

Efficient generation of query plans containing group-by, join, and groupjoin

Marius Eich¹  · Pit Fender² · Guido Moerkotte¹

Received: 13 January 2017 / Revised: 27 July 2017 / Accepted: 2 August 2017
© Springer-Verlag GmbH Germany 2017

Abstract It has been a recognized fact for many years that query execution can benefit from pushing grouping 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, the 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 extend the idea of eager aggregation further by combining it with the introduction of groupjoins. We evaluate the resulting plan generator with respect to runtime and memory consumption.

Keywords Aggregation · Join ordering · Dynamic programming · Query optimization

1 Introduction

The idea of reordering grouping operators and joins was suggested more than two decades ago [2, 16–19]. However, it was always limited to inner joins only. In previous work

[8], we revived the topic by showing that the optimal placement of grouping operators is possible in the presence of non-inner joins as well, thus enabling the application of this technique to a whole new class of queries. Aside from the required algebraic equivalences, we also proposed a plan generator capable of reordering grouping and a wide range of different join operators. Figure 1 shows an example query against the TPC-H schema. We measured the runtime of this query at a TPC-H scale factor of one on a disk-based and a main-memory system. The results were 54,797 ms on the disk-based system and 7974 ms on the main-memory system. By rewriting the query such that the arguments of the full outer join are grouped before the join, we reduced the runtime to 55 ms on the disk-based system and 16 ms on the main-memory system. This execution order corresponds to the plan produced by our plan generator. The rewritten query is shown in Fig. 2.

While the aforementioned plan generator performs well for small queries, queries with more than ten relations can only be handled by abandoning optimality and relying on heuristics. The reason for this limitation is the lack of an effective optimality-preserving pruning criterion limiting the size of the search space and thereby allowing 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 grouping to each of these edges and on top of the root, resulting in $2n - 1$ possible positions 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.

✉ Marius Eich
mariaus.eich@uni-mannheim.de

¹ School of Business Informatics and Mathematics, University of Mannheim, Mannheim, Germany

² Oracle Labs, Redwood Shores, CA, USA

```

select s_nationkey, c_nationkey,
       count(distinct s_suppkey),
       count(distinct c_custkey)
from
  supplier full outer join customer
  on s_nationkey = c_nationkey
group by s_nationkey, c_nationkey;

```

Fig. 1 Query containing full outer join and group-by

```

select s_nationkey, c_nationkey,
       scnt, ccnt
from
  (select s_nationkey,
         count(distinct s_suppkey) as scnt
   from supplier
   group by s_nationkey) as s
full outer join
  (select c_nationkey,
         count(distinct c_custkey) as ccnt
   from customer
   group by c_nationkey) as c
on s.s_nationkey = c.c_nationkey;

```

Fig. 2 Rewritten query with pushed-down grouping

Recently, we gave an in-depth analysis of five optimality-preserving pruning criteria [5]. They were derived by a careful investigation of keys and functional dependencies. The experimental evaluation of the different pruning approaches showed that they can speed up the plan generator by orders of magnitude.

In this paper, we provide some additional information concerning the aforementioned pruning criteria. We also extend the plan generator further by incorporating a transformation described in previous work [14]. It consists of replacing a sequence of a grouping and a join by a single operator called *groupjoin*. Once the plan generator is capable of reordering grouping operators and joins, the introduction of groupjoins seems obvious, but some conditions have to be fulfilled to make the groupjoin applicable. A detailed description of these conditions and how compliance with them can be ensured during plan generation is one of the contributions of this paper.

The paper is organized as follows: Sect. 2 covers some basic concepts such as properties of aggregate functions and algebraic operators. We then continue with an overview of the equivalences needed to reorder join and grouping in Sect. 3. This section also covers the introduction of groupjoins during plan generation. In Sect. 4, we recap the basics of a dynamic programming-based plan generator and the extensions that are necessary to implement eager aggregation, the introduction of groupjoins, and optimality-preserving pruning in such a plan generator. Section 5 is dedicated to the plan properties needed for our pruning approaches and how they can be captured during plan generation. The pruning criteria them-

selves are covered in Sects. 6, 7, 8 and 9. Section 10 contains experimental results. Section 11 concludes the paper.

2 Preliminaries

We start with different characteristics of aggregate functions. Subsequently, we will introduce the algebraic operators used throughout the rest of the paper.

2.1 Aggregate functions and their properties

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 $F = (b_1 : \text{sum}(a), b_2 : \text{count}(*))$. 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 aggregation vector F is denoted by $\mathcal{F}(p)$ or $\mathcal{F}(F)$, respectively. The following properties of aggregate functions will be illustrated by some examples in the next section.

2.1.1 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 1 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 .

2.1.2 Decomposability

Another property of aggregate functions that is of particular interest in this paper is decomposability [3]:

Definition 2 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:

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

In contrast to the functions above, $\text{sum}(\text{distinct})$ and $\text{count}(\text{distinct})$ 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 $\text{avg}(X) = \text{sum}(X)/\text{count}(X)$. Since both sum and count are decomposable, we can decompose avg as follows:

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

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

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

2.1.3 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 [16].

For SQL aggregate functions, we have that

- \min , \max , $\text{sum}(\text{distinct})$, $\text{count}(\text{distinct})$, $\text{avg}(\text{distinct})$ 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 $F = (b_1 : \text{agg}_1(a_1), \dots, b_m : \text{agg}_m(a_m))$ be an aggregation vector. Further, let c be some other attribute. In our case, c is an attribute holding the result of $\text{count}(\ast)$. Then, we define $F \otimes c$ as

$$F \otimes c := (b_1 : \text{agg}'_1(e_1), \dots, b_m : \text{agg}'_m(e_m))$$

with

$$\text{agg}'_i(e_i) = \begin{cases} \text{agg}_i(e_i) & \text{if } \text{agg}_i \text{ is duplicate-agnostic,} \\ \text{agg}_i(e_i \ast c) & \text{if } \text{agg}_i \text{ is sum,} \\ \text{sum}(c) & \text{if } \text{agg}_i(e_i) = \text{count}(\ast), \end{cases}$$

and if $\text{agg}_i(e_i)$ is $\text{count}(e_i)$, then $\text{agg}'_i(e_i) := \text{sum}(e_i = \text{null} ? 0 : c)$.

2.2 Algebraic operators

In this subsection, we introduce the algebraic operators used throughout the rest of the paper. Although set notation is used in the definitions, all operators work on bags of tuples.

We denote the grouping operator by Γ . It can be defined as

$$\begin{aligned} \Gamma_{\theta G; b; f}(e) &:= \{y \circ [b : x] \mid y \in \Pi_G^D(e), \\ &x = f(\{z \mid z \in e, z.G \theta y.G\})\} \end{aligned}$$

for some set of grouping attributes G , a single attribute b , an aggregate function f and a comparison operator $\theta \in \{=, \neq, \leq, \geq, <, >\}$. Tuple concatenation is denoted by \circ . We denote by $\Pi_A^D(e)$ 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 duplicates. The function f is then applied to groups of tuples taken from this expression. The groups are determined by the comparison operator θ . Afterward, a new tuple consisting of the grouping attributes' values and an attribute b holding the corresponding value returned by the aggregate function f is constructed. The grouping criterion may be defined on several attributes. If all θ equal '=', we abbreviate $\Gamma_{=G; b; f}$ by $\Gamma_{G; b; f}$.

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

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

where the attribute values b_1, \dots, b_k are obtained by applying the aggregation vector $F = (f_1, \dots, f_k)$, consisting of k aggregate functions, to the tuples grouped according to θ .

The map operator χ extends every input tuple by new attributes:

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

As usual, selection is defined as

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

The join operators we consider are the (inner) join (\bowtie), left semijoin (\ltimes), left antijoin (\triangleright), left outer join (\ltimes), full outer join (\bowtie), and groupjoin (\bowtie). The definitions of these operators are given in Fig. 3. Most of these operators are rather standard. However, both the left and the full outer join are generalized such that for tuples not finding a join partner, default values can be provided instead of null padding. More specifically, let $D^i = d_1^i : c_1^i, \dots, d_k^i : c_k^i (i = 1, 2)$ be two vectors assigning constants c_j^i to attributes d_j^i . The definitions of the left and full outer join with defaults are given in 7 and 8, respectively. There, \perp_A denotes a tuple containing the value null in all attributes from attribute set A . Figure 4 provides examples.

The last row defines the left groupjoin \bowtie , introduced by von Bültingsloewen [1]. First, for a given tuple $t_1 \in e_1$, it determines the set 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 b , containing the result of this aggregation. Figure 4 gives an example.

3 Equivalences

This section is organized into three parts. The first part shows how to push down or pull up a grouping operator, the second

$$\begin{aligned} e_1 \times e_2 &:= \{r \circ s \mid r \in e_1, s \in e_2\} \\ e_1 \ltimes_p e_2 &:= \{r \circ s \mid r \in e_1, s \in e_2, p(r, s)\} \\ e_1 \ltimes_p e_2 &:= \{r \mid r \in e_1, \exists s \in e_2, p(r, s)\} \\ e_1 \triangleright_p e_2 &:= \{r \mid r \in e_1, \nexists s \in e_2, p(r, s)\} \\ e_1 \ltimes_p e_2 &:= (e_1 \ltimes_p e_2) \cup ((\perp_{\mathcal{A}(e_2)} \times \{ \perp_{\mathcal{A}(e_2)} \}) \\ e_1 \bowtie_p e_2 &:= (e_1 \ltimes_p e_2) \cup \\ &\quad \cup ((e_1 \triangleright_p e_2) \times \{ \perp_{\mathcal{A}(e_2)} \}) \\ &\quad \cup (\{ \perp_{\mathcal{A}(e_1)} \} \times (e_2 \triangleright_p e_1)) \\ e_1 \bowtie_p^{D^2} e_2 &:= (e_1 \ltimes_p e_2) \cup \\ &\quad ((e_1 \triangleright_p e_2) \times \{ \perp_{\mathcal{A}(e_2) \setminus \mathcal{A}(D^2)} \circ [D^2] \}) \\ e_1 \bowtie_p^{D^1, D^2} e_2 &:= (e_1 \ltimes_p e_2) \cup \\ &\quad ((e_1 \triangleright_p e_2) \times \{ \perp_{\mathcal{A}(e_2) \setminus \mathcal{A}(D^2)} \circ [D^2] \}) \cup \\ &\quad (\{ \perp_{\mathcal{A}(e_1) \setminus \mathcal{A}(D^1)} \circ [D^1] \} \times (e_2 \triangleright_p e_1)) \\ e_1 \bowtie_{p;b:f} e_2 &:= \{r \circ [b : x] \mid r \in e_1, \\ &\quad x = f(\{s \mid s \in e_2, p(r, s)\})\} \end{aligned}$$

Fig. 3 Join operators

e_1			e_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	3	4	2

$e_1 \ltimes_{e_1.a=e_2.e}^{e:7} e_2$						$e_1 \ltimes_{e_1.a=e_2.e}^{b:7;e:7} e_2$					
a	b	c	d	e	f	a	b	c	d	e	f
0	0	1	0	0	1	0	0	1	0	0	1
1	0	1	1	1	1	1	0	1	1	1	1
2	1	3	2	2	1	2	1	3	2	2	1
3	2	3	-	7	-	3	2	3	-	7	-
-	-	-	-	-	-	-	7	-	3	4	2

$e_1 \bowtie_{e_1.a=e_2.f;g:sum(e_2.f)} e_2$			
a	b	c	g
1	0	1	3
2	1	3	2

Fig. 4 Examples of different join operators

part shows how to eliminate an unnecessary top grouping operator. The third part covers the replacement of a sequence consisting of a join and a grouping by a groupjoin.

3.1 Pushing group-by

Since the work by Yan and Larson [16–20] is the most general, we take it as the basis for our work. Figure 5 shows 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 Eqv. 20 is already known from previous work [9]. The proofs of all equivalences are provided in our technical report [7].

Within the equivalences, a couple of simple abbreviations as well as some conventions occur. We introduce them in this short paragraph and illustrate them by means of two examples afterward. 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 assumes that F is splittable into F_1 and F_2 . If F_1 or F_2 does not occur in some 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.

$$\begin{aligned}
 & \text{Eager and Lazy Group-by-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:\text{count}(*))})(e_1) \bowtie_p e_2 \\
 \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:\text{count}(*))})(e_1) \bowtie_p e_2 \\
 \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:\text{count}(*))})(e_1) \bowtie_p^{F_1^1(\{\perp\}),c_1:1;-} e_2 \\
 \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:\text{count}(*))})(e_2) \\
 \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(\{\perp\}),c_2:1} \Gamma_{G_2^+;F_2^1 \circ (c_2:\text{count}(*))})(e_2) \\
 \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(\{\perp\}),c_2:1} \Gamma_{G_2^+;F_2^1 \circ (c_2:\text{count}(*))})(e_2) \\
 & \text{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) \\
 \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) \\
 \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^{F_1^1(\{\perp\});-} e_2) \\
 \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)) \\
 \Gamma_{G;F}(e_1 \bowtie_p e_2) & \equiv \Gamma_{G;F_2^2}(e_1 \bowtie_p^{F_2^1(\{\perp\})} \Gamma_{G_2^+;F_2^1}(e_2)) \\
 \Gamma_{G;F}(e_1 \bowtie_p e_2) & \equiv \Gamma_{G;F_2^2}(e_1 \bowtie_p^{-;F_2^1(\{\perp\})} \Gamma_{G_2^+;F_2^1}(e_2)) \\
 & \text{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:\text{count}(*)})(e_1) \bowtie_p e_2 \\
 \Gamma_{G;F}(e_1 \bowtie_p e_2) & \equiv \Gamma_{G;(F_2 \otimes c_1)}(\Gamma_{G_1^+;(c_1:\text{count}(*))})(e_1) \bowtie_p e_2 \\
 \Gamma_{G;F}(e_1 \bowtie_p e_2) & \equiv \Gamma_{G;(F_2 \otimes c_1)}(\Gamma_{G_1^+;(c_1:\text{count}(*))})(e_1) \bowtie_p^{c_1:1;-} e_2 \\
 \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:\text{count}(*)})(e_2) \\
 \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:\text{count}(*)})(e_2) \\
 \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:\text{count}(*))})(e_2) \\
 & \text{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:\text{count}(*)})(e_2) \\
 \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:\text{count}(*)})(e_2) \\
 \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^{F_1^1(\{\perp\});c_2:1} \Gamma_{G_2^+;(c_2:\text{count}(*))})(e_2) \\
 \Gamma_{G;F}(e_1 \bowtie_p e_2) & \equiv \Gamma_{G;(F_2^2 \otimes c_1)}(\Gamma_{G_1^+;c_1:\text{count}(*)})(e_1) \bowtie_p \Gamma_{G_2^+;F_2^1}(e_2) \\
 \Gamma_{G;F}(e_1 \bowtie_p e_2) & \equiv \Gamma_{G;(F_2^2 \otimes c_1)}(\Gamma_{G_1^+;c_1:\text{count}(*)})(e_1) \bowtie_p^{F_2^1(\{\perp\})} \Gamma_{G_2^+;F_2^1}(e_2) \\
 \Gamma_{G;F}(e_1 \bowtie_p e_2) & \equiv \Gamma_{G;(F_2^2 \otimes c_1)}(\Gamma_{G_1^+;(c_1:\text{count}(*))})(e_1) \bowtie_p^{c_1:1;F_2^1(\{\perp\})} \Gamma_{G_2^+;F_2^1}(e_2) \\
 & \text{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:\text{count}(*))})(e_1) \bowtie_p \Gamma_{G_2^+;F_2^1 \circ (c_2:\text{count}(*))}(e_2) \\
 \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:\text{count}(*))})(e_1) \bowtie_p^{F_2^1(\{\perp\}),c_2:1} \Gamma_{G_2^+;F_2^1 \circ (c_2:\text{count}(*))}(e_2) \\
 \Gamma_{G;F}(e_1 \bowtie_p e_2) & \equiv \Gamma^2(\Gamma_{G_1^+;F_1^{1,1} \circ (c_1:\text{count}(*))})(e_1) \bowtie_p^{F_1^{1,1}(\{\perp\}),c_1:1;F_2^{1,1}(\{\perp\}),c_2:1} \Gamma_{G_2^+;F_2^{1,1} \circ (c_2:\text{count}(*))}(e_2) \\
 & \text{Others} \\
 \Gamma_{G;F}(e_1 \bowtie_q e_2) & \equiv \Gamma_{G;F}(e_1) \bowtie_p e_2 \quad (\mathcal{F}(q) \cap \mathcal{A}(e_1)) \subseteq G \\
 \Gamma_{G;F}(e_1 \bowtie_q e_2) & \equiv \Gamma_{G;F}(e_1) \bowtie_p e_2 \quad (\mathcal{F}(q) \cap \mathcal{A}(e_1)) \subseteq G \\
 \Gamma_{G;F}(e_1 \bowtie_{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:\text{count}(*))})(e_1) \bowtie_{J_1 \theta J_2; \overline{F}} e_2) \\
 \Gamma_{G;F}(e_1 \bowtie_{J_1 \theta J_2; \overline{F}} e_2) & \equiv \Gamma_{G;F_1^2}(\Gamma_{G_1^+;F_1^1}(e_1) \bowtie_{J_1 \theta J_2; \overline{F}} e_2) \\
 \Gamma_{G;F}(e_1 \bowtie_{J_1 \theta J_2; \overline{F}} e_2) & \equiv \Gamma_{G;(F_2 \otimes c_1)}(\Gamma_{G_1^+;(c_1:\text{count}(*))})(e_1) \bowtie_{J_1 \theta J_2; \overline{F}} e_2)
 \end{aligned}$$

Fig. 5 Equivalences for eager and lazy aggregation

3.1.1 Example 1: join

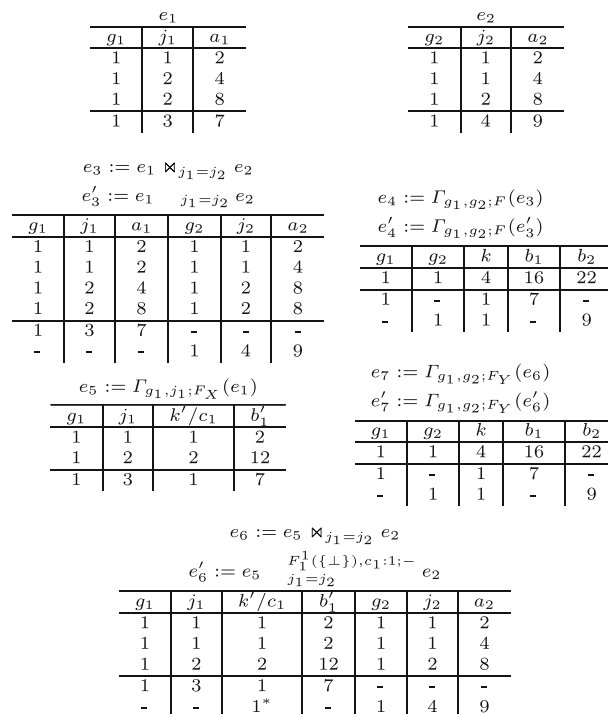
Figure 6 shows two relations e_1 and e_2 , which will be used to illustrate Eqv. 10 as well as Eqv. 12.

Let us start with Eqv. 10. 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 Fig. 6 serve as input. The calculation of the result of the left-hand side of Eqv. 10 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 $F = k : \text{count}(*), b_1 : \text{sum}(a_1), b_2 : \text{sum}(a_2)$. The result is given as e_4 . In our 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 Eqv. 10, 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 gives us $F_1 = (k : \text{count}(*), b_1 : \text{sum}(a_1))$ and it does not matter whether we add k to F_1 or F_2 , since it does not reference any attributes. Next, we need to decompose F_1 into F_1^1 and F_1^2 according to the insights of Sect. 2.1. This results in $F_1^1 = (k' : \text{count}(*), b_1' : \text{sum}(a_1))$ and $F_1^2 = (k : \text{sum}(k'), b_1 : \text{sum}(b_1'))$. The inner grouping operator of Eqv. 10 requires us to add an attribute $c_1 : \text{count}(*)$ to F_1^1 , which we abbreviate by F_X . Since there already exists one $\text{count}(*)$, the result of which is stored in k' , we keep only one of them in Fig. 6 and call it 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 Fig. 6. 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. Eqv. 10 requires us to calculate $F_2 \otimes c_1$. Looking back at the end of Sect. 2.1, we see that sum is duplicate-sensitive and that $F_2 \otimes c_1 = b_2 : \text{sum}(c_1 * a_2)$. Concatenating this aggregation vector with F_1^2 , as demanded by Eqv. 10, gives us F_Y , as specified in Fig. 6. The final result of the right-hand side of Eqv. 10, calculated as $e_7 = \Gamma_{g_1, g_2; F_Y}(e_6)$, is given in Fig. 6. Note that it is equal to the result of the left-hand side (e_4).

3.1.2 Example 2: full outer join

The second example reuses the relations e_1 and e_2 given in Fig. 6. But this time we calculate the full outer join instead of the inner join, and we apply Eqv. 12. The cor-



where

$$\begin{aligned}
 F &= (k : \text{count}(*), b_1 : \text{sum}(a_1), b_2 : \text{sum}(a_2)) \\
 F_1 &= (k : \text{count}(*), b_1 : \text{sum}(a_1)) \\
 F_2 &= (b_2 : \text{sum}(a_2)) \\
 F_1^1 &= (k' : \text{count}(*), b_1' : \text{sum}(a_1)) \\
 F_1^2 &= (k : \text{sum}(k'), b_1 : \text{sum}(b_1')) \\
 F_X &= F_1^1 \circ (c_1 : \text{count}(*)) \\
 F_Y &= (F_2 \otimes c_1) \circ F_1^2 \\
 &= (b_2 : \text{sum}(c_1 * a_2), k : \text{sum}(k'), b_1 : \text{sum}(b_1'))
 \end{aligned}$$

$$\text{and } G = \{g_1, g_2\}, G_1^+ = \{g_1, j_1\}.$$

Fig. 6 Example for Eqvs. 10 and 12

responding 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 $e_1 \bowtie_{j_1=j_2} e_2$ 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 outer join on the right-hand side of Eqv. 12. 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, $F_1^1(\{\perp\})$ evaluates to 1 for k ($\text{count}(*)$ on a relation with a single element), and null for a_2 , since SQL's sum returns null for sets containing only null values. Thus prepared, we can calculate the right-hand side of Eqv. 12 via e_5 and e_6 . For the latter, we use a full outer join with default. Finally, e_7 is calculated by grouping e_6 , leading to the same result as e_4 .

3.1.3 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 outer join, deriving one from the other is simple. For non-commutative operators like the left outer join, an additional proof is necessary. Now, instead of pushing the grouping operator into only one argument, we can combine both equivalences to push it 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*, *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 arbitrary aggregation vector \bar{F} . All have in common that after they are applied, only the attributes from their left input are accessible. Thus, the grouping operator can only be pushed into their left argument.

3.2 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 $F = (b_1 : \text{agg}_1(a_1), \dots, b_k : \text{agg}_k(a_k))$. Clearly, this is only possible if G is a superkey for e and e is duplicate-free, since in this case, there exists exactly one tuple in e for each group. The only detail left is to calculate the aggregation vector F . This can be done via a map operator as in the following expression:

$$\Gamma_{G;F}(e) \equiv \Pi_C(\chi_{\hat{F}}(e)), \quad (42)$$

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

$$\hat{F} := (b_1 : \text{agg}_1(\{a_1\}), \dots, b_k : \text{agg}_k(\{a_k\})).$$

Remark In general, K is a superkey for relation R if $K \rightarrow \mathcal{A}(R)$ 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.

3.3 Replacing group-by and left outer join by groupjoin

We can replace a sequence of a grouping and a left outer join by a single groupjoin [14]:

$$\Gamma_{G;F}(e_1 \bowtie_{A_1=A_2} e_2) \equiv \Pi_C(e_1 \bowtie_{A_1=A_2;F} e_2) \quad (43)$$

if

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

We denote by $\text{TID}(e)$ the tuple identifier for e . All these requirements are mandatory [14].

The preconditions for Equivalence 43 refer to functional dependencies holding in the join result. As we will 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 43. Note that they are not equivalent.

Theorem 1

$$\Gamma_{G;F}(e_1 \bowtie_{A_1=A_2} e_2) \equiv \Pi_C(e_1 \bowtie_{A_1=A_2;F} e_2) \quad (44)$$

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 43 and 44 are equal, we only have to prove that the first three constraints from Equivalence 44 imply the first three from Equivalence 43. We refer to the different requirements by the number of the respective equivalence followed by the number of the requirement. For example, 44::1 refers to the first requirement listed under Equivalence 44. We prove each implication in a separate paragraph.

44::1 \Rightarrow 43::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:

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

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 attribute values are equal if they are both null, as suggested by Paulley [15].

44::2 \Rightarrow 43::2 The second requirement can be strengthened to $G \rightarrow \text{TID}(e_1)$. In other words, G has to be a superkey for e_1 . Again, we express this in terms of superkeys: $\exists K \in \kappa(e_1), K \subseteq G$.

44::3 \Rightarrow 43::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: $K \in \kappa(e_2), K \subseteq A_2$. \square

3.4 Replacing group-by and inner join by groupjoin

We can replace a sequence of a grouping and an inner join by a single groupjoin [14]:

$$\Gamma_{G;F}(e_1 \bowtie_{A_1=A_2} e_2) \equiv \Pi_C(\sigma_{c_2>0}(e_1 \bowtie_{A_1=A_2; F \circ (c_2:count(*))} e_2)) \quad (45)$$

if

1. $G \rightarrow G_2^+$ holds in $e_1 \bowtie_{A_1=A_2} e_2$,
2. $G_1, G_2^+ \rightarrow \text{TID}(e_1)$ 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.

Theorem 2

$$\Gamma_{G;F}(e_1 \bowtie_{A_1=A_2} e_2) \equiv \Pi_C(\sigma_{c_2>0}(e_1 \bowtie_{A_1=A_2; F \circ (c_2:count(*))} e_2)) \quad (46)$$

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

$$\Gamma_{G;F}(e_1 \bowtie_{A_1=A_2} e_2) \equiv \Pi_C(e_1 \bowtie_{A_1=A_2; F \circ (c_2:count(*))} e_2) \quad (47)$$

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. $(e_1 \bowtie_{A_1=A_2} e_2) = e_1$, and
5. $\mathcal{F}(F) \subseteq \mathcal{A}(e_2)$.

Proof The only difference between Theorem 2 and Corollary 1 is that the latter omits the selection after the groupjoin. In Equivalence 46, 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 9), but not in the result of a join followed by a grouping. If the fourth condition stated above holds, i.e., $(e_1 \bowtie_{A_1=A_2} e_2) = e_1$, all tuples from e_1 find a join partner in e_2 and the selection can be omitted. This case holds if there is a foreign-key constraint where A_1 references A_2 and no selection was applied to e_2 before the join. \square

3.5 Some remarks concerning the groupjoin

Information on how to implement the groupjoin can be found in previous work [14]. There, a straightforward implementation combining join and grouping is proposed. One system that currently supports the groupjoin is HyPer [11].

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 materialization of a large intermediate result.

4 Dynamic programming

We briefly repeat the basics of a plan generator based on dynamic programming (DP). The plan generator is then extended to allow for the reordering of joins and grouping operators, as well as the introduction of groupjoins.

4.1 Plan-generation basics

Figure 7 shows the basic structure of a typical DP-based plan generator. Its input consists of three major pieces: the set of relations to be joined and the set of join operators with associated predicates from the input query as well as a query hypergraph. A hypergraph is defined as follows:

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 $u \cap v = \emptyset$.

DP-PLANGEN

```

// Input: a set of relations  $R = \{R_0, \dots, R_{n-1}\}$ 
           a set of operators  $O$  with associated predicates
           a query hypergraph  $H$ 
// Output: an optimal bushy operator tree
1  for all  $R_i \in R$ 
2     $DPTable[R_i] = R_i$  // initial access paths
3  for all csg-cmp-pairs  $(S_1, S_2)$  of  $H$ 
4    for all  $\diamond_p \in O$ 
5      if APPLICABLE( $S_1, S_2, \diamond_p$ )
6        BUILDPLANS( $S_1, S_2, \diamond_p$ )
7        if  $\diamond_p$  is commutative
8          BUILDPLANS( $S_2, S_1, \diamond_p$ )
9  return  $DPTable[R]$ 

```

BUILDPLAN(S_1, S_2, \diamond_p)

```

1   $OptimalCost = \infty$ 
2   $S = S_1 \cup S_2$ 
3   $T_1 = DPTable[S_1]$ 
4   $T_2 = DPTable[S_2]$ 
5  if  $DPTable[S] \neq NULL$ 
6     $OptimalCost = COST(DPTable[S])$ 
7  if  $COST(T_1 \diamond_p T_2) < OptimalCost$ 
8     $OptimalCost = COST(T_1 \diamond_p T_2)$ 
9   $DPTable[S] = (T_1 \diamond_p T_2)$ 

```

Fig. 7 Plan generator based on DP

The hypergraph is constructed by a conflict detector such that the nodes of the graph represent the relations referenced in the input query and every edge in the graph represents a join predicate between two or more relations. The conflict detector encodes reordering conflicts as far as possible into the hypergraph by appropriately extending the relation sets that make up a hyperedge [12]. This is necessary since inner joins and outer joins are not freely reorderable.

The major data structure used by the plan generator is the *DP table*, which stores an optimal plan for every plan class. A plan class comprises all plans that are equivalent with respect to an equivalence relation. In a typical DP-based plan generator like the one shown in the figure, plans are considered to be equivalent if they produce the same result. However, different equivalence criteria are possible and we will see an example of this in the following subsection. Plan properties can be classified 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.

The plan generator consists of four major components. The first component initializes the DP table with the 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 4 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.

The ccps are enumerated in an order suitable for DP; that is, before a pair (S_1, S_2) is emitted, all ccps contained in S_1 and S_2 are emitted. Such an enumerator has been described in previous work [13].

The third component (Line 5) is an applicability test for operators. It builds upon the conflict representation and checks whether some operator \diamond_p can be safely applied. This check is necessary since it is not possible to exactly cover all reordering conflicts within a hypergraph representation of the query [12].

The fourth component (BUILDPLAN) is a procedure that builds plans using some operator \diamond_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 DP table. 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 grouping operators.

4.2 Extending the plan generator

We introduce the new routine OPTREES shown in Fig. 8. Its arguments are two join trees T_1 and T_2 , and a join operator \diamond_p . The result consists of a set of up to nine different trees joining T_1 and T_2 with newly introduced grouping operators and groupjoins.

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 \diamond_p without any grouping.

When a join tree containing all the relations in the query is created, that is, $(S_1 \cup S_2) = R$, we have to add another grouping on top of \diamond_p if the grouping attributes do not comprise a superkey (see Sect. 3). NEEDSGROUPING checks this condition.

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, i.e., it corresponds to one of the equivalences from Sect. 3. The VALID subroutine checks for this correspondence. Additionally, we have to avoid the case in which the grouping attributes G_i^+ form a superkey for the set S_i , with $i \in \{1, 2\}$, because then the grouping would be a waste. And again, if necessary, we have to add a grouping on top. Up to three more trees are built with different combinations of grouping the left or right input of the join until the subroutine GROUPJOINTREES is called in the last line of OPTREES. If the introduction of groupjoins is not desired, we can simply return the set *Trees* at the end of OPTREES.

```

OPTREES( $T_1, T_2, \diamond_p$ )
1  $S_1 = \mathcal{T}(T_1)$ 
2  $S_2 = \mathcal{T}(T_2)$ 
3  $S = S_1 \cup S_2$ 
4  $Trees = \emptyset$ 
5  $NewTree = (T_1 \diamond_p T_2)$ 
6 if  $S == R \wedge \text{NEEDSGROUPING}(G, NewTree)$ 
7    $NewTree = (\Gamma_G(NewTree))$ 
8    $Trees.\text{INSERT}(NewTree)$ 
9    $NewTree = \Gamma_{G_1^+}(T_1) \diamond_p T_2$ 
10 if  $\text{VALID}(NewTree) \wedge \text{NEEDSGROUPING}(G_1^+, NewTree)$ 
11   if  $S == R \wedge \text{NEEDSGROUPING}(G, NewTree)$ 
12      $NewTree = (\Gamma_G(NewTree))$ 
13      $Trees.\text{INSERT}(NewTree)$ 
14    $NewTree = T_1 \diamond_p \Gamma_{G_2^+}(T_2)$ 
15 if  $\text{VALID}(NewTree) \wedge \text{NEEDSGROUPING}(G_2^+, NewTree)$ 
16   if  $S == R \wedge \text{NEEDSGROUPING}(G, NewTree)$ 
17      $NewTree = (\Gamma_G(NewTree))$ 
18      $Trees.\text{INSERT}(NewTree)$ 
19    $NewTree = \Gamma_{G_1^+}(T_1) \diamond_p \Gamma_{G_2^+}(T_2)$ 
20 if  $\text{VALID}(NewTree)$ 
     $\wedge \text{NEEDSGROUPING}(G_1^+, NewTree)$ 
     $\wedge \text{NEEDSGROUPING}(G_2^+, NewTree)$ 
21   if  $S == R \wedge \text{NEEDSGROUPING}(G, NewTree)$ 
22      $NewTree = (\Gamma_G(NewTree))$ 
23      $Trees.\text{INSERT}(NewTree)$ 
24 return  $\text{GROUPJOINTREES}(S, Trees)$ 

NEEDSGROUPING( $G, T$ )
1 if  $G \rightarrow \mathcal{A}(T) \wedge$  the result of  $T$  is duplicate-free
2   return FALSE
3 else
4   return TRUE

```

Fig. 8 OPTREES and NEEDSGROUPING

The pseudocode for GROUPJOINTREES is given in Fig. 9. As argument we pass the relation set S and the set $Trees$ which at this point contains up to four different join trees for S , as depicted on the left-hand side of Fig. 10. For each tree, 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 check the requirements for Equivalences 43/45 or 44/46 from Sect. 3, depending on whether we have full information about functional dependencies or only superkeys available. A call to the routine ISGROUPJOINAPPLICABLE checks if the aforementioned requirements are met. If the routine returns *true*, we add the resulting tree to the set *GroupjoinTrees*. In the pseudocode, we include all selections that may be necessary according to the equivalences for the groupjoin. We refer to the left and right subtree of T_1 by $T_{1,1}$ and $T_{1,2}$, respectively. 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 Fig. 10 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 Fig. 9).

```

GROUPJOINTREES( $S, Trees$ )
1  $GroupjoinTrees = \emptyset$ 
   // Final grouping  $\Gamma_G$  is optional:
2 for all  $T = (\Gamma_G(T_1 \diamond_p T_2))$  in  $Trees$ 
3   if  $S == R \wedge \text{ISGROUPED}(T) \wedge \text{GROUPJOINAPPLICABLE}(T)$ 
4      $GroupjoinTrees.\text{INSERT}(\sigma_{p_s}(T_1 \bowtie_p T_2))$ 
5   if  $\text{ISGROUPED}(T_1) \wedge \text{GROUPJOINAPPLICABLE}(T_1)$ 
6      $GroupjoinTrees.\text{INSERT}((\sigma_{p_{s1}}(T_{1,1} \bowtie_{p_1} T_{1,2})) \diamond_p T_2)$ 
7     if  $S == R \wedge \text{ISGROUPED}(T) \wedge \text{GROUPJOINAPPLICABLE}((\sigma_{p_{s1}}(T_{1,1} \bowtie_{p_1} T_{1,2})) \diamond_p T_2)$ 
8        $GroupjoinTrees.\text{INSERT}(\sigma_{p_s}((\sigma_{p_{s1}}(T_{1,1} \bowtie_{p_1} T_{1,2})) \bowtie_p T_2))$ 
9     if  $\text{ISGROUPED}(T_2) \wedge \text{GROUPJOINAPPLICABLE}(T_2)$ 
10    ... // Build tree with groupjoin in the right subtree
11 if  $\text{ISGROUPED}(T_1) \wedge \text{ISGROUPED}(T_2) \wedge \text{GROUPJOINAPPLICABLE}(T_1) \wedge \text{GROUPJOINAPPLICABLE}(T_2)$ 
12    ... // Build tree with groupjoins in both subtrees
13 return  $Trees \cup GroupjoinTrees$ 

```

Fig. 9 GROUPJOINTREES

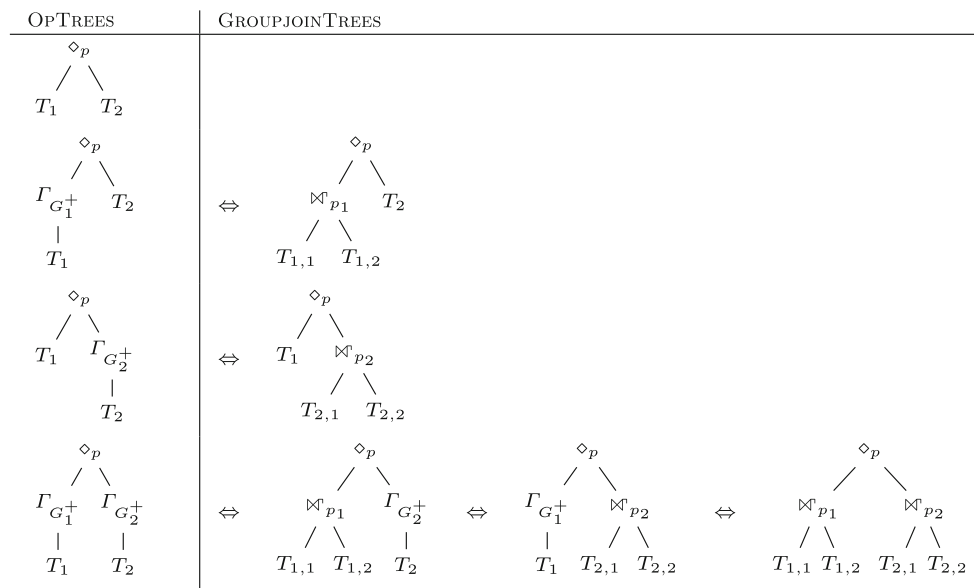


Fig. 10 Trees enumerated by OPTREES and GROUPJOINTREES

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 not apply once eager aggregation is taken into consideration. If we push a grouping operator into one or both arguments of a join operator, this can influence two properties of the respective subtree: The cardinality of the tree’s result may be reduced and the functional dependencies holding in the result may be altered. A reduced cardinality can reduce the cost of subsequent operations and thereby (more than) compensate the cost of the additional grouping operator. The functional dependencies, on the other hand, determine whether or not we need a final grouping on top to fix the query result (see Sect. 3). 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. Please refer to our previous work for more details and an example [8].

For this reason, the dynamic programming table is modified to store for each plan class not only one plan but a set of plans. In addition to this, the plans of one class are no longer equivalent in the sense that they all produce the same result. Instead, we use the following equivalence relation for defining 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

BUILDALLPLANS(S_1, S_2, \diamond_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}

```

Fig. 11 BUILDALLPLANS

plan properties, we thus consider the cardinality of a plan a physical property since it can differ between plans of the same class. The same is true for the key properties and functional dependencies holding in the result of a plan.

Figure 11 shows the routine BUILDALLPLANS, which is derived from the routine BUILDPLANS depicted in Fig. 7 and illustrates the modifications necessary to take these new aspects into account. As before, we enumerate all ccps (S_1, S_2) with $S = S_1 \cup S_2$. We then call BUILDALLPLANS instead of BUILDPLAN in Lines 6 and 8 of the algorithm shown in Fig. 7. In the new subroutine, every tree for S_1 is combined with every tree for S_2 using two loops. We call OPTREES for each pair of join trees, which results in up to nine different trees for every combination. The newly created trees are added to the tree set for S .

Eventually, we face the situation where $S = R$ holds and we need to build a join tree for the complete query.

```

PRUNEDOMINATEDPLANS( $S, T$ )
1  for  $T_{old} \in DPTable[S]$ 
2  if  $T_{old}$  dominates  $T$ 
3  return
4  if  $T$  dominates  $T_{old}$ 
5     $DPTable[S].REMOVE(T_{old})$ 
6   $DPTable[S].INSERT(T)$ 

```

Fig. 12 PRUNEDOMINATEDPLANS

At this point, we call another subroutine named INSERTTO-LEVELPLAN. Inside this routine, we compare the join trees for S to find the one with minimal cost 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 grouping by calling NEEDSGROUPING (Fig. 8). In contrast to the other relation sets, we do not have to keep a set of trees for R , but only the best tree found so far and replace it if a better one is found.

For n relations, the runtime complexity of this algorithm is $O(2^{2n-1}\#ccp)$ if $\#ccp$ denotes the number of ccps for the query (see Definition 4).

4.3 Optimality-preserving pruning

As we have seen in the previous section, keeping all possible trees in the solution table guarantees an optimal solution but 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 12 shows the routine PRUNEDOMINATEDPLANS, which discards all trees that are *dominated* by some other 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. For this purpose, we replace Line 8 in BUILDALLPLANS by PRUNEDOMINATEDPLANS(S, T). The loop beginning in Line 1 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} , then T_{old} is deleted from the DP table. In this case, the loop continues because more dominated trees that can be discarded may exist. Finally, T is added to the set for S .

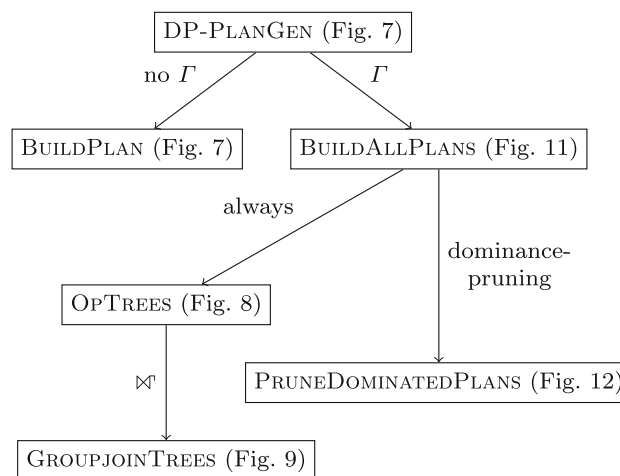


Fig. 13 Configuration options for the plan generator

In Sects. 6, 7, 8 and 9, we discuss different notions of dominance and evaluate them with respect to their effectiveness as a pruning criterion.

To summarize this section, Fig. 13 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 \boxtimes), and dominance-pruning.

5 Interesting plan properties and their derivation

In this section we provide rules for computing properties of query plans that we use to determine dominance.

5.1 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. Note also that attributes contained in the keys resulting from the full and left outer join can have the value null. We therefore assume that null values are treated as suggested by Pauley, i.e., two attributes values are equal if they are both null [15]. 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 Pauley's definition of functional dependency, where two attributes with value null are treated as equal [15]. Initially, FDs for a base relation are deduced from the keys declared in the database schema. Below, we will fre-

quently use the closure¹ of a given set of FDs, denoted by FD^+ .

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 $\{a_1, a_2, \dots, a_n\}$ 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:

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

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

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

Definite attributes We denote by $NN(e)$ the set of definite attributes in an expression e . Definite attributes are attributes that never have the value null. If e is a base relation, $NN(e)$ contains the attributes that are declared as “not null” in the database schema.

5.2 Deriving interesting properties

We provide rules for computing the four sets bottom-up in an operator tree possibly containing all algebraic operators covered in Sect. 2.2. The rules concerning EC and FD are taken from Paulley [15]. 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.

One concept that is useful for the computation of the aforementioned properties is null rejection of a predicate p on attribute a . It is defined as follows [10]:

Definition 5 A predicate p rejects nulls on attribute a if it does not evaluate to true if a is null.

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

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

5.2.1 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 [8]:

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

5.2.2 Left outer join

Consider the left outer join of expressions e_1 and e_2 : $e_1 \bowtie_p e_2$. Since the left outer join 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 $\kappa(e_1 \bowtie_{a_1=a_2} e_2) = \kappa(e_1)$.

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

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

where p is an arbitrary predicate.

Functional dependencies All FDs holding in e_1 , the preserved side of the outer join, continue to hold in the join result. Dependencies from e_2 , the null-supplying side of the outer join, 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:

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

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 outer join of expressions e_1 and e_2 , where a_1 belongs to e_1 and a_2 belongs to e_2 : $e_1 \bowtie_{a_1=a_2} e_2$. In this case, the following equation holds:

$$FD(e_1 \bowtie_{a_1=a_2} e_2) = FD^+(e_1) \cup \{(\alpha \rightarrow \beta) \in FD^+(e_2) \mid (\alpha \cap (NN(e_2) \cup NR(p)) \neq \emptyset)\} \cup \{a_1 \rightarrow a_2\}.$$

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

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

Definite attributes Since the left outer join 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: $NN(e_1 \bowtie_p e_2) = NN(e_1)$.

5.2.3 Full outer join

Consider the full outer join of expressions e_1 and e_2 : $e_1 \bowtie_p e_2$.

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:

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

where p is an arbitrary join predicate.

Functional dependencies Since in the full outer join 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 outer join. 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.

$$FD(e_1 \bowtie_{a_1=a_2} e_2) = \{(\alpha \rightarrow \beta) \in FD^+(e_1) \mid (\alpha \cap NN(e_1) \neq \emptyset) \vee (p \text{ is null-rejecting in } \mathcal{F}(p) \cap \mathcal{A}(e_1))\} \cup \{(\alpha \rightarrow \beta) \in FD^+(e_2) \mid (\alpha \cap NN(e_2) \neq \emptyset) \vee (p \text{ is null-rejecting in } \mathcal{F}(p) \cap \mathcal{A}(e_2))\}.$$

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 outer join, equality constraints from both join arguments remain valid in the result of a full outer join: $EC(e_1 \bowtie_p e_2) = EC(e_1) \cup EC(e_2)$.

Definite attributes The full outer join can introduce null values in all attributes contained in the join result. This means that there are no definite attributes after the join: $NN(e_1 \bowtie_p e_2) = \emptyset$.

5.2.4 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 ($e_1 \bowtie_p^g e_2$) of expression e_1 and e_2 . According to our definitions from Sect. 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 $\kappa(e_1 \diamond e_2) = \kappa(e_1)$, for $\diamond \in \{\bowtie, \triangleright, \bowtie_{G:A:F}\}$.

Functional dependencies $FD(e_1 \diamond_p e_2) = FD^+(e_1)$, for $\diamond \in \{\bowtie, \triangleright\}$.

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

Equality constraints $EC(e_1 \diamond_p e_2) = EC(e_1)$, for $\diamond \in \{\bowtie, \triangleright, \bowtie\}$.

Definite attributes $NN(e_1 \diamond_p e_2) = NN(e_1)$, for $\diamond \in \{\bowtie, \triangleright\}$.

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

5.2.5 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: $\kappa(\Gamma_{G:A:F}(e)) = \{K \in \kappa(e) \mid K \subseteq G\}$.

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 grouping, 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 determine the aggregation attributes:

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

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

```

ATTRIBUTE CLOSURE( $FD, EC, \alpha$ )
1   $result = \alpha$ 
2  repeat
3     $hasChanged = FALSE$ 
4    for all  $e \in EC$ 
5      if  $(e \cap result) \neq \emptyset$ 
6         $result = result \cup e$ 
7    for all FDs  $\beta \rightarrow \gamma$  in  $FD$ 
8      if  $\beta \subseteq result$ 
9         $result = result \cup \gamma$ 
10      $hasChanged = TRUE$ 
11  until  $hasChanged = FALSE$ 
12  return  $result$ 
    
```

Fig. 14 ATTRIBUTE CLOSURE

5.3 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 $\alpha \rightarrow \beta$, 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 $\beta' \rightarrow \gamma$ with $\beta' \subseteq result$, which gives the transitive dependency $\alpha \rightarrow \gamma$. In this case, we add γ to the result and repeat the whole process until there are no more changes.

The pseudocode for ATTRIBUTE CLOSURE is given in Fig. 14. 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.

5.4 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 store 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 Fig. 14.

6 Pruning with functional dependencies

First, we define f-dominance [8]:

Definition 6 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 Sect. 2, a grouping on top of the final join may be necessary to compensate this difference.

Theorem 3 Let T_2 be an arbitrary 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.

In order to avoid the overhead associated with computing FD^+ , which is used to define f-dominance, the plan generator described in our previous work applies the following pruning criterion, which we call k-dominance because it is based on keys [8]:

Definition 7 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 an arbitrary 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:

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

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

Correctness proofs for Theorems 3 and 4 can be found in our technical report [6]. In short, the proofs go as follows. Assume that T_1 is constructed from T_2 by replacing T_2^{sub} by T_1^{sub} . Since T_1^{sub} is cheaper than T_2^{sub} and has a smaller cardinality, T_1 has to be cheaper than the original T_2 . The only case in which T_2 could be less expensive than T_1 is the one where T_1 needs a final grouping and T_2 does not. It can be shown that this will never be the case, because all functional dependencies (or keys) arising in T_2 between the newly inserted T_2^{sub} and the root of the tree must also 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 15 shows two operator trees for the same query on relations R_0, \dots, 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 $i \in (0, \dots, 3)$. In addition to the operators, the trees shown in Fig. 15 contain special nodes displaying the keys valid at the respective point in the tree according to the key computation rules from Sect. 5. 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. 7). 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

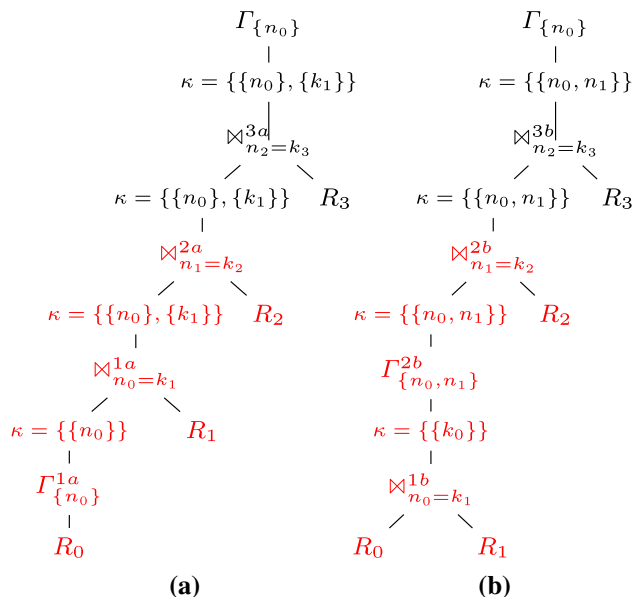


Fig. 15 Two operator trees with keys

Table 1 Functional dependencies for Fig. 15

	Figure 15a		Figure 15b	
	AC ⁺	EC	AC ⁺	EC
R_0	$\{k_0\} \rightarrow \{k_0, n_0\}$	\emptyset	$\{k_0\} \rightarrow \{k_0, n_0\}$	\emptyset
R_1	$\{k_1\} \rightarrow \{k_1, n_1\}$	\emptyset	$\{k_1\} \rightarrow \{k_1, n_1\}$	\emptyset
R_2	$\{k_2\} \rightarrow \{k_2, n_2\}$	\emptyset	$\{k_2\} \rightarrow \{k_2, n_2\}$	\emptyset
R_3	$\{k_3\} \rightarrow \{k_3, n_3\}$	\emptyset	$\{k_3\} \rightarrow \{k_3, n_3\}$	\emptyset
Γ^1	$\{n_0\} \rightarrow \{n_0\}$	\emptyset	–	\emptyset
\bowtie^1	$\{\{n_0, k_1\}\} \rightarrow \{\{n_0, k_1\}, n_1\}$	$\{n_0, k_1\}$	$\{\{n_0, k_1\}\} \rightarrow \{\{n_0, k_1\}, n_1\}$	$\{n_0, k_1\}$
Γ^2	–	$\{n_0, k_1\}$	$\{n_0, n_1\} \rightarrow \{n_0, n_1\}$	\emptyset
\bowtie^2	$\{\{n_0, k_1\}\} \rightarrow \{\{n_0, k_1\}, \{n_1, k_2\}, n_2\}$	$\{n_0, k_1\}$	$\{n_0\} \rightarrow \{n_0, \{n_1, k_2\}, n_2\}$	$\{n_1, k_2\}$
	$\{\{n_1, k_2\}\} \rightarrow \{\{n_1, k_2\}, n_2\}$	$\{n_1, k_2\}$	$\{n_0, n_1\} \rightarrow \{n_0, \{n_1, k_2\}, n_2\}$	
			$\{\{n_1, k_2\}\} \rightarrow \{\{n_1, k_2\}, n_2\}$	

check if $\kappa(\bowtie^{2a}) \subseteq \kappa(\bowtie^{2b})$ holds. Here and in the following examples we write $\kappa(\diamond)/FD^+(\diamond)$ instead of $\kappa(T)/FD^+(T)$, respectively, where \diamond is the operator at the root of T . Obviously, the aforementioned condition is not fulfilled 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 Fig. 15. 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 Sect. 5. 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 Fig. 15a, 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 Sect. 5 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 Fig. 15a in favor of the one in Fig. 15b by considering the FDs holding in both trees instead of the keys? That is, we need to check if the following relationship holds:

$$FD^+(\bowtie^{2a}) \subseteq FD^+(\bowtie^{2b}). \quad (48)$$

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 Eq. 48 holds. We can therefore safely discard the more expensive tree.

Taking a closer look at Table 1, we also see that $\kappa(\bowtie^{2b}) = \{\{n_0\}\}$, since all attributes present in the tree are determined by n_0 . The key resulting from the key computation shown in Fig. 15 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

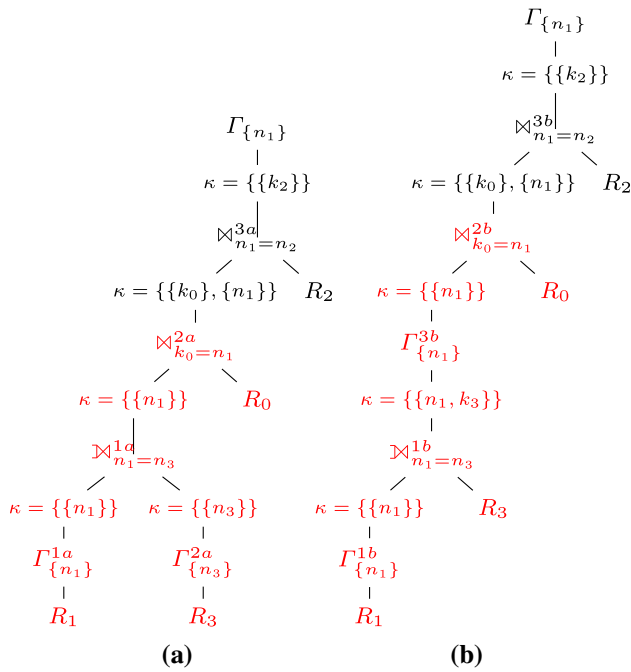


Fig. 16 Two operator trees with keys

the opposite holds, especially in the presence of non-inner joins. We present an example in Fig. 16.

We assume the same relation schemas as in the previous example, and we are again interested in discarding the tree in Fig. 16a because it is more expensive with equal cardinality as the one in Fig. 16b. 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 7. 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 $\{k_0, n_1\} \rightarrow \{n_3\}$, 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 outer join \bowtie^1 , we replace it by an inner join. This results in an equivalence class $\{n_1, n_3\}$, which is later extended to $\{k_0, n_1, n_3\}$. Thereby, the problematic dependency from above is turned 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} .

7 Pruning with restricted keys

Using f-dominance instead of k-dominance often is beneficial as it enables more opportunities for pruning. But computing and comparing the needed properties is more expensive. In this section, we propose a third pruning approach that makes use of keys and at the same time allows for an effective pruning. Again, we provide an example, consisting of two alternative join trees for the same query. They are shown in Fig. 17.

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 Fig. 17a, 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 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

Table 2 Functional dependencies for Fig. 16

	Figure 16a		Figure 16b	
	AC ⁺	EC	AC ⁺	EC
R_0	$\{k_0\} \rightarrow \{k_0, n_0\}$	\emptyset	$\{k_0\} \rightarrow \{k_0, n_0\}$	\emptyset
R_1	$\{k_1\} \rightarrow \{k_1, n_1\}$	\emptyset	$\{k_1\} \rightarrow \{k_1, n_1\}$	\emptyset
R_3	$\{k_3\} \rightarrow \{k_3, n_3\}$	\emptyset	$\{k_3\} \rightarrow \{k_3, n_3\}$	\emptyset
Γ^1	$\{n_1\} \rightarrow \{n_1\}$	\emptyset	$\{n_1\} \rightarrow \{n_1\}$	\emptyset
Γ^2	$\{n_3\} \rightarrow \{n_3\}$	\emptyset	–	\emptyset
\bowtie^1	$\{n_1\} \rightarrow \{n_1, n_3\}$	\emptyset	$\{n_1\} \rightarrow \{n_1, n_3\}$	\emptyset
	$\{n_3\} \rightarrow \{n_3\}$		$\{k_3\} \rightarrow \{k_3, n_3\}$	
Γ^3	–	\emptyset	$\{n_1\} \rightarrow \{n_1\}$	\emptyset
\bowtie^2	$\{k_0, n_1\} \rightarrow \{k_0, n_1, n_0, n_3\}$	$\{k_0, n_1\}$	$\{k_0, n_1\} \rightarrow \{k_0, n_1, n_0\}$	$\{k_0, n_1\}$
	$\{n_3\} \rightarrow \{n_3\}$			

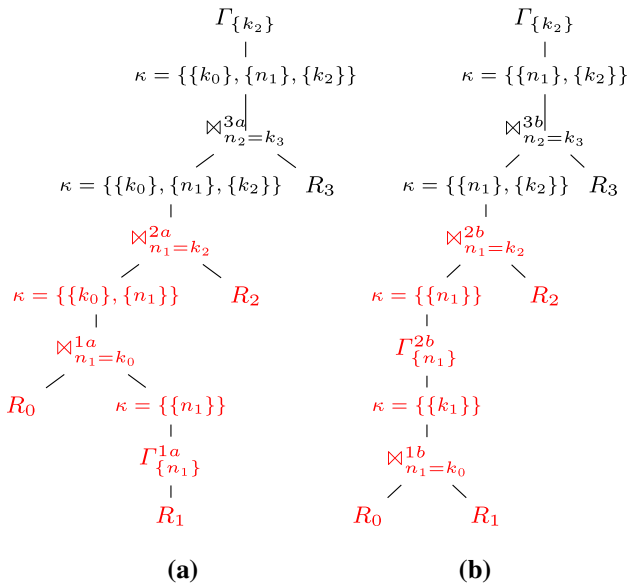


Fig. 17 Two Operator Trees with Keys

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 grouping. The same argument implies that we can also ignore $\{n_1\}$. 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:

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

We can now define rk-dominance:

Definition 8 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*

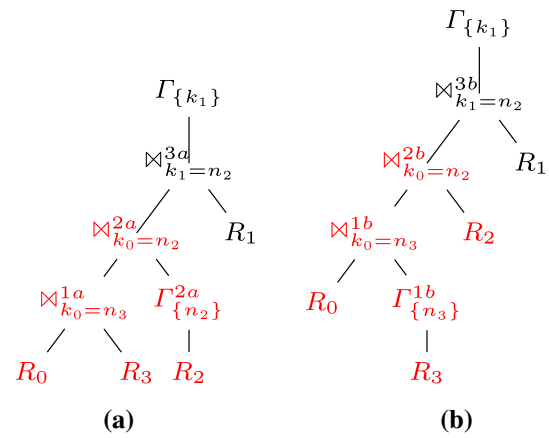


Fig. 18 Two operator trees

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 .

8 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 FD set* 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 Fig. 18, where we assume that the subtree for relation set $\{R_0, R_2, R_3\}$ in Fig. 18a is more expensive than the one in Fig. 18b. Table 3 contains the FDs and 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 the comparison of the two trees, we can discard the subtree from Fig. 18a. 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 9 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 .

Correctness proofs for both theorems referring to restricted property sets (Theorems 5 and 6) can be found in our technical report [6]. In short, they can be proved by showing that functional dependencies or keys not contained in the restricted sets are never propagated in such a way that they end up in one of the restricted property sets further up in the tree.

9 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 sometimes 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.

10 Evaluation

We evaluate our algorithms with respect to runtime, memory consumption and plan quality. However, since the plans without groupjoins produced by the different plan generators are all equivalent in terms of plan quality, we do not compare their costs. Adding groupjoins, we can sometimes reduce the costs further. We therefore provide a comparison of resulting plan costs between the algorithms with and without groupjoins. Memory consumption is measured as the number of entries in the DP table after successful plan generation. This number also hints at the effectiveness of the pruning criteria.

We implemented all pruning approaches discussed in the previous sections in the plan generator DPHypE, which is based on the plan generator DPHyp [13] and capable of reordering non-inner joins and grouping operators. We denote by *f/k/rk/rf/rkrf* the plan generator employing f-/k-/rk-/rf-/rkrf-dominance, respectively. In addition to this, we extended *rk* and *rf* to enable the introduction of groupjoins. The resulting algorithms are referred to as *girk* and *girf*.

The workload consists of randomly generated join trees, cardinalities for the base relations, and selectivities. A grouping operator with a randomly chosen set of grouping

Table 3 Functional dependencies for Fig. 18

	Figure 18a AC ⁺	Figure 18b AC ⁺	Figure 18a/18b 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	$\{\{k_0, n_2, n_3\}\} \rightarrow \{\{k_0, n_2, n_3\}, n_0\}$ $\{k_3\} \rightarrow \{k_3, \{k_0, n_2, n_3\}, n_0\}$	$\{\{k_0, n_2, n_3\}\} \rightarrow \{\{k_0, n_2, n_3\}, n_0\}$ $\{k_2\} \rightarrow \{k_2, \{k_0, n_2, n_3\}, n_0\}$	$\{k_0, n_2, n_3\}$

attributes is placed at the top of the join tree. The results given in this section are averages for 10,000 queries for a given number of relations in the join 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. The join predicates are binary equality predicates. With respect to the join operators, we distinguish two cases: trees with only inner joins and trees where at each inner node one operator $\diamond \in \{\bowtie, \Join, \Join\}$ is randomly chosen. All experiments were conducted on an Intel Xeon E5-2690 V2 @ 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 does not lie on join reordering, which has been thoroughly investigated in prior work and is in its 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–key predicates.

For an evaluation of the general effectiveness of reordering grouping and join operators, the reader is referred to previous work [8, 16–19] containing experiments not only for a synthetic workload, but also for selected benchmark queries. For the general effectiveness of the groupjoin, the reader is referred to the previous work on the topic, where its impact on the runtimes of TPC-H queries is investigated [14].

10.1 Plan generator without groupjoins

The following subsections deal with the plan generators applying eager aggregation without the introduction of groupjoins.

10.1.1 Runtime

Figures 19 and 20 show the runtimes of the five plan generators without groupjoins for queries with 5 to 15 relations. The runtimes shown in Fig. 19 result from queries containing only inner joins, while Fig. 20 depicts queries containing inner, left outer and full outer joins. The search space for queries with only inner joins is larger than for queries containing non-inner joins, since inner joins are freely reorderable. This is why the runtimes in Fig. 19 are higher than the ones in Fig. 20. Since the runtimes of *k* and *f* are so high, we did not run them for larger numbers of relations.

In both cases, we fix the proportion of foreign-key–key join predicates to 80%. 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

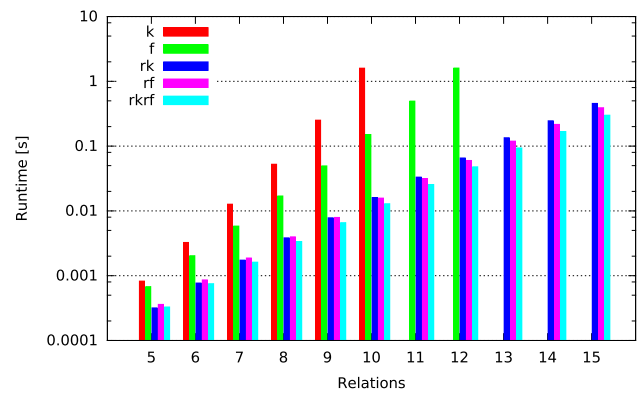


Fig. 19 Runtimes with inner joins

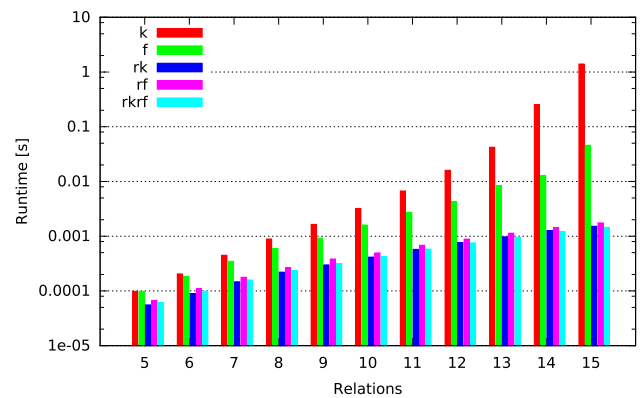


Fig. 20 Runtimes with inner and outer joins

small, making the comparison of said sets faster and increasing the chance of one plan dominating another. We consider 80% 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 s on average, while *rkrf* requires only 0.0015 s, 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 and arbitrary join operators, we measured the following numbers for “time per plan comparison”: 23 / 306 / 2073 / 3705 / 3077 nanoseconds for *k* - / *f* - / *rk* - / *rf* - / *rkrf*-dominance, respectively. Note that these numbers are based on the assumption that the plan com-

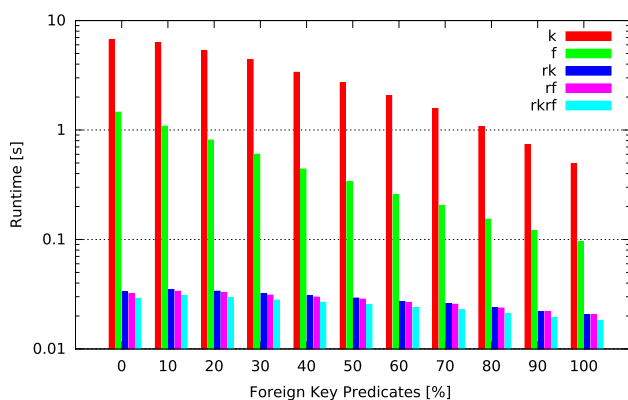


Fig. 21 Runtimes for 10 relations with inner joins

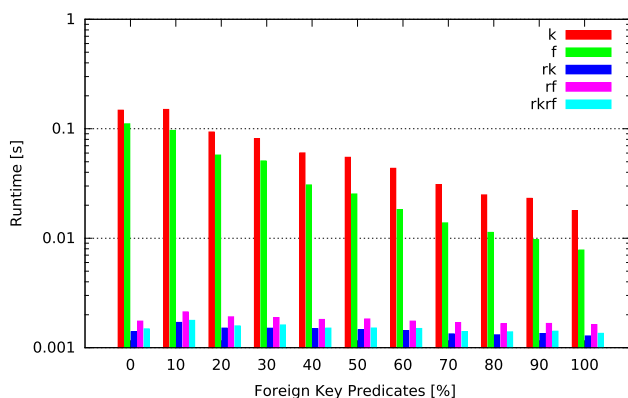


Fig. 22 Runtimes for 10 relations with inner and outer joins

parisons are the dominating influence on the plan generator's runtime, which may not always be true.

When considering queries containing only inner joins, we observe the following trend for larger queries (see Fig. 19): 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 21 and 22 show the runtimes for queries with 10 relations and an increasing percentage of foreign-key-key joins from 0 to 100 with 10,000 input trees each.

10.1.2 Memory usage

The reasons for the runtime differences between k and the rest become obvious when we look at the number of DP table entries produced by the different algorithms, as depicted in Figs. 23 and 24. As suggested by our observations in the previous sections, the least effective pruning criterion is k -dominance and the most effective is rf -dominance. Com-

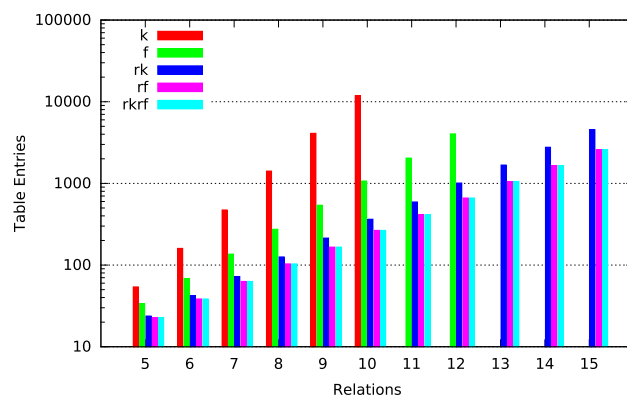


Fig. 23 Table entries with inner joins

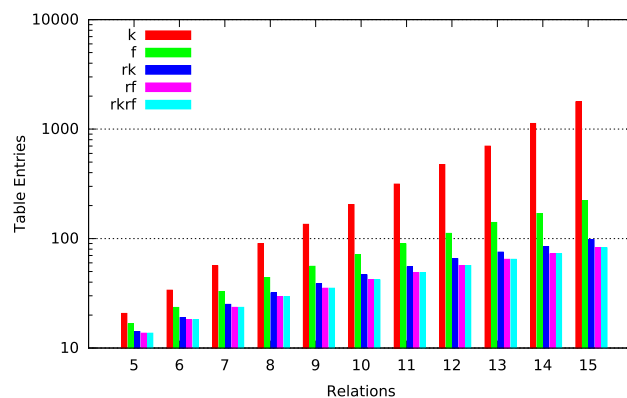


Fig. 24 Table entries with inner and outer joins

binning 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 outer joins, the average number of table entries is $1800 / 82$ 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, we have $12,000 / 270$ table entries on average for the same two plan generators and queries with 10 relations.

10.2 Plan generator with groupjoins

The following subsections deal with the plan generators applying eager aggregation and groupjoins.

10.2.1 Optimal plan cost

Once we enable the introduction of groupjoins in our plan generator, 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

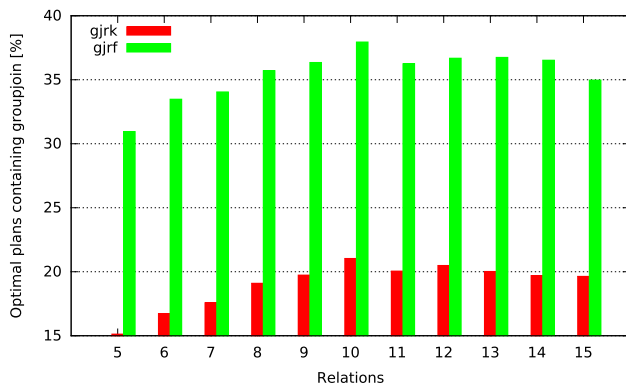


Fig. 25 Percentage of optimal plans containing groupjoins with inner joins

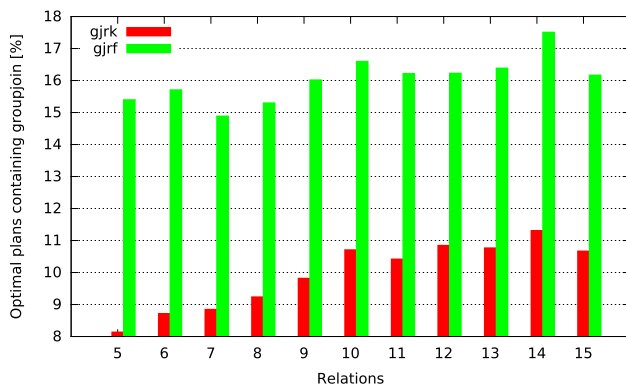


Fig. 26 Percentage of optimal plans containing groupjoins with inner and outer joins

in Sect. 3, we cannot precisely check the requirements for applying a groupjoin if we only consider keys. Thus, we may fail to introduce a groupjoin if only keys are known, even though it would be possible. Figures 25 and 26 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% of the plans produced by *gjrf* contain at least one groupjoin. For *gjrj* this number amounts to 21%.

Table 4 contains the costs achieved by *gjrj* and *gjrf* in relation to the cost achieved without groupjoins for 7, 10 and 15 relations. We only provide the minimum value, i.e., the best relative cost achieved over all 10,000 queries of one size. That is because groupjoins can only be applied in a fraction of the considered queries and the cost savings achieved by introducing groupjoins fluctuates considerably depending on the characteristics of the query (see Sect. 3.5 and [14]). But as we will 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,

Table 4 Relative plan cost groupjoins/no groupjoins

Relations	Inner joins		Inner/outer joins	
	<i>gjrj</i>	<i>gjrf</i>	<i>gjrj</i>	<i>gjrf</i>
7	0.72	0.45	0.79	0.74
10	0.72	0.53	0.83	0.83
15	0.75	0.53	0.90	0.7

we can sometimes reduce the plan cost to less than 50%. The differences between *gjrj* 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 considered.

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 outer joins, because in this scenario few groupjoins can be applied.

For a demonstration of the groupjoin's effectiveness when applied to benchmark queries, the reader is referred to the previous work, where a speedup factor of more than three is reported for TPC-H query 13 after introducing groupjoins. The complete benchmark is sped up by a factor of 1.5 [14].

10.2.2 Runtime

Figures 27 and 28 show the runtimes of the two plan generators with groupjoins and their counterparts without groupjoins. The figures show that the difference in runtimes between the respective variants is 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 will see in the next subsection, this number is little affected by the introduction of groupjoins.

10.2.3 Memory usage

Figures 29 and 30 show the number of table entries stored in the DP table after plan generation for the algorithms with

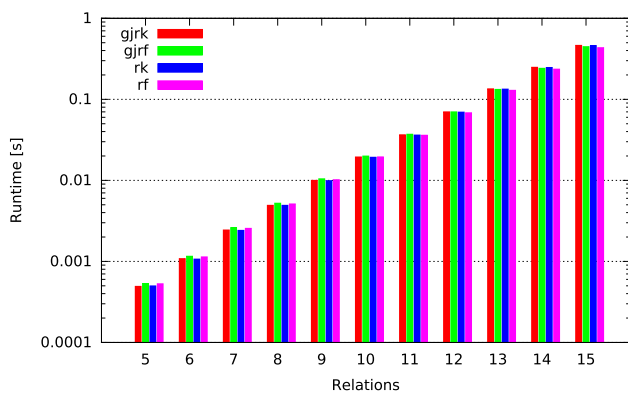


Fig. 27 Runtime with groupjoins and inner joins

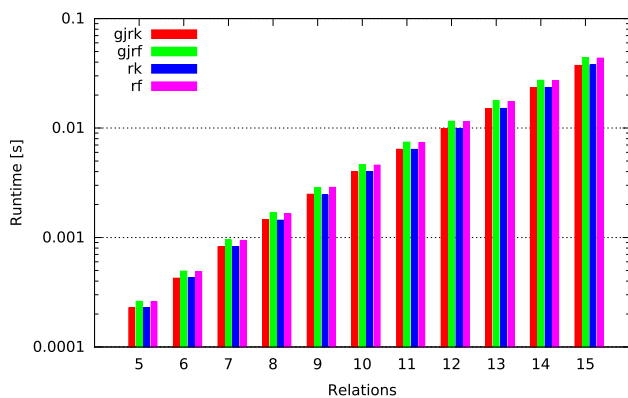


Fig. 28 Runtime with groupjoins, inner and outer joins

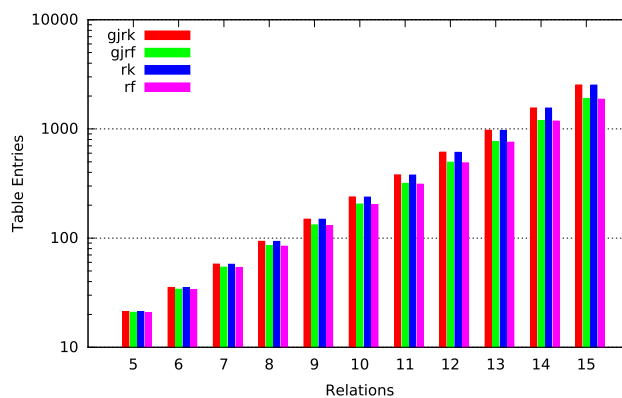


Fig. 29 Table entries with groupjoins and inner joins

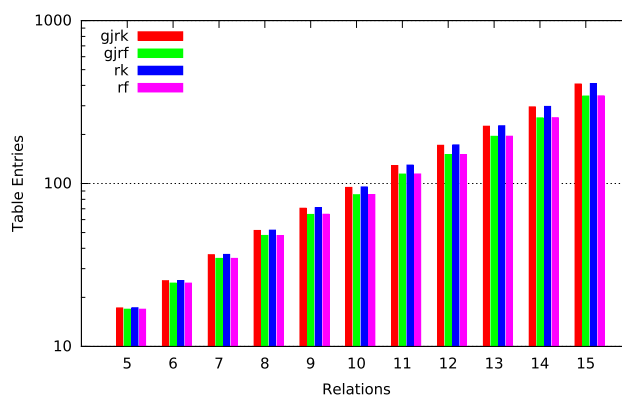


Fig. 30 Table entries with groupjoins, inner and outer joins

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 a left outer join or inner join and a grouping, 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 some other 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.

11 Conclusion

We presented a complete framework for the optimization of queries containing joins and grouping. To this end, we extended the concept of eager aggregation to make it applicable to a variety of join operators. This forms the foundation for designing a plan generator that is capable of optimizing not only the join order, but also the placement of grouping

operators. We then enabled the introduction of groupjoins to further improve the resulting plans. Defining a set of optimality-preserving pruning criteria allows the application of these optimization techniques to large queries. To this end, we first analyzed the plan properties needed for pruning and how they can be derived during plan generation. We then showed that using functional dependencies instead of keys, as proposed in previous work [8], reveals 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.

Our experiments reveal the influence of pruning on the runtime of the plan generator, leading to a speed up factor of up to several orders of magnitude when compared to the only existing approach. The additional introduction of groupjoins allows to further reduce the cost of the resulting plans, while at the same time imposing almost no additional overhead. However, when groupjoins are considered, the plan generators relying on keys instead of functional dependencies are no longer guaranteed to find the optimal plan.

Acknowledgements We thank Simone Seeger for her help preparing the manuscript and the reviewers for their helpful feedback.

References

1. von Bültingsloewen, G.: Optimizing SQL queries for parallel execution. *SIGMOD Rec.* **18**, 17–22 (1989)
2. Chaudhuri, S., Shim, K.: Including group-by in query optimization. In: *Proceedings of International Conference on Very Large Data Bases (VLDB)*, vol 94, pp. 354–366 (1994)
3. Cluet, S., Moerkotte, G.: Efficient evaluation of aggregates on bulk types. In: *International Workshop on Database Programming Languages* (1995)
4. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: *Introduction to Algorithms*, 3rd edn. The MIT Press, Cambridge (2009)
5. Eich, M., Fender, P., Moerkotte, G.: Faster plan generation through consideration of functional dependencies and keys. In: *Proceedings of International Conference on Very Large Data Bases (VLDB)*, vol 9(10), pp. 756–767 (2016)
6. Eich, M., Fender, P., Moerkotte, G.: Efficient generation of query plans containing group-by, join, and groupjoin. Technical report, University of Mannheim (2017)
7. Eich, M., Moerkotte, G.: Dynamic programming: The next step. Technical report, University of Mannheim (2014)
8. Eich, M., Moerkotte, G.: Dynamic programming: the next step. In: *Proceedings of IEEE Conference on Data Engineering*, pp. 903–914 (2015)
9. Galindo-Legaria, C., Joshi, M.: Orthogonal optimization of subqueries and aggregation. In: *Proceedings of the ACM SIGMOD Conference on Management of Data*, pp. 571–581 (2001)
10. Galindo-Legaria, C., Rosenthal, A.: Outerjoin simplification and reordering for query optimization. *ACM Trans. Database Syst.* **22**(1), 43–74 (1997)
11. Kemper, A., Neumann, T.: Hyper: A hybrid OLTP & OLAP main memory database system based on virtual memory snapshots. In: *Proceedings of IEEE Conference on Data Engineering*, pp. 195–206 (2011)
12. Moerkotte, G., Fender, P., Eich, M.: On the correct and complete enumeration of the core search space. In: *Proceedings of the ACM SIGMOD Conference on Management of Data*, pp. 493–504 (2013)
13. Moerkotte, G., Neumann, T.: Dynamic programming strikes back. In: *Proceedings of the ACM SIGMOD Conference on Management of Data*, pp. 539–552 (2008)
14. Moerkotte, G., Neumann, T.: Accelerating queries with group-by and join by groupjoin. In: *Proceedings of International Conference on Very Large Data Bases (VLDB)*, vol 4(11) (2011)
15. Paulley, G.: Exploiting functional dependence in query optimization. Ph.D. thesis, University of Waterloo (2000)
16. Yan, W.: Rewriting optimization of SQL queries containing group-by. Ph.D. thesis, University of Waterloo (1995)
17. Yan, W., Larson, P.A.: Performing group-by before join. Technical Report CS 93-46, Dept. of Computer Science, University of Waterloo, Canada (1993)
18. Yan, W., Larson, P.A.: Performing group-by before join. In: *Proceedings of IEEE Conference on Data Engineering*, pp. 89–100 (1994)
19. Yan, W., Larson, P.A.: Eager aggregation and lazy aggregation. In: *Proceedings of International Conference on Very Large Data Bases (VLDB)*, vol 95, pp. 345–357 (1995)
20. Yan, W., Larson, P.A.: Interchanging the order of grouping and join. Technical Report CS 95-09, Dept. of Computer Science, University of Waterloo, Canada (1995)