Faster Plan Generation through Consideration of Functional Dependencies and Keys

Marius Eich
University of Mannheim
marius.eich@uni-mannheim.de

Pit Fender
Oracle Labs
pit.fender@oracle.com

Guido Moerkotte
University of Mannheim
moerkotte@uni-mannheim.de

ABSTRACT

It has been a recognized fact for many years that query execution can benefit from pushing group-by operators down in the operator tree and applying them before a join. This so-called eager aggregation reduces the size(s) of the join argument(s), making join evaluation faster. Lately, the idea enjoyed a revival when it was applied to outer joins for the first time and incorporated in a state-of-the-art plan generator. However, this recent approach is highly dependent on the use of heuristics because of the exponential growth of the search space that goes along with eager aggregation. Finding an optimal solution for larger queries calls for effective optimality preserving pruning mechanisms to reduce the search space size as far as possible. By a more thorough investigation of functional dependencies and keys, we provide a set of new pruning criteria and evaluate their effectiveness with respect to the runtime and memory consumption of the resulting plan generator.

1. INTRODUCTION

The idea of reordering group-by operators and joins was proposed already two decades ago ([12, 13, 14, 11, 1]) and has since been implemented in many commercial query optimizers. However, it was always limited to inner joins only.

In a recent paper, Eich and Moerkotte revived the topic by showing that the optimal placement of group-by operators is possible in the presence of non-inner joins as well, thus enabling query optimizers to apply this powerful optimization technique to a whole new class of queries.

They describe a plan generator capable of reordering group-by and a wide range of different join operators. While their approach performs well for small queries, queries with more than ten relations can only be handled by abandoning optimality and relying on heuristics [4].

The reason for this limitation is the lack of an effective optimality-preserving pruning criterion to limit the size of the search space and thereby allow the optimization of larger queries. A quick complexity analysis shows the importance of pruning in this context: A binary operator tree with n relations contains $2n-2$ edges, and we can attach a group-by to each of these edges and on top of the root, resulting in $2n-1$ possible positions for a group-by. If one considers all valid combinations of these positions for every tree, the additional overhead caused by the optimal placement of group-by operators is in $O(2^{2n-1})$.

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 group-by at this position does not need to be considered.

We give an in-depth analysis of four new optimality-preserving pruning criteria and an existing one that was proposed in [4]. They are derived by a careful investigation of keys and functional dependencies. We describe the pruning criteria with the help of some examples and evaluate them experimentally, thereby showing that they can speed up the plan generator by orders of magnitude. The correctness proofs for all pruning criteria discussed in this paper can be found in [3].

Section 2 contains some preliminaries concerning the notation used in this paper and the basics of a bottom-up plan generator. In Section 3 we take a closer look at the information needed during plan generation which is captured in the form of interesting plan properties. Our main contribution is contained in Sections 4, 5, 6 and 7, where we discuss the different pruning criteria. In Section 8, we show the results of our experiments and subsequently conclude the paper in Section 9.

2. PRELIMINARIES

2.1 Algebraic Operators

In this section we provide definitions for the algebraic operators we will be using throughout the rest of the paper. We use standard set notation to denote bags.

We define the group-by operator Γ as

$$\Gamma_{G;a_1:f_1,\dots,a_k:f_k}(e) := \{y \circ [a_1 : x_1, \dots, a_k : x_k] \mid y \in \Pi^D_G(e), \\ x_i = f_i(\{z|z \in e,\ z.G = y.G\})\},$$

for some set of grouping attributes G. The attributes $a_1 \dots a_k$ are created by applying the aggregation vector $F = (f_1, \dots, f_k)$, consisting of k aggregate functions, to the grouped tuples. We denote by $\Pi^D_A(e)$ the duplicate-removing projection onto the set of attributes A, applied to the expression e. The resulting relation only contains values for those attributes that are contained in A and no duplicate values. The aggregate functions contained in F are then applied to groups of tuples taken from this relation. The groups contain tuples with equal values in the grouping attributes. The results are stored in attributes $a_1, \dots, a_k$.

Henceforth, we use a shorter notation for the group-by, where we abbreviate the specification of the aggregation vector and the

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org.
Proceedings of the VLDB Endowment, Vol. 9, No. 10
Copyright 2016 VLDB Endowment 2150-8097/16/06.

$$
\begin{array}{rcll}
e_1 \times e_2 & := & \{r \circ s | r \in e_1, s \in e_2\} & (1)\\
e_1 \bowtie_p e_2 & := & \{r \circ s | r \in e_1, s \in e_2, p(r,s)\} & (2)\\
e_1 \ltimes_p e_2 & := & \{r | r \in e_1, \exists s \in e_2, p(r,s)\} & (3)\\
e_1 \triangleright_p e_2 & := & \{r | r \in e_1, \nexists s \in e_2, p(r,s)\} & (4)\\
e_1 ⟕_p e_2 & := & (e_1 \bowtie_p e_2) \cup ((e_1 \triangleright_p e_2) \times \{\perp_{\mathcal{A}(e_2)}\} & (5)\\
e_1 ⟗_p e_2 & := & (e_1 \bowtie_p e_2) & \\
 & & \cup((e_1 \triangleright_p e_2) \times \{\perp_{\mathcal{A}(e_2)}\} & \\
 & & \cup(\{\perp_{\mathcal{A}(e_1)}\} \times (e_2 \triangleright_p e_1)) & (6)\\
e_1 \bowtie\!\Gamma_{p;g:f}\, e_2 & := & \{r \circ [g : G] | r \in e_1, & \\
 & & G = f(\{s | s \in e_2, p(r,s)\})\} & (7)
\end{array}
$$

e_1

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

e_2

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

$e_1 \bowtie\!\Gamma_{e_1.a=e_2.f;g:sum(e_2.f)}\, e_2$

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

Figure 1: Join Operators and Example for Groupjoin

aggregation results. We denote by $\Gamma_{G;A:F}$ the group-by of an expression e with grouping attributes G, aggregation vector F and aggregation attributes A.

The join operators we consider are the (inner) join ($\bowtie$), left semijoin $\ltimes$, left antijoin ($\triangleright$), left outerjoin (⟕), full outerjoin (⟗), and groupjoin ($\bowtie\!\Gamma$). The definitions of these join operators are given in Figure 1. There, $\circ$ denotes tuple concatenation and $\perp_A$ denotes the attribute set A with all contained attributes set to null. Further, we denote by $\mathcal{A}(e)$ the set containing all attributes provided by e. Most of the shown operators are rather standard.

The last row defines the left groupjoin $\bowtie\!\Gamma$, introduced by von Bültzingsloewen [10]. First, for a given tuple $t_1 \in e_1$, it determines the sets of all join partners for t_1 in e_2 using 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 1 provides an example.

2.2 Pushing Group-By

There are several possible ways of applying a group-by before a join, depending on the type of join operator, the set of grouping attributes specified in the query, and the attributes that are to be aggregated. Here, we give only a rough overview of the reasoning behind these transformations. For more detail, the reader is referred to the work by Yan and Larson, who coined the term of eager/lazy aggregation for inner joins [12, 13, 15, 14, 11], meaning pushing down/pulling up a group-by below/above a join in the operator tree. Thereby, they provided the basis for the work by Eich and Moerkotte [4], who extended the approach to non-inner joins and applied it in a state-of-the-art plan generator.

In general, when evaluating a query containing a group-by and one or several join operators, join arguments can be grouped before the join to reduce their cardinalities, thus making the join itself cheaper. However, this transformation typically has an influence on the query result, meaning that one further operation after the evaluation of the final join is needed to achieve the desired result. In many cases, this additional operation consists of a final group-by at

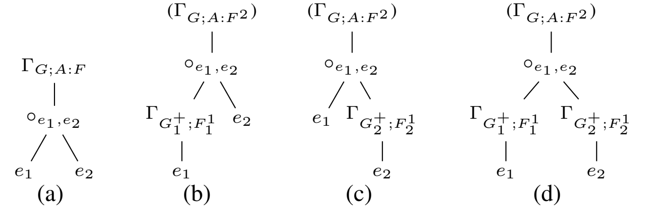

Figure 2: Possible trees for group-by and join

the top of the operator tree to fix the query result. In other cases, we only need to apply the aggregate functions specified by the user, since the query result is already grouped as desired. The latter case can be identified by the fact that all attributes contained in the query result are functionally determined by the grouping attributes specified in the input query, rendering an additional group-by obsolete. This is illustrated by the following implication, where we denote by $|e|$ the cardinality of the relation defined by expression e:

$$G \to \mathcal{A}(e) \Rightarrow |\Gamma_{G;A:F}(e)| = |e|.$$

In other words: if the grouping attributes form a key of the relation that is grouped, the group-by has no effect besides the application of the aggregation vector F.

Figure 2 provides an overview of the different possible evaluation strategies for a query containing a join of expressions e_1 and e_2 and a group-by with grouping attributes G and aggregation vector F. The first tree represents the conventional approach of evaluating the group-by after the join. The remaining trees show the different possibilities of grouping one or both join arguments before the join. In these cases, the aggregation vector typically needs to be decomposed and split, i.e., it has to be computed in two steps before and after the join to achieve the correct end result, which is indciated by the subscripts and superscripts of the aggregation vector applied at the respective group-by below the join. We put the final group-by in brackets to indicate that it may not be needed. We denote by $\circ_{e_1,e_2}$ an arbitrary join operator with a join predicate referencing e_1 and e_2. We also assume that the join operator $\circ$ allows pushing down a group-by into both of its arguments, which is not always the case. It is important to choose the correct set of grouping attributes for the "eager" group-by operators that are performed before the join. We denote by G_1^+/G_2^+ the sets of grouping attributes to be applied on top of e_1/e_2, respectively. They contain all attributes that are "still needed" after the group-by, i.e., all attributes provided by e_1/e_2 that are referenced in a predicate further up in the tree or as part of the grouping attributes specified in the query. That is because the group-by removes all attributes not contained in its set of grouping attributes, thereby possibly rendering the evaluation of subsequent predicates or groupings impossible. For more detailed information regarding all issues discussed in this paragraph, the reader is referred to the previous work on the topic ([12, 13, 14, 11, 4]).

2.3 Dynamic Programming

We briefly repeat the basics of a plan generator based on dynamic programming (DP). The plan generator is then extended to allow the reordering of joins and group-by operators. In this, we closely adhere to [4].

2.3.1 Plan Generation Basics

Figure 3 shows the basic structure of a typical DP-based plan generator. Its input consists of three major pieces: the set of re-

lations to be joined, the set of operators to be used for this, and a hypergraph representing the query graph. Clearly, the relations and the operators are derived from the initial SQL query in a straightforward manner. The hypergraph is constructed by a conflict detector [6]. It encodes possible reordering conflicts as far as possible into the hypergraph. This is necessary since inner joins and outer joins are not freely reorderable.

The major data structure used is the $DPTable$, which stores (an) optimal plan(s) for a given set of relations. The basic algorithm in Figure 3 uses a single plan per $DPTable$ entry. Subsequently, we will see that for our purposes we have to store multiple plans per $DPTable$ entry.

The plan generator consists of four major components. The first component initializes the $DPTable$ with plans for access paths for single relations, such as table scans and index accesses (Line 1,2). The second one enumerates connected-subgraph-complement-pairs (ccp for short) of the hypergraph H (Line 3), where a ccp is defined as follows:

DEFINITION 1. *Let $H = (V, E)$ be a hypergraph and S_1, S_2 two subsets of V. (S_1, S_2) is a* ccp *if the following three conditions hold:*

1. $S_1 \cap S_2 = \emptyset$,
2. *S_1 and S_2 induce connected subgraphs of H, and*
3. *$\exists (u, v) \in E,\ u \subseteq S_1 \wedge v \subseteq S_2$, that is S_1 and S_2 are connected by some edge.*

An efficient enumerator for ccps has been proposed in [7].

The third component (Line 5) is an applicability test for operators. It builds upon the conflict representation and checks whether some operator $\circ_p$ can be safely applied. This is necessary since it is not possible to exactly cover all reordering conflicts within a hypergraph representation of the query [6].

The fourth component (BUILDPLANS) is a procedure that builds plans using some operator $\circ_p$ as the top operator and the optimal plans for the subsets of relations S_1 and S_2, which can be looked up in the $DPTable$. Finally, the optimal plan is returned (Line 9).

This basic algorithm can be extended in such a way that it can reorder not only join operators but also join and group-by operators.

2.3.2 Extending the Plan Generator

To this end, the routine OPTREES is introduced (Figure 4). Its arguments are two join trees T_1 and T_2, and a join operator $\circ_p$. The result consists of a set of at most four trees which join T_1 and T_2, including all possible variants of eager aggregation.

The relation sets S_1 and S_2 are obtained from T_1 and T_2, respectively, by extracting their leaf nodes. The first tree is the one which joins T_1 and T_2 using $\circ_p$ without any grouping.

One situation that requires some care is when a join tree containing all the relations in our query is created. That is, $S = R$ holds, where R is the set of all relations. In this case, we have to add another group-by on top of $\circ_p$ if and only if the grouping attributes do not comprise a (super-)key (see Section 2.2). This is checked by calling NEEDSGROUPING.

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, which is achieved by a call to the subroutine VALID. 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 group-by would be a waste. And again, if necessary, we have to add a group-by on top.

```
DP-PLANGEN
  // Input: a set of relations R = {R_0, ..., R_{n-1}}
            a set of operators O with associated predicates
            a query hypergraph 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 csg-cmp-pairs (S_1, S_2) of H
4     for all ∘_p ∈ O
5         if APPLICABLE(S_1, S_2, ∘_p)
6             BUILDPLANS(S_1, S_2, ∘_p)
7             if ∘_p is commutative
8                 BUILDPLANS(S_2, S_1, ∘_p)
9 return DPTable[R]

BUILDPLANS(S_1, S_2, ∘_p)
1 OptimalCost = ∞
2 S = S_1 ∪ S_2
3 T_1 = DPTable[S_1]
4 T_2 = DPTable[S_2]
5 if DPTable[S] ≠ NULL
6     OptimalCost = COST(DPTable[S])
7 if COST(T_1 ∘_p T_2) < OptimalCost
8     OptimalCost = COST(T_1 ∘_p T_2)
9     DPTable[S] = (T_1 ∘_p T_2)
```

Figure 3: Basic DP Algorithm

Once the routine terminates, the returned set *Trees* contains up to four different join trees, as depicted further up in Figure 2.

To find the best possible join tree taking eager aggregation into account, we have to keep all subtrees found by our plan generator, combine them to produce all possible trees for our query and pick the best one. That is, we cannot just keep the cheapest plan for each plan class, as is typically the case when only reordering join operators.

This is because Bellman's principle of optimality, which is needed to make DP applicable, does no longer hold once eager aggregation is taken into consideration. The reason for this is the fact that applying a group-by at an arbitrary point in the operator tree influences the functional dependencies holding in all the subsequent intermediate results. These dependencies finally determine whether or not we need a final group-by on top to fix the query result (see Section 2.2). The final group-by causes an additional cost that can destroy the optimality of the plan. Consequently, we also have to keep the more expensive subplans for each intermediate result because they might turn out to be a part of the optimal solution in the end.

To achieve this, the dynamic programming table is modified to contain not only one optimal join tree for every set $S \subseteq R$, but a list of possible trees. Figure 5 shows the routine BUILDPLANSALL, which is derived from the routine BUILDPLANS depicted in Figure 3 and illustrates the necessary modifications.

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. We call OPTREES 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 list for S.

Eventually, we face the situation where $S = R$ holds and we

```
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 = (Γ_G(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 = (Γ_G(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 = (Γ_G(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 = (Γ_G(NewTree))
23     Trees.insert(NewTree)
24 return Trees
```

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

Figure 4: OpTrees **and** NeedsGrouping

need to build a join tree for the complete query. At this point, we call another subroutine named InsertTopLevelPlan. Inside this routine, we compare the join trees for S to find the one with minimal costs because there are no subsequent join operators that need to be taken into account. Before we can do this, we have to decide whether we need a top-level group-by by calling NeedsGrouping (Figure 4). In contrast to the other relation sets, we do not have to keep a list of trees for R, but only the best tree found so far and replace it if a better one is found.

The runtime complexity of this algorithm is $O(2^{2n-1}\#\text{ccp})$ for n relations if #ccp denotes the number of ccps for the query (see Definition 1).

2.3.3 *Optimality Preserving Pruning*

As we have seen in the previous section, keeping all possible trees in the solution table guarantees an optimal solution but, on the other hand, 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 and still preserve the optimality of the resulting solution. In other words, we are looking for an effective optimality-preserving pruning criterion.

Figure 6 shows the routine PruneDominatedPlans, which discards all trees that are *dominated* by any other tree found so far.

```
BuildPlansAll(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].Append(T)
```

```
InsertTopLevelPlan(S, T)
1 if DPTable[S] == ∅ ∨ Cost(T) < Cost(DPTable[S])
2     DPTable[S] = ∅
3     DPTable.Append(T)
```

Figure 5: BuildPlansAll

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

Figure 6: PruneDominatedPlans

The routine expects as arguments a set of relations S and a join tree T for this set. It is called from inside BuildPlansAll. To this end, we replace line 8 in BuildPlansAll by

$$\text{PruneDominatedPlans}(S, T).$$

The loop beginning in line 1 of PruneDominatedPlans iterates 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 list 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 existing trees because more dominated trees to discard may exist. Eventually, the loop ends and T is added to the list for S.

In the rest of this paper, we discuss several notions of dominance and evaluate them with respect to their effectiveness as a pruning criterion.

3. INTERESTING PLAN PROPERTIES AND THEIR DERIVATION

In this section we provide rules for computing interesting properties of query plans that we use during plan generation.

3.1 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. Note also that the keys resulting from the full and left outerjoin contain null values. We therefore assume

that null values are treated as suggested in [9], i.e., two attributes are equal if they agree in value or they are both null. We assume that we know the keys of the base relations from the database schema.

Rules for computing the keys are taken from [4].

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 [9]. Initially, FDs for a base relation are deduced from the keys declared in the database schema. In the rest of this work, we frequently use the closure of a given set of FDs, denoted by FD^{+1}.

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. 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: $EC[A] = c \in EC, A \in c$.

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.

3.2 Deriving Interesting Properties

We provide rules for computing the three sets bottom-up in an operator tree possibly containing all algebraic operators covered in Section 2.1.

The rules concerning EC and FD are taken from Paulley [9]. For simplicity, we make some restrictions on the join predicates we consider. We assume (possibly) conjunctive predicates with each conjunct referencing exactly two relations.

One concept useful for the computation of these abovementioned properties is *null-rejection* of a predicate p on attribute A. It is defined as follows [5]:

DEFINITION 2. *A predicate p rejects nulls on attribute A if it does not evaluate to true if A is null:*

$$p[A \in null] \neq true.$$

$NR(p)$ is the set of attributes, on which predicate p rejects nulls.

3.2.1 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 [4]:

- 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
$$\kappa(e_1 \bowtie_q e_2) = \bigcup_{k_1 \in \kappa(e_1), k_2 \in \kappa(e_2)} k_1 \cup k_2.$$
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: $NN(e_1 \bowtie_p e_2) = NN(e_1) \cup NN(e_2) \cup NR(p)$.

3.2.2 Left Outerjoin

Consider the left outerjoin of expressions e_1 and e_2: $e_1 ⟕_p e_2$. 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 $\kappa(e_1 ⟕_{A_1=A_2} e_2) = \kappa(e_1)$.

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

$$\kappa(e_1 ⟕_q e_2) = \bigcup_{k_1 \in \kappa(e_1), k_2 \in \kappa(e_2)} k_1 \cup k_2,$$

where q 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.

[1] By closure we mean the set of all dependencies derivable from a given set of dependencies as the term is commonly understood.

This gives rise to the following equation, where p is an arbitrary predicate:

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

In the case of an equality predicate, we do not get a new equivalence class, as 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: $e_1 ⟕_{A_1=A_2} e_2$. In this case, the following equation holds:

$$FD(e_1 ⟕_{A_1=A_2} e_2) = FD^+(e_1) \cup \{(\alpha \to \beta) \in FD^+(e_2) \mid (\alpha \cap (NN(e_2) \cup NR(p)) \neq \emptyset)\} \cup \{A_1 \to A_2\}.$$

Equality Constraints. Equality constraints from both join arguments continue to hold in the join result, resulting in the following equation: $EC(e_1 ⟕_p e_2) = EC(e_1) \cup EC(e_2)$.

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: $NN(e_1 ⟕_p e_2) = NN(e_1)$.

3.2.3 Full Outerjoin

Consider the full outerjoin of expressions e_1 and e_2: $e_1 ⟗_p e_2$.

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

$$\kappa(e_1 ⟗_p e_2) = \bigcup_{k_1 \in \kappa(e_1), k_2 \in \kappa(e_2)} k_1 \cup k_2,$$

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/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.

$$FD(e_1 ⟗_{A_1=A_2} e_2) = \{(\alpha \to \beta) \in FD^+(e_1) \mid (\alpha \cap NN(e_1) \neq \emptyset) \vee (p \text{ is null-rejecting in } \mathcal{F}(p) \cap \mathcal{A}(e_1))\} \cup \{(\alpha \to \beta) \in FD^+(e_2) \mid (\alpha \cap NN(e_2) \neq \emptyset) \vee (p \text{ is null-rejecting in } \mathcal{F}(p) \cap \mathcal{A}(e_2))\}.$$

Here, we denote by $\mathcal{F}(p)$ the set of attributes occurring freely in predicate p.

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: $EC(e_1 ⟗_p e_2) = EC(e_1) \cup EC(e_2)$.

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: $NN(e_1 ⟗_p e_2) = \emptyset$.

3.2.4 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 ($e_1 ⟕\Gamma_p e_2$) of expression e_1 and e_2.

Keys. Since the attributes from the right input are no longer available in the join result and the result is duplicate-free by definition, we always have $\kappa(e_1 \circ e_2) = \kappa(e_1)$, for $\circ \in \{\ltimes, \triangleright, ⟕\Gamma_{G;A:F}\}$.

Functional Dependencies. Each of the abovementioned operators produces a relation containing only attributes from e_1. Therefore, all FDs referencing attributes from e_2 are no longer valid after the join. The ones from e_1 continue to hold. We can therefore state the following equations: $FD(e_1 \circ_p e_2) = FD^+(e_1)$, for $\circ \in \{\ltimes, \triangleright\}$.

In the left groupjoin, the attributes in G determine the ones in A: $FD(e_1 ⟕\Gamma_{p;G;A:F} e_2) = FD^+(e_1) \cup \{G \to A\}$.

Equality Constraints. Equality constraints holding in the left input (e_1 in our example) continue to hold in the join result. Since no attributes from the right input are contained in the join result, the equivalence classes from that input are no longer valid after the join: $EC(e_1 \circ_p e_2) = EC(e_1)$, for $\circ \in \{\ltimes, \triangleright, ⟕\Gamma\}$.

Definite Attributes. None of the abovementioned join operators introduces null values. Therefore, the definite attributes of the join result are the ones from e_1: $NN(e_1 \circ_p e_2) = NN(e_1)$, for $\circ \in \{\ltimes, \triangleright\}$.

In the left groupjoin, an attribute $a \in A$ is definite if the aggregation function it results from does not return null. This depends on whether or not the argument of the aggregation function is definite and on the characteristics of the aggregation 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: $NN(e_1 ⟕\Gamma_{p;G;A:F} e_2) = NN(e_1) \cup A$.

3.2.5 Group-By

The result of a group-by applied to some 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. Consider a group-by applied to expression e: $\Gamma_{G;A:F}(e)$. The grouping attributes G can be a superkey of the group-by's argument e. In this case all keys contained in G remain keys after applying the group-by: $\kappa(\Gamma_{G;A:F}(e)) = \{k \in \kappa(e) | k \subset G\}$.

Otherwise, the 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 group-by, all FDs referring only to the grouping attributes or a subset thereof remain valid. That is, we keep those dependencies where both sides are contained in the grouping attributes. Additionally, the grouping attributes functionally 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 group-by. $EC(\Gamma_G(e)) = \{c \cap G \mid c \in EC(e), c \cap G \neq \emptyset\}$

Definite Attributes. A group-by does not introduce new null values. The aggregation results in attribute set A may be definite

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 7: ATTRIBUTECLOSURE

under the same conditions as for the groupjoin. In this case, the following holds: $NN(\Gamma_{G;A:F}(e)) = (NN(e) \cap G) \cup A$.

3.3 Computing the Attribute Closure

During plan generation, we are interested in the *attribute closure* of a set of attributes α, denoted 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 β' to β 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 $\beta' \to \gamma$ which gives the transitive dependency $\alpha \to \gamma$. In this case, we add γ to the right-hand side of our original dependency and repeat the whole process until there are no more changes.

The pseudocode for ATTRIBUTECLOSURE is given in Figure 7. 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 to be computed.

3.4 Implementation in a Plan Generator

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 [2]. 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 occuring 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 7.

4. PRUNING WITH FUNCTIONAL DEPENDENCIES

The correctness proofs for all pruning criteria discussed in this paper can be found in [3]. First, we define f-dominance [4]:

DEFINITION 3. *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. $FD^+(T_1) \supseteq FD^+(T_2)$.

It is important to note that the compared trees are not necessarily equivalent due to the contained group-by operaions. As discussed in Section 2, a group-by on top of the final join may be necessary to compensate this.

In order to avoid the overhead associated with computing FD^+, which is used to define f-dominance, the plan generator described in [4] applies the following pruning criterion, which we call *k-dominance* because it is based on keys:

DEFINITION 4. *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. $\kappa(T_1) \supseteq \kappa(T_2)$.

With the following example, we show that there are cases where one tree f-dominates but does not k-dominate another tree. In such cases it can be beneficial to use FDs instead of keys to better exploit the pruning potential.

Figure 8 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 NK_i, with $i \in (0, \ldots, 3)$. In addition to the operators, the trees shown in Figure 8 contain special nodes displaying the keys valid at the respective point in the tree according to the key computation rules from Section 3. We assign numbers to the operators to make them easier to identify.

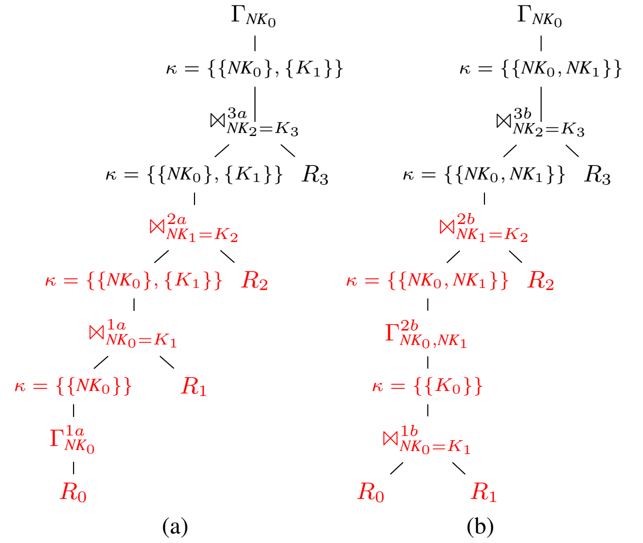

Figure 8: Two Operator Trees with Keys

Assume that during plan generation we compare the subtrees for the relation set $\{R_0, R_1, R_2\}$ marked red in the figure 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. 4). Assume further that the tree on the right has lower cost than the one on the left and equal cardinality. Therefore, the only criterion 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, this criterion is not met, and we decide to keep the more expensive subtree. We will now use f-dominance as the pruning criterion.

Table 1 shows the FDs and equivalence classes for each intermediate result of the join trees depicted in Figure 8. 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 3.

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 group-by on top of R_0 is applied in Figure 8(a), 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 NK_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 NK_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 $\{\{NK_0, K_1\}\} \rightarrow \{\{NK_0, K_1\}, NK_1\}$ subsumes the following dependencies:

$$\begin{aligned} \{NK_0\} &\rightarrow \{NK_0, K_1, NK_1\}, \\ \{K_1\} &\rightarrow \{NK_0, K_1, NK_1\}. \end{aligned}$$

Applying the closure computation algorithm from Section 3 and replacing attributes by their equivalence classes yields the dependencies and equivalence classes shown in the table.

We can now return to our original problem: can we discard the more expensive tree from Figure 8(a) in favor of the one in Figure 8(b) by considering the FDs holding in both trees instead of the keys? That is, we need to check if the following relationship holds:

$$FD^+(\bowtie^{2a}) \subseteq FD^+(\bowtie^{2b}). \qquad (8)$$

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:

$$\{\{NK_0, K_1\}\} \rightarrow \{\{NK_0, K_1\}, \{NK_1, K_2\}, NK_2\}$$

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

$$\{NK_0\} \rightarrow \{NK_0, \{NK_1, K_2\}, NK_2\}.$$

In our example, we find a match for every dependency from the left side of the table leading us to the conclusion that Eq. 8 holds. We can therefore safely discard the more expensive tree.

Taking a closer look at Table 1, we also see that $\kappa(\bowtie^{2b}) = \{\{NK_0\}\}$, since all attributes present in the tree are determined by NK_0. The key resulting from the key computation shown in Figure 8 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 this does not hold, especially in the presence of non-inner joins. We present an example in Figure 9.

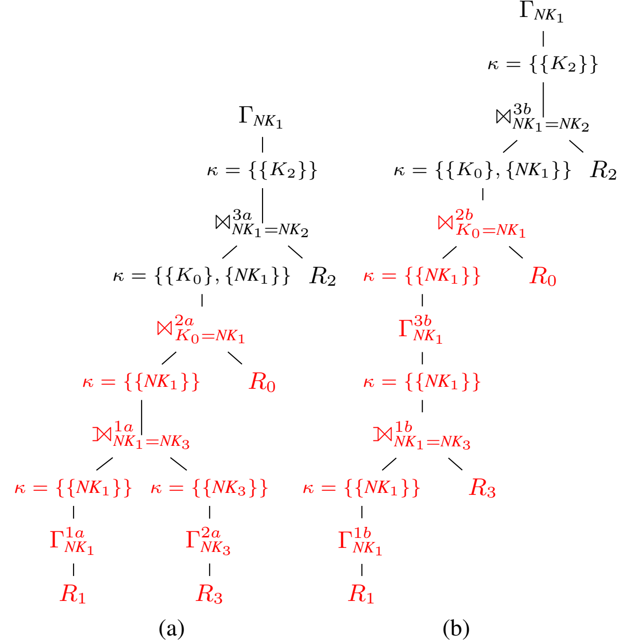

Figure 9: Two Operator Trees with Keys

We assume the same relation schemas as in the previous example, and we are again interested in discarding the tree in Figure 9(a) because it is more expensive with equal cardinality as the one in Figure 9(b). 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 4.

On the other hand, the requirements for f-dominance are not fulfilled, as can be seen in Table 2, which contains the FDs and equality constraints up to $\bowtie^2$, the root of the two subtrees we are comparing.

The dependency $\{\{K_0, NK_1\}\} \rightarrow \{NK_3\}$, which is contained in $AC^+(\bowtie^{2a})$, is not contained in $AC^+(\bowtie^{2b})$. This is because attribute NK_3 is not available in the latter, since it is removed by Γ^3. To see that this is caused by the left outerjoin $⟕^1$, we replace it by an inner join.

This results in an equivalence class $\{NK_1, NK_3\}$, which is later extended to $\{K_0, NK_1, NK_3\}$, turning the problematic dependency from above into $\{\{K_0, NK_1, NK_3\}\} \rightarrow \{\{K_0, NK_1, NK_3\}\}$. Since we only need to find one attribute from each equivalence class on the correct side of another dependency, the dependency $\{\{K_0, NK_1\}\} \rightarrow \{\{K_0, NK_1\}\}$ holding in $\bowtie^{2b}$ satisfies the conditions for f-dominance.

5. PRUNING WITH RESTRICTED KEYS

Using f-dominance instead of k-dominance is often beneficial in terms of better pruning possibilities. But computing and comparing the needed properties is more expensive. In this section, we propose a third pruning approach that makes use of the keys and

Table 1: Functional Dependencies for Figure 8

	Figure 8(a)		Figure 8(b)	
	AC⁺	**EC**	**AC⁺**	**EC**
R_0	$\{K_0\} \rightarrow \{K_0, NK_0\}$	$\emptyset$	$\{K_0\} \rightarrow \{K_0, NK_0\}$	$\emptyset$
R_1	$\{K_1\} \rightarrow \{K_1, NK_1\}$	$\emptyset$	$\{K_1\} \rightarrow \{K_1, NK_1\}$	$\emptyset$
R_2	$\{K_2\} \rightarrow \{K_2, NK_2\}$	$\emptyset$	$\{K_2\} \rightarrow \{K_2, NK_2\}$	$\emptyset$
R_3	$\{K_3\} \rightarrow \{K_3, NK_3\}$	$\emptyset$	$\{K_3\} \rightarrow \{K_3, NK_3\}$	$\emptyset$
Γ^1	$\{NK_0\} \rightarrow \{NK_0\}$	$\emptyset$	-	$\emptyset$
$\bowtie^1$	$\{\{NK_0, K_1\}\} \rightarrow \{\{NK_0, K_1\}, NK_1\}$	$\{NK_0, K_1\}$	$\{\{NK_0, K_1\}\} \rightarrow \{\{NK_0, K_1\}, NK_1\}$	$\{NK_0, K_1\}$
Γ^2	-	$\{NK_0, K_1\}$	$\{NK_0, NK_1\} \rightarrow \{NK_0, NK_1\}$ $\{NK_0\} \rightarrow \{NK_0, NK_1\}$	$\emptyset$
$\bowtie^2$	$\{\{NK_0, K_1\}\} \rightarrow \{\{NK_0, K_1\}, \{NK_1, K_2\}, NK_2\}$ $\{\{NK_1, K_2\}\} \rightarrow \{\{NK_1, K_2\}, NK_2\}$	$\{NK_0, K_1\}$ $\{NK_1, K_2\}$	$\{NK_0\} \rightarrow \{NK_0, \{NK_1, K_2\}, NK_2\}$ $\{NK_0, NK_1\} \rightarrow \{NK_0, \{NK_1, K_2\}, NK_2\}$ $\{\{NK_1, K_2\}\} \rightarrow \{\{Nk_1, K_2\}, NK_2\}$	$\{NK_1, K_2\}$
$\bowtie^3$	$\{\{NK_0, K_1\}\} \rightarrow \{\{NK_0, K_1\}, \{NK_1, K_2\}, \{NK_2, K_3\}, NK_3\}$ $\{\{K_2, NK_1\}\} \rightarrow \{\{NK_1, K_2\}, \{NK_2, K_3\}, NK_3\}$ $\{\{K_3, NK_2\}\} \rightarrow \{\{NK_2, K_3\}, NK_3\}$	$\{NK_0, K_1\}$ $\{NK_1, K_2\}$ $\{NK_2, K_3\}$	$\{NK_0\} \rightarrow \{NK_0, \{NK_1, K_2\}, \{NK_2, K_3\}, NK_3\}$ $\{NK_0, NK_1\} \rightarrow \{NK_0, \{NK_1, K_2\}, \{NK_2, K_3\}, NK_3\}$ $\{\{NK_1, K_2\}\} \rightarrow \{\{NK_1, K_2\}, \{NK_2, K_3\}, NK_3\}$ $\{\{NK_2, K_3\}\} \rightarrow \{\{NK_2, K_3\}, NK_3\}$	$\{NK_1, K_2\}$ $\{NK_2, K_3\}$

Table 2: Functional Dependencies for Figure 9

	Figure 9(a)		Figure 9(b)	
	AC⁺	**EC**	**AC⁺**	**EC**
R_0	$\{K_0\} \rightarrow \{K_0, NK_0\}$	$\emptyset$	$\{K_0\} \rightarrow \{K_0, NK_0\}$	$\emptyset$
R_1	$\{K_1\} \rightarrow \{K_1, NK_1\}$	$\emptyset$	$\{K_1\} \rightarrow \{K_1, NK_1\}$	$\emptyset$
R_3	$\{K_3\} \rightarrow \{K_3, NK_3\}$	$\emptyset$	$\{K_3\} \rightarrow \{K_3, NK_3\}$	$\emptyset$
Γ^1	$\{NK_1\} \rightarrow \{NK_1\}$	$\emptyset$	$\{NK_1\} \rightarrow \{NK_1\}$	$\emptyset$
Γ^2	$\{NK_3\} \rightarrow \{NK_3\}$	$\emptyset$	-	$\emptyset$
$\bowtie^1$	$\{NK_1\} \rightarrow \{NK_1, NK_3\}$ $\{NK_3\} \rightarrow \{NK_3\}$	$\emptyset$	$\{NK_1\} \rightarrow \{NK_1, NK_3\}$ $\{K_3\} \rightarrow \{K_3, NK_3\}$	$\emptyset$
Γ^3	-	$\emptyset$	$\{NK_1\} \rightarrow \{NK_1\}$	$\emptyset$
$\bowtie^2$	$\{\{K_0, NK_1\}\} \rightarrow \{\{K_0, NK_1\}, NK_0, NK_3\}$ $\{NK_3\} \rightarrow \{NK_3\}$	$\{K_0, NK_1\}$	$\{\{K_0, NK_1\}\} \rightarrow \{\{K_0, NK_1\}, NK_0\}$	$\{K_0, NK_1\}$

at the same time allows effective pruning. Again, we provide an example consisting of two alternative join trees for the same query. They are shown in Figure 10.

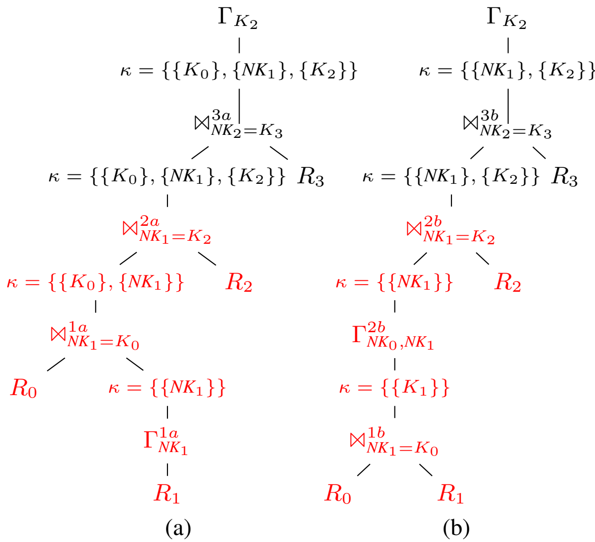

Figure 10: Two Operator Trees with Keys

As before, we are comparing the subtrees marked red, and we are interested in discarding the subtree in Figure 10(a), assuming that it is more expensive than its counterpart on the right side and 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 group-by 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. Therefore, it does not influence the key constraints that hold in the following intermediate results which in turn determine the necessity of the final group-by. The same argument implies that we can also ignore $\{NK_1\}$. Applying this to 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:

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

We can now define *rk-dominance*:

DEFINITION 5. *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. *$\kappa^-(T_1) \supseteq \kappa^-(T_2)$.*

6. PRUNING WITH RESTRICTED FDS

So far we have observed that in general we can prune more subplans with FDs than with keys and even more with restricted keys. Using a *restricted* set of FDs promises to further increase the pruning capabilities of our plan generator.

Again, we start by giving an example consisting of two operator trees as shown in Figure 11, where we assume that the red subtree in Figure 11(a) is more expensive than the one in Figure 11(b).

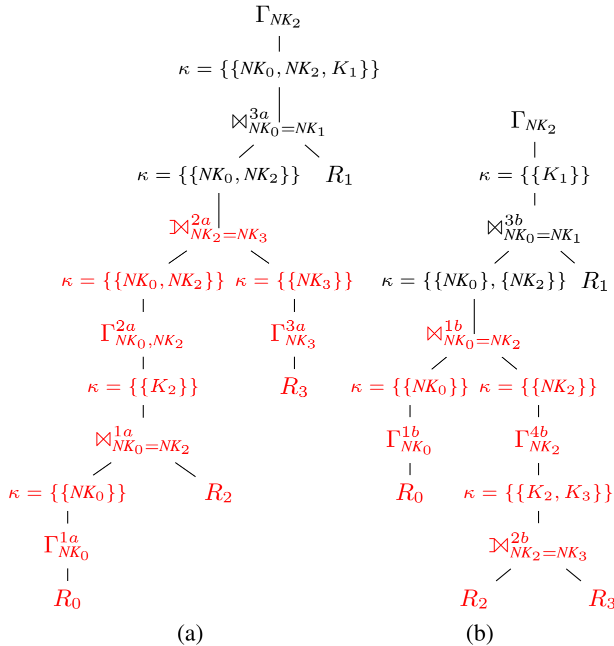

Figure 11: Two Operator Trees with Keys

Table 3 contains the FDs and equality constraints holding in each intermediate result up to the root nodes of the two subtrees. From the information in the table and the key sets given in the figure, we conclude that none of the different notions of dominance we have considered so far suffice to discard the left tree in this case, because their requirements are not fulfilled.

The third requirement for f-dominance is violated because the dependency $\{\{NK_0, NK_2\}\} \rightarrow \{NK_3\}$ is contained in $AC^+(\bowtie^{2a})$, but not in $AC^+(\bowtie^{1b})$.

The attribute NK_3 is not referenced in any of the join predicates above the root nodes of the two subtrees. It is also not part of the grouping attributes at the top-most grouping operator. Removing it from the right-hand side of the dependency in question allows us to discard the subtree from Figure 11(a). In analogy to the restricted key set κ^-, we define the restricted set of FDs 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 6. *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. *$FD^-(T_1) \supseteq FD^-(T_2)$.*

7. PRUNING WITH KEYS AND 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 to prune 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 which maximizes the pruning capabilities of the plan generator and at the same time minimizes the overhead associated with 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.

8. EVALUATION

We evaluate the performance of our pruning techniques with respect to runtime and memory consumption. By runtime we mean the time taken for plan generation, including cardinality estimation. We do not measure the execution time of the resulting plans since they are all optimal with respect to our cost model and therefore expected to have equal runtimes. We measure memory consumption as the number of entries in the DP-table after successful plan generation. This number also gives an impression of how effective the respective pruning criterion is.

We implemented all five approaches discussed in the previous sections in the plan generator DPHypE, which is based on the plan generator DPHyp described in [8] and capable of reordering non-inner joins and group-by operators. We denote by DPHypE$^{f/k/rk/rf/rkrf}$ the plan generator employing f-/k-/rk-/rf-/rkrf-dominance, respectively.

The workload consists of randomly generated join trees. Cardinalities for the base relations and selectivities for predicates are also generated randomly. A randomly chosen group-by is placed at the top of the tree. The results given in this section are average values of 10,000 queries for a given parameter value, e.g. the number of relations. All experiments were run on an Intel Xeon E5-2690 V2 CPU at 3.00 GHz.

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. The focus of our work lies not on join reordering, which has been thoroughly investigated in existing works and is in its

Table 3: Functional Dependencies for Figure 11

	Figure 11(a)		Figure 11(b)	
	AC+	**EC**	**AC+**	**EC**
R_0	$\{K_0\} \rightarrow \{K_0, NK_0\}$	$\emptyset$	$\{K_0\} \rightarrow \{K_0, NK_0\}$	$\emptyset$
R_2	$\{K_2\} \rightarrow \{K_2, NK_2\}$	$\emptyset$	$\{K_2\} \rightarrow \{K_2, NK_2\}$	$\emptyset$
R_3	$\{K_3\} \rightarrow \{K_3, NK_3\}$	$\emptyset$	$\{K_3\} \rightarrow \{K_3, NK_3\}$	$\emptyset$
Γ^1	$\{NK_0\} \rightarrow \{NK_0\}$	$\emptyset$	$\{NK_0\} \rightarrow \{NK_0\}$	$\emptyset$
$\bowtie^1$	$\{\{NK_0, NK_2\}\} \rightarrow \{\{NK_0, NK_2\}\}$ $\{K_2\} \rightarrow \{K_2, \{NK_0, NK_2\}\}$	$\{NK_0, NK_2\}$	$\{\{NK_0, NK_2\}\} \rightarrow \{\{NK_0, NK_2\}\}$	$\{NK_0, NK_2\}$
Γ^2	$\{\{NK_0, NK_2\}\} \rightarrow \{\{NK_0, NK_2\}\}$	$\{NK_0, NK_2\}$	-	-
Γ^3	$\{NK_3\} \rightarrow \{NK_3\}$	$\emptyset$	-	-
$⟕^2$	$\{\{NK_0, NK_2\}\} \rightarrow \{\{NK_0, NK_2\}, NK_3\}$ $\{NK_3\} \rightarrow \{NK_3\}$	$\{NK_0, NK_2\}$	$\{NK_2\} \rightarrow \{NK_2, NK_3\}$ $\{K_2\} \rightarrow \{K_2, NK_2, NK_3\}$ $\{K_3\} \rightarrow \{K_3, NK_3\}$	$\emptyset$
Γ^4	-	-	$\{NK_2\} \rightarrow \{NK_2\}$	$\emptyset$

complexity highly influenced by the query shape. Instead, we are interested in the complexity added by the optimization techniques discussed in this paper, which is strongly influenced by other factors, such as the number of foreign-key predicates and the presence of non-inner joins.

For an evaluation of the general effectiveness of reordering group-by and join operators the reader is referred to the previous work on the topic ([12, 13, 14, 11, 4]) containing experiments not only with a synthetic workload, but also with selected benchmark queries.

8.1 Runtime

Figures 12 and 13 compare the runtimes of the plan generators for queries with 5 to 15 relations. The runtimes shown in Figure 12 result from queries containing only inner joins, while Figure 13 depicts queries without any restrictions on the operator types. The search space for queries with only inner joins is larger than for queries containing non-inner joins, since inner joins are freely reorderable and a group-by can always be pushed into both join arguments. This is why the runtimes in Figure 12 are higher than the ones in Figure 13. Therefore, we did not run all plan generators up to 15 relations with inner joins only.

In both cases, we assume a proportion of foreign key predicates of eighty percent. The proportion of foreign key joins has an impact on the runtime of the plan generators, especially the ones dealing with key sets. Since a foreign key join often results in the propagation of only one key, the key sets are kept small, making the test of k-dominance and rk-dominance fast. We consider eighty percent a rather cautious assumption. We assume this number to be higher in most real queries. For a more detailed analysis of the impact that the proportion of foreign key predicates has on the runtime, the reader is referred to our technical report ([3]).

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 the number of relations. For queries with 15 relations and different join operators, DPHypE^{k} needs 1.4 seconds on average, while DPHypE^{rkrf} only runs 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, DPHypE^{rk} and DPHypE^{rkrf} are faster than DPHypE^{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

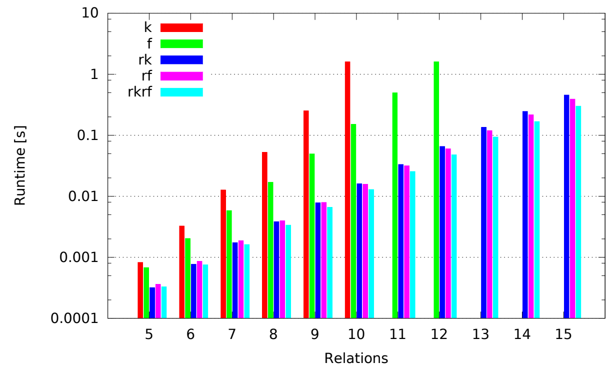

Figure 12: Runtimes for 5 to 15 relations, only inner joins

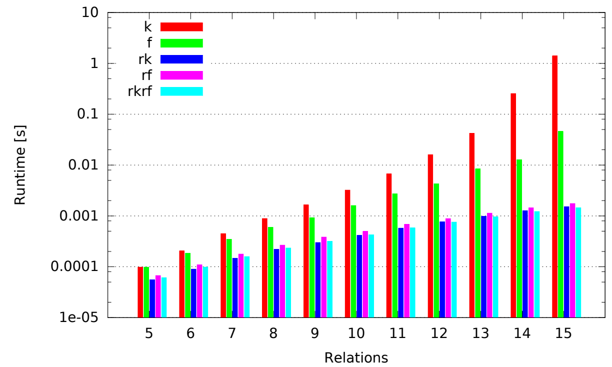

Figure 13: Runtimes for 5 to 15 relations, all join operators

15 relations and arbitrary join operators, we got the following numbers for "runtime 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 with inner joins only, we observe some differences. As we can see in Figure 12, there is a trend for larger queries: since the search space is so large with inner joins only, the search space restriction achieved by the pruning criterion becomes more critical, causing rk-dominance to become less and less

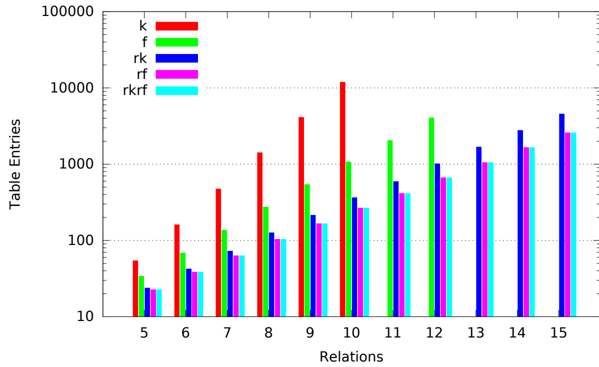

Figure 14: Table Entries for 5 to 15 relations, only inner joins

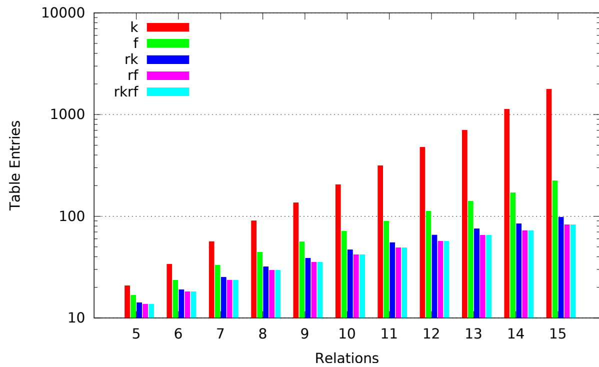

Figure 15: Table Entries for 5 to 15 relations, all join operators

efficient when compared to rf-dominance and rkrf-dominance.

8.2 Memory Usage

The reasons for the runtime differences between DPhypE^{k} and the rest become obvious when we look at the dp-table entries produced by the different algorithms as depicted in Figures 14 and 15. 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.

If we allow different join operators, the average number of table entries is 1800 / 82 for DPHypE^{k} / DPHypE^{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, we have 12,000 / 270 table entries on average for the same two plan generators and queries with 10 relations.

9. CONCLUSION

We presented a set of pruning criteria applicable in a bottom-up plan generator reordering (outer)joins and group-by. To this end, we first analyzed the plan properties needed for effective pruning and how they can be derived during plan generation.

We then showed that using functional dependencies instead of keys, as proposed in [4], reveals much better pruning opportunities. Restricting the set of functional dependencies and keys to contain only the information crucial to guarantee an optimal solution makes the resulting pruning criteria even more effective.

All this led to a speed up factor of up to several orders of magnitude when compared to the only existing approach, rendering non-optimal heuristics obsolete even for larger queries.

Acknowledgment. We thank Simone Seeger for her help preparing the mansucript and the anonymous reviewers for their helpful comments.

10. REFERENCES

[1] S. Chaudhuri and K. Shim. Including group-by in query optimization. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 354–366, 1994.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[3] M. Eich, P. Fender, and G. Moerkotte. Faster plan generation through consideration of functional dependencies and keys. Technical report, University of Mannheim, 2016.

[4] M. Eich and G. Moerkotte. Dynamic programming: The next step. In *Proc. IEEE Conference on Data Engineering*, pages 903–914, 2015.

[5] C. Galindo-Legaria and A. Rosenthal. Outerjoin simplification and reordering for query optimization. *ACM Trans. Database Syst.*, 22(1):43–74, Mar. 1997.

[6] G. Moerkotte, P. Fender, and M. 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.

[7] G. Moerkotte and T. Neumann. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 930–941, 2006.

[8] G. Moerkotte and T. Neumann. Dynamic programming strikes back. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 539–552, 2008.

[9] G. Paulley. *Exploiting Functional Dependence in Query Optimization*. PhD thesis, University of Waterloo, 2000.

[10] G. von Bultzingsloewen. Optimizing sql queries for parallel execution. *SIGMOD Rec.*, 18, December 1989.

[11] W. Yan. *Rewriting Optimization of SQL Queries Containing GROUP-BY*. PhD thesis, University of Waterloo, 1995.

[12] 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.

[13] W. Yan and P.-A. Larson. Performing group-by before join. In *Proc. IEEE Conference on Data Engineering*, pages 89–100, 1994.

[14] W. Yan and P.-A. Larson. Eager aggregation and lazy aggregation. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 345–357, 1995.

[15] 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.