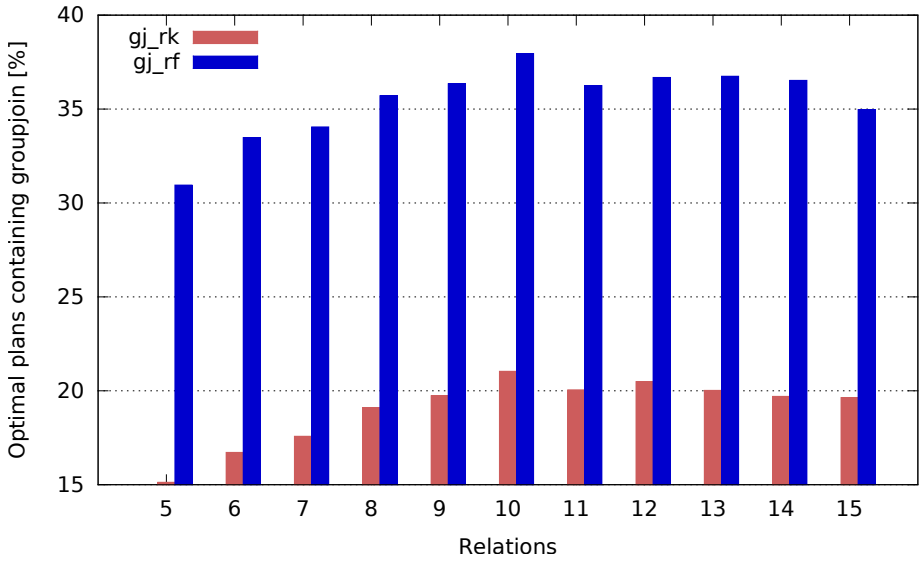


Figure 5.40: Pct. of optimal plans containing groupjoins with inner joins

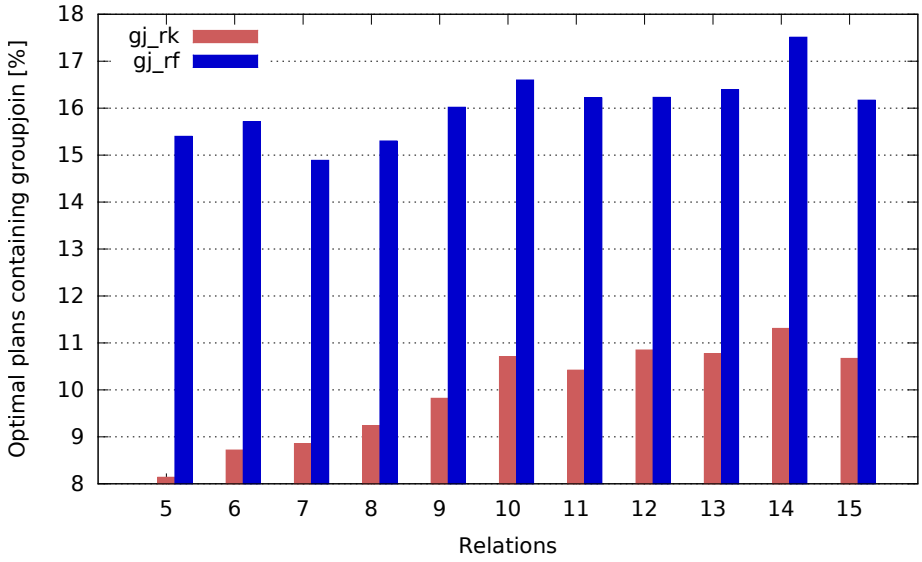


Figure 5.41: Pct. of optimal plans containing groupjoins with inner- and outerjoins

Figures 5.40 and 5.41 illustrate this by showing the percentage of optimal plans containing a groupjoin resulting from either of the two algorithms. For example, out of the 10,000 input queries with only inner joins and ten relations, 38 percent of the plans produced by *gjrf* contain at least one groupjoin. For *gjrk* this number amounts to 21 percent.

Table 5.7 contains the costs achieved by *gjrk* and *gjrf* in relation to the costs achieved without groupjoins for 2 to 15 relations. We only provide the minimum value, i.e., the best relative cost achieved over all 10,000 queries

Table 5.7: Relative plan costs groupjoins/no groupjoins

	inner joins		inner/outerjoins	
Relations	gjrk	gjrf	gjrk	gjrf
2	0.58	0.58	0.58	0.58
3	0.74	0.69	0.74	0.69
4	0.75	0.72	0.78	0.73
5	0.75	0.62	0.8	0.69
6	0.68	0.67	0.77	0.68
7	0.72	0.45	0.79	0.74
8	0.67	0.63	0.81	0.81
9	0.7	0.58	0.86	0.75
10	0.72	0.53	0.83	0.83
11	0.74	0.55	0.88	0.8
12	0.71	0.63	0.91	0.78
13	0.65	0.58	0.89	0.84
14	0.73	0.58	0.89	0.81
15	0.75	0.53	0.90	0.7

of one size. That is because groupjoins can only be applied in a fraction of the considered queries and the cost saving achieved by introducing groupjoins fluctuates considerably depending on the characteristics of the query. But as we are going to see subsequently, groupjoins do not add much complexity on top of eager aggregation, so there is no real downside of introducing them. Therefore, the values shown in the table give an impression of the potential that is wasted by doing without them. As the values indicate, groupjoins can sometimes reduce the plan cost to less than 50 percent. The differences between *gjrk* and *gjrf* reveal that we may no longer be able to find the optimal plan by relying on keys instead of functional dependencies as soon as groupjoins are taken into consideration.

Due to the random nature of our workload, the grouping attributes are not necessarily equal to the attributes referenced in the join predicates contained in the query. In real queries, the grouping attributes and the join attributes often overlap, which enables the application of the groupjoin. Thus, the groupjoin can only replace a fraction of the joins in our input queries and its benefits are sometimes outweighed by the costs of the remaining joins. As can be seen in the table, this effect is most pronounced when the key-based plan generator is applied to queries containing outerjoins, because in this scenario few groupjoins can be applied. In previous work a speedup factor of more than three was reported for TPC-H query 13 after introducing groupjoins. The complete benchmark was sped up by a factor of 1.5 [33].

Runtime

Figures 5.42 and 5.43 show the runtimes of the two plan generators with groupjoins and their counterparts without groupjoins. The figures show that

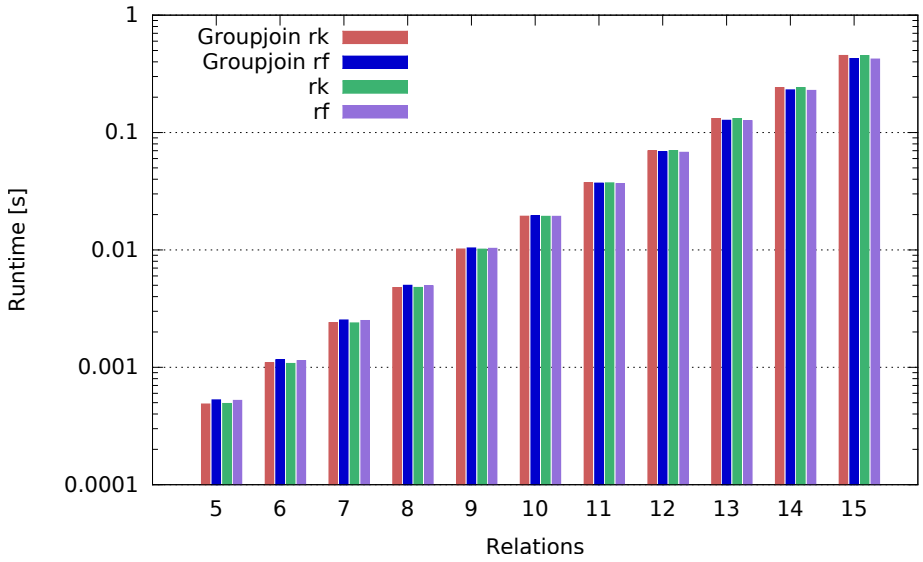


Figure 5.42: Runtime with groupjoins and inner joins

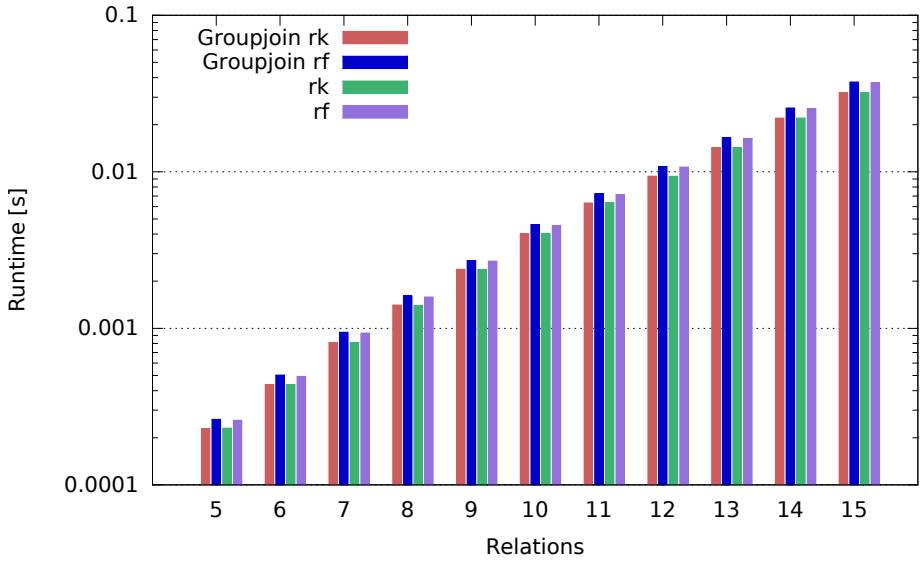


Figure 5.43: Runtime with groupjoins, inner- and outerjoins

the differences in runtime between the respective variants are marginal. This proves that the overhead caused by the introduction of groupjoins is negligible and outweighed by the potential cost savings demonstrated in the previous subsection, even if they only occur in a fraction of the tested queries. This can be explained by the fact that the main overhead caused by the new extension lies in checking the requirements for introducing a groupjoin. However, the dominating influence factor on the runtimes of our algorithms is the number of plans in the plan table. As we are going to see in the next subsection, this number is little affected by the introduction of groupjoins.

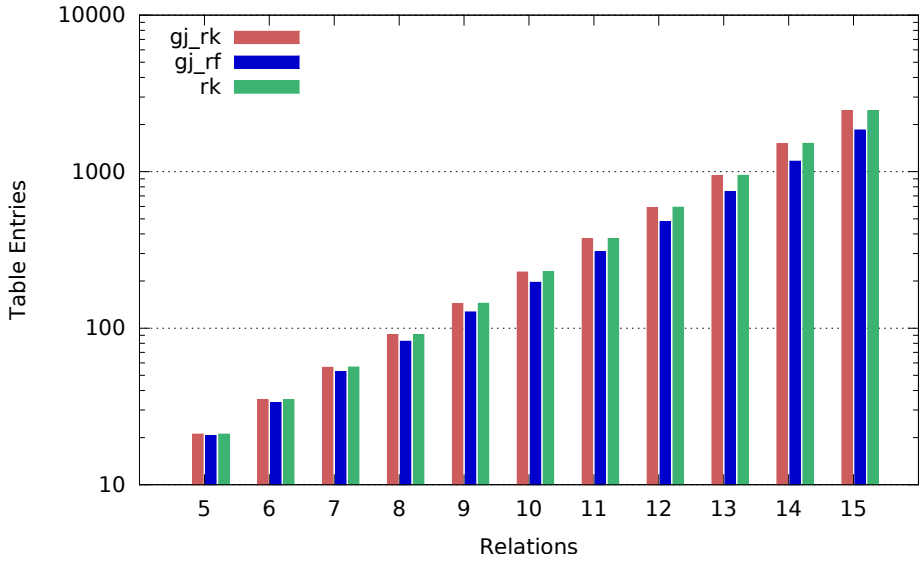


Figure 5.44: Number of table entries with groupjoins and inner joins

Memory Usage

Figures 5.44 and 5.45 show the number of table entries stored in the DP-Table after plan generation for the algorithms with groupjoins and their counterparts without. The numbers are almost equal, which can be explained as follows: when a groupjoin is applied to replace a sequence of grouping and left outerjoin or inner join, the resulting subplan has the same properties (cardinality, keys and functional dependencies) as the original one. The only difference lies in the plan cost. Thus, in many cases, the groupjoin plan just replaces the corresponding plan with pushed-down grouping. But there are also cases where a plan containing a grouping and a join is dominated by another plan and would not have been inserted into the plan table in the first place, whereas the corresponding groupjoin plan is inserted because of its lower cost. In these cases, the introduction of groupjoins increases the number of plans stored in the DP table.

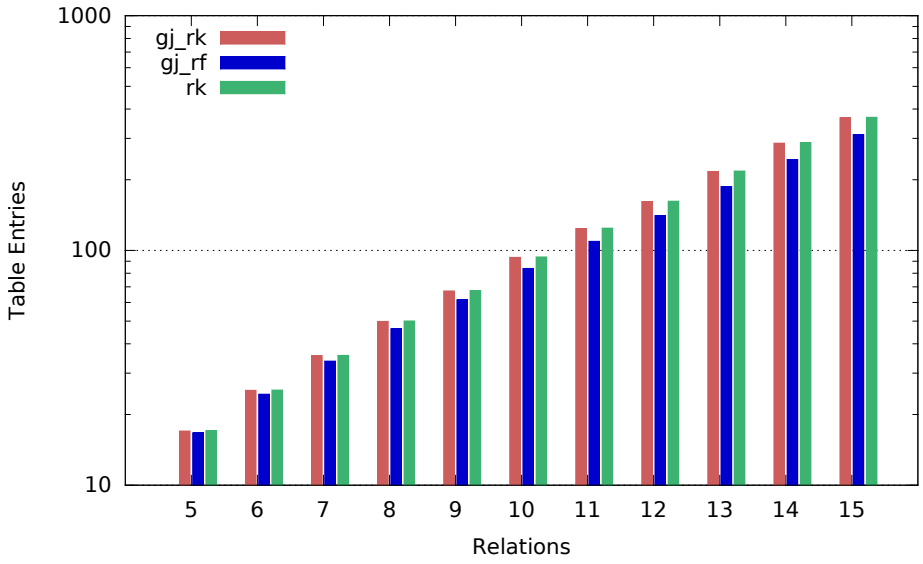


Figure 5.45: Number of table entries with groupjoins, inner- and outerjoins

6 Conclusion and Outlook

The conclusion consists of two parts. The first part summarizes the previous chapters. The second part contains an outlook on some outstanding query optimization problems and possible solutions based on the tools presented in this thesis.

6.1 Summary

The first contribution of this thesis is the description of a modular architecture for generative plan generators. While the focus of this work lies on DP-based plan generators, only few modifications are necessary to apply the same concepts to memoization-based plan generators, since the two only differ in the way their enumerators work. Decomposing the plan generator into several independent modules, that can be exchanged individually, results in an extensible plan generation framework. It allows for the combination of any desired enumerator with an optional conflict detector and a custom plan builder. Exchanging the plan builder to make it capable of exploiting plan properties offers great potential for extending the overall functionality of the plan generator.

Building on this, two open problems connected to generative plan generators were solved. The first one is the correct and complete reordering of join operators with different reordering properties. We showed that the existing solutions for this problem are faulty, since they allow the generation of invalid plans. With our approach we do not only achieve correctness, but also completeness, i.e., one of our conflict detectors allows for the complete and correct enumeration of the core search space. Moreover, our solution is easier to extend than the existing ones, making the addition of new operators as simple as adding their reordering properties to a set of tables [31].

The second problem we tackled is the efficient reordering of grouping operators and join operators. The solution to this problem consists of two major parts: the deduction of algebraic equivalences for reordering grouping and non-inner joins and the description of an efficient plan generator implementing this transformation. The key to the first part is the proposition of a slightly modified version of the outerjoin that pads non-matching tuples with freely choosable default values instead of null values. The implementation in a plan generator strongly depends on the availability of effective pruning measures to reduce the size of the plan generator's search space. Our experiments show that this is possible with a fairly small overhead, more precisely by maintaining information about key properties during plan generation [6, 8].

The aforementioned approach exposes a pattern that can be applied to solve a wide range of problems. It consists of the identification of interesting plan properties and the deduction of pruning criteria based on these properties. To

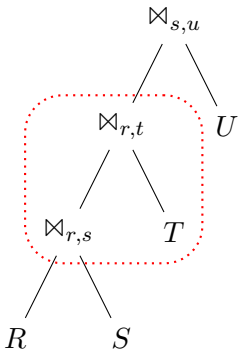


Figure 6.1: Hash team

demonstrate the wide applicability of this approach, we provide an outlook on possible solutions for a few selected query optimization problems in the following section.

6.2 Outlook

This section provides an overview of some open issues in the context of generative plan generators that can be solved with the help of plan properties. We do not describe the resulting algorithms in great detail, but rather sketch out a possible solution and leave the rest to the reader.

6.2.1 Hash Teams

If several successive join operators in a query plan are implemented as hash joins hashing on the same attributes, they can share a hash table. This way, multiple join operators can be applied with a single build phase.

For an example, consider the operator tree in Figure 6.1. Both join predicates associated with the join operators inside the red box reference attribute r. Hence, we can use R as the build input for the first hash join and probe the hash table first with tuples from S and then from T to compute the result of both joins using a single hash table. The same general idea applies if more than two successive join operators share a common join attribute.

Gräfe et al. introduced the term *hash team* for such a group of hash join operators. Their paper provides some information on the implementation of hash teams in a query execution engine [19]. However, it leaves open the question how to efficiently generate query plans taking the introduction of hash teams into account. To fill this gap we once more aim to find a solution that can be integrated in any of the basic DP-based plan generators presented in Chapter 3.

In addition to optimizing the join order, we now have to decide for each join operator whether or not we want to combine it with its predecessor to form a hash team, if possible. This, in turn, depends on which join argument is chosen as the build input for each hash join. Without the option of sharing hash tables,

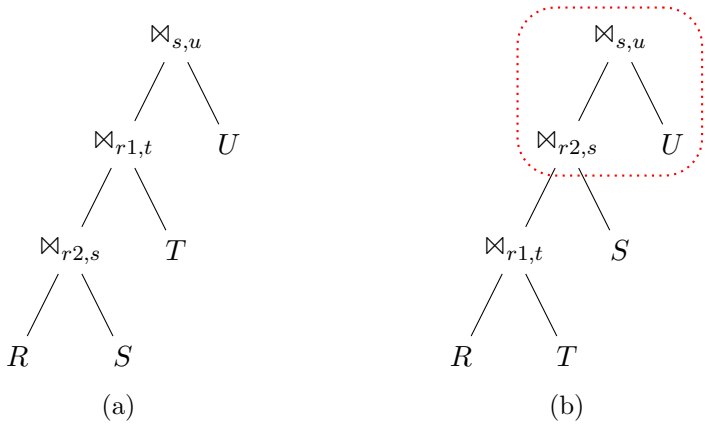


Figure 6.2: Alternative join trees with hash team

the smaller input is typically used as the build input. This is usually the best choice for one join operator alone. However, it may be beneficial to choose the bigger input if this enables the application of a hash team in which the same hash table can be shared between two or more join operators.

Figure 6.2 shows two exemplary operator trees for joining four relations $\{R, S, T, U\}$. The main difference between the two is the subtree joining the relation set $\{R, S, T\}$. While the tree on the left-hand side first joins relations R and S and then joins the result with T, the tree on the right-hand side first joins R and T and finally adds U. Moreover, the two join operators at the top of the right tree both hash on s, which is indicated by the underlining of s. This way, the tree on the right-hand side can combine the two joins contained in the red box in a hash team. Since this might be cheaper than executing the join operators independently, the tree on the right-hand side may be cheaper than its counterpart on the left-hand side. This may apply even if its join order would otherwise not be optimal.

Thus, choosing the cheapest join order for $\{R, S, T\}$ and discarding the tree applying a more expensive join order, as we would do in a traditional bottom-up plan generator, may hinder the subsequent introduction of a hash team and thereby destroy the optimality of the final solution. In other words, a suboptimal subsolution may be part of the optimal overall solution, which contradicts Bellman's Principle of Optimality. Consequently, we have to keep more than one plan for every plan class instead of discarding suboptimal plans solely based on their estimated cost. Thus, the modifications we need to make to our plan generator closely resemble the ones described in great detail in Section 5.5. More precisely, the plan generator's plan builder has to be extended in such a way that it builds all possible plans for joining a given ccp (S_1, S_2) and inserts them all into the DP table. Later on, these plans are used for building plans for the ccp containing relation set S with $S = S_1 \cup S_2$.

Figure 6.3 shows three equivalent operator trees that have to be considered for the ccp $(\{R, S\}, \{T\})$ when hash teams are taken into account. Note, however, that only the cheaper tree out of the first two trees has to be kept in the DP table. That is because in both these trees the top-most join operator hashes on

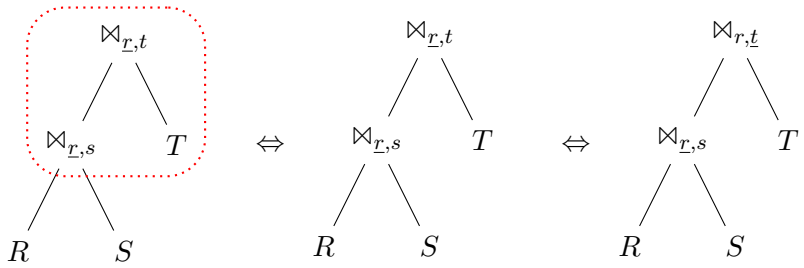


Figure 6.3: Trees for ccp $(\{R, S\}, \{T\})$

r. However, this is true only under the assumption that implementing a hash team is always cheaper than consecutively building two or more identical hash tables.

By now, we have learned that storing all possible plans severely limits the applicability of the plan generator because only very small queries can be optimized with this approach. Following our tried and tested approach, we aim to identify an optimality-preserving pruning criterion that allows to reduce the number of plans stored in the DP table without jeopardizing the optimality of the resulting plans. To this end, we define the set of hash attributes $h(T)$ of an operator tree T. It contains the attributes for which a hash table is built at the top-most operator in T. There are two possible plans for joining $\{R, S\}$ and $\{T\}$, as shown in Figure 6.2a: one where $h(T) = \{r1\}$ and one where $h(T) = \{t\}$. On the other hand, the corresponding subtree in Figure 6.2b can have the values $h(T) = \{r2\}$ or $h(T) = \{s\}$

To decide whether a certain plan T_1 dominates another plan T_2 in this context, we have to compare the cost and the hash attributes of the respective plans. This leads to the following definition of *hash-dominance*:

Definition 15. *A join tree T_1 hash-dominates another join tree T_2 for the same set of relations if all of the following conditions hold:*

1. $Cost(T_1) \leq Cost(T_2)$
2. $h(T_1) = h(T_2)$.

Since not all attributes referenced in a join predicate are again used in a subsequent predicate, we are only interested in those attributes that are contained in a set $J^+(T)$. It contains all attributes provided by the relations contained in the join tree T that are referenced in a join predicate outside of T. With this, we define the *restricted set of hash attributes* h^- as follows:

$$h^-(T) = h(T) \cap J^+(T).$$

We can now define rh-dominance as follows:

Definition 16. *A join tree T_1 rh-dominates another join tree T_2 for the same set of relations if all of the following conditions hold:*

1. $Cost(T_1) \leq Cost(T_2)$
2. $h^-(T1) = h^-(T2)$.

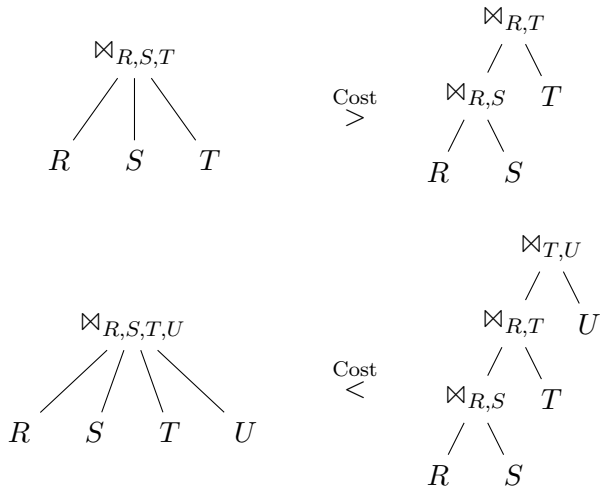


Figure 6.4: N-way join vs. sequence of binary joins

6.2.2 N-Way Joins

Somewhat similar to hash teams are n-way joins. While hash teams constitute a specialized optimization for multiple hash joins, n-way joins allow the combination of several binary joins into a single operator. This concept is not necessarily limited to hash-based algorithms, since many different n-way join implementations exist [24, 39, 42].

Applying an n-way join instead of a sequence of binary joins can have several advantages. Firstly, it reduces the number of intermediate results. Secondly, the operator can adapt to the sizes of the join arguments during processing, making it more resilient to bad cardinality estimates. Moreover, some implementations can already emit a prefix of the result before any of the inputs is completely processed. This is impossible with a sequence of hash joins, because there the respective build inputs have to be processed completely before any result tuples can be produced.

For the plan generator, however, the availability of n-way joins poses the challenge of deciding when to apply an n-way join instead of a sequence of "regular" binary join operators. If an n-way join is always cheaper than the equivalent combination of binary joins, there is no real problem. In this case, we can check if an n-way join is applicable and, if so, always prefer the plan with the n-way join. Whether or not an n-way join can be applied, strongly depends on the implementation of the n-way join and the characteristics of the joins it is supposed to replace. For example, outerjoins may not be supported, meaning that a sequence of binary joins including an outerjoin cannot be replaced by an n-way join. We are not going into more detail on how meeting these requirements can be ensured, since representing the respective properties with adequate data structures is merely an implementation detail.

What is more interesting at this point is the scenario where an n-way join joining a small number of relations is more expensive than the equivalent se-

quence of binary operators, but as soon as more relations are added, the n-way join becomes the cheaper alternative. Figure 6.4 shows an example. In the example, the n-way join joining $\{R, S, T\}$ is more expensive than two binary joins achieving the same result. Once U is added, applying the n-way join instead of multiple binary joins becomes the better alternative. This could be due to some overhead linked to the implementation of the n-way join that pays off only with a larger number of relations. Thus, the plan generator should not discard the n-way plan for $\{R, S, T\}$, even though it is more expensive than the alternative plan, since another join might later be included in the n-way join. So again, none of the two plans can be discarded because Bellman's Principle of Optimality is compromised in this case.

As before, our goal is to reduce the number of plans stored in the DP table by an appropriate definition of dominance. The simple solution in this case is to add a single property to each plan that equates to *true* if the topmost operator in the plan is an n-way join, and *false* otherwise. We denote this property by *n-way(T)* for an operator tree T and define *n-way-dominance* as follows:

Definition 17. *A join tree T_1 n-way-dominates another join tree T_2 for the same set of relations if all of the following conditions hold:*

1. $Cost(T_1) \leq Cost(T_2)$
2. $n\text{-}way(T_1) = n\text{-}way(T_2)$.

In other words, two plans are comparable only if both of them contain an n-way join as their topmost operator. In this case we only have to store the cheaper of the two in the DP table.

6.2.3 Distribution

In a distributed setting the cost of an execution plan strongly depends on the associated communication cost. Oftentimes, the most significant cost that occurs when two relations residing on two different sites in the distributed system are joined, is the cost of shipping one or both relations to one common site. The shipping cost depends on the amount of data that needs to be transferred and which sites are involved in the join. The amount of data, in turn, depends on the join algorithm. Numerous approaches for implementing distributed joins exist and many of them are designed to minimize the communication cost.

In this setting the cost of a plan alone is not enough to determine if it is better than another plan. Instead, one has to take into account the site on which the result of the plan is produced. Consider Figure 6.5 for a simple example.

In the example the distributed system consists of two sites: site A and site B. Copies of relations R and S are kept on both sites, whereas T resides exclusively on site A and U resides exclusively on site B. Two possible plans for the plan class defined by $\{R, S, T\}$ are shown in the figure. The tree on the left represents the plan where the three relations are joined locally on site A, causing no communication cost. In the plan on the right S is shipped to site B to be joined with R, before T is added to the plan. Since this causes some communication

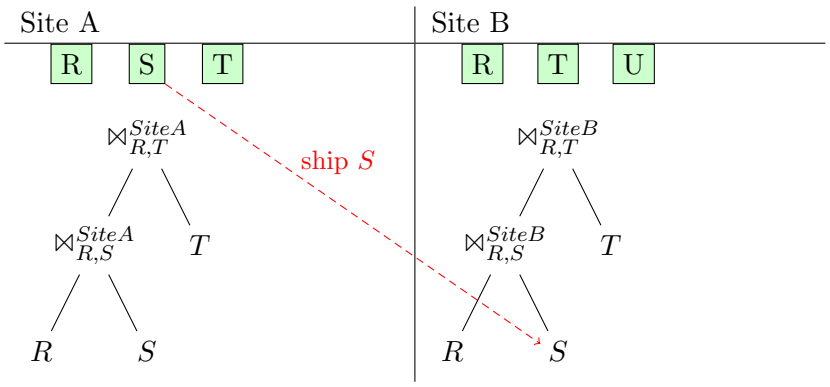


Figure 6.5: Join trees in a distributed system

cost, the plan on the right is more expensive than the one on the left and is not kept in the DP table. Next, relation U needs to be joined with T. Since U resides on site B and the result of the left plan resides on site A, one of the two has to be shipped to the other. If both these options are more expensive than shipping S from site A to site B, the previously discarded plan actually was better than all the remaining options.

Once more, this can be solved by adding a physical plan property that indicates the site on which the result of a plan resides. We denote this by $site(T)$ for an operator tree T. Two plans are comparable only if they produce their respective result at the same site. This is captured by our definition of *site-dominance*:

Definition 18. *A join tree T_1 site-dominates another join tree T_2 for the same set of relations if all of the following conditions hold:*

1. $Cost(T_1) \leq Cost(T_2)$
2. $site(T_1) = site(T_2)$.

With this, the plan on the right will never be dominated by the plan on the left. However, one case is conceivable where this would be desirable: if the combined cost of computing the result of the left tree and shipping it to site B is smaller than the cost of the right plan, there is no need to keep the right plan.

Operators that are introduced into the query plan to ensure a certain value of a plan property are known as *enforcers* [18] or *glue operators* [28]. We denote by $ship(T)$ the application of a *shipping* enforcer to T. It moves the result of T to the site of the tree that T is compared with. With this, we can rewrite site-dominance as follows:

Definition 19. *A join tree T_1 site-dominates another join tree T_2 for the same set of relations if all of the following conditions hold:*

1. $Cost(ship(T_1)) \leq Cost(T_2)$

This way the search space of the plan generator can be further reduced.

Bibliography

[1] G. Bhargava, P. Goel, and B. Iyer. Hypergraph based reorderings of outer join queries with complex predicates. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 304–315, 1995.

[2] Surajit Chaudhuri and Kyuseok Shim. Including group-by in query optimization. In *Proc. of the Int. Conf. on Very Large Data Bases*, pages 354–366, 1994.

[3] Sophie Cluet and Guido Moerkotte. Efficient evaluation of aggregates on bulk types. In *In Proc. Int. Workshop on Database Programming Languages*, 1995.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[5] Transaction Processing Performance Council. TPC Benchmark H. `http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.2.pdf`, 2017.

[6] M. Eich and G. Moerkotte. Dynamic programming: The next step. Technical report, University of Mannheim, 2014.

[7] M. Eich and G. Moerkotte. Dynamic programming: The next step. In *Proc. of the IEEE Int. Conf. on Data Engineering*, pages 903–914, 2015.

[8] Marius Eich, Pit Fender, and Guido Moerkotte. Faster plan generation through consideration of functional dependencies and keys. In *Proc. of the Int. Conf. on Very Large Data Bases*, pages 756–767, 2016.

[9] Marius Eich, Pit Fender, and Guido Moerkotte. Efficient generation of query plans containing group-by, join, and groupjoin. *The VLDB Journal*, Forthcoming 2017.

[10] P. Fender and G. Moerkotte. A new, highly efficient, and easy to implement top-down join enumeration algorithm. In *Proc. of the IEEE Int. Conf. on Data Engineering*, pages 864–875, 2011.

[11] P. Fender and G. Moerkotte. Reassessing top-down join enumeration. *IEEE Transactions on Knowledge and Data Engineering*, 24(10):1803–1818, 2012.

[12] P. Fender and G. Moerkotte. Top down plan generation: From theory to practice. In *Proc. of the IEEE Int. Conf. on Data Engineering*, pages 1105–1116, 2013.

[13] P. Fender, G. Moerkotte, T. Neumann, and V. Leis. Effective and robust pruning for top-down join enumeration algorithms. In *Proc. of the IEEE Int. Conf. on Data Engineering*, pages 414–425, 2012.

[14] Pit Fender and Guido Moerkotte. Counter strike: Generic top-down join enumeration for hypergraphs. In *Proc. of the Int. Conf. on Very Large Data Bases*, pages 1822–1833, 2013.

[15] C. Galindo-Legaria and M. Joshi. Orthogonal optimization of subqueries and aggregation. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 571–581, 2001.

[16] C. Galindo-Legaria and A. Rosenthal. How to extend a conventional optimizer to handle one- and two-sided outerjoin. In *Proc. of the IEEE Int. Conf. on Data Engineering*, pages 402–409, 1992.

[17] C. Galindo-Legaria and A. Rosenthal. Outerjoin simplification and reordering for query optimization. *ACM Transactions on Database Systems*, 22(1):43–73, Marc 1997.

[18] G. Graefe and W. McKenna. Extensibility and search efficiency in the volcano optimizer generator. In *Proc. of the IEEE Int. Conf. on Data Engineering*, pages 209–218, 1993.

[19] Goetz Graefe, Ross Bunker, and Shaun Cooper. Hash joins and hash teams in microsoft SQL server. In *Proc. of the Int. Conf. on Very Large Data Bases*, pages 86–97, 1998.

[20] Goetz Graefe and David J. DeWitt. The exodus optimizer generator. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 160–172, 1987.

[21] Laura M. Haas, Michael J. Carey, Miron Livny, and Amit Shukla. Seeking the truth about ad hoc join costs. *VLDB Journal*, 6(3):241–256, 1997.

[22] Toshihide Ibaraki and Tiko Kameda. On the optimal nesting order for computing n-relational joins. *ACM Transactions on Database Systems*, 9(3):482–502, September 1984.

[23] Alfons Kemper and Thomas Neumann. Hyper: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Proc. of the IEEE Int. Conf. on Data Engineering*, pages 195–206, 2011.

[24] Ramon Lawrence. Using slice join for efficient evaluation of multi-way joins. *Data and Knowledge Engineering*, 67(1):118–139, 2008.

[25] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? In *Proc. of the Int. Conf. on Very Large Data Bases*, pages 204–215, 2015.

[26] J. Liebehenschel. Ranking and unranking of lexicographically ordered words: An average-case analysis. *Journal of Automata, Languages, and Combinatorics*, 2(4):227–268, 1997.

[27] J. Liebehenschel. Lexicographical generation of a generalized dyck language. Technical Report 5/98, University of Frankfurt, 1998.

[28] Guy M. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 18–27, 1988.

[29] G. Moerkotte. Building query compilers. `http://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf`, 2014.

[30] G. Moerkotte and T. Neumann. Dynamic programming strikes back. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 539–552, 2008.

[31] Guido Moerkotte, Pit Fender, and Marius Eich. On the correct and complete enumeration of the core search space. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 493–504, 2013.

[32] Guido Moerkotte and Thomas Neumann. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In *Proc. of the Int. Conf. on Very Large Data Bases*, pages 930–941, 2006.

[33] Guido Moerkotte and Thomas Neumann. Accelerating queries with group-by and join by groupjoin. *Proc. of the Int. Conf. on Very Large Data Bases*, 4(11), 2011.

[34] K. Ono and G. Lohman. Measuring the complexity of join enumeration in query optimization. In *Proc. of the Int. Conf. on Very Large Data Bases*, pages 314–325, 1990.

[35] G.N. Paulley. Exploiting functional dependence in query optimization. PhD thesis, University of Waterloo, Canada, 2000.

[36] J. Rao, B. Lindsay, G. Lohman, H.Pirahesh, and D. Simmen. Using EELs: A practical approach to outerjoin and antijoin reordering. Technical Report RJ 10203, IBM, 2000.

[37] J. Rao, B. Lindsay, G. Lohman, H. Pirahesh, and D. Simmen. Using EELs: A practical approach to outerjoin and antijoin reordering. In *Proc. of the IEEE Int. Conf. on Data Engineering*, pages 595–606, 2001.

[38] A. Rosenthal and C. Galindo-Legaria. Query graphs, implementing trees, and freely-reorderable outerjoins. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 291–299, 1990.

[39] N. Roussopoulos and H. Kang. A pipeline n-way join algorithm based on the 2-way semijoin program. *IEEE Transactions on Knowledge and Data Engineering*, 3(4):486–495, 1991.

[40] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, 1979.

[41] B. Vance and D. Maier. Rapid bushy join-order optimization with cartesian products. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 35–46, 1996.

[42] Stratis D. Viglas, Jeffrey F. Naughton, and Josef Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *Proc. of the Int. Conf. on Very Large Data Bases*, pages 285–296, 2003.

[43] G. von Bültzingsloewen. Optimizing SQL queries for parallel execution. *SIGMOD Record*, 18(4):17–22, 1989.

[44] W. Yan. Rewriting optimization of SQL queries containing group-by. PhD thesis, University of Waterloo, Canada, 1995.

[45] W. Yan and P.-A. Larson. Performing group-by before join. Technical Report CS 93-46, Dept. of Computer Science, University of Waterloo, Canada, 1993.

[46] W. Yan and P.-A. Larson. Performing group-by before join. In *Proc. of the IEEE Int. Conf. on Data Engineering*, pages 89–100, 1994.

[47] W. Yan and P.-A. Larson. Eager aggregation and lazy aggregation. *Proc. of the Int. Conf. on Very Large Data Bases*, pages 345–357, 1995.

[48] W. Yan and P.-A. Larson. Interchanging the order of grouping and join. Technical Report CS 95-09, Dept. of Computer Science, University of Waterloo, Canada, 1995.