Dynamic Programming: The Next Step

Marius Eich [1], Guido Moerkotte [2]

University of Mannheim
Mannheim, Germany
[1] marius.eich@uni-mannheim.de
[2] moerkotte@uni-mannheim.de

***Abstract*—We fill two gaps in the literature. First, we give a comprehensive set of equivalences allowing reordering of grouping with non-inner joins. Second, we show how to incorporate the optimal placement of grouping into a state-of-the-art dynamic programming (DP)-based plan generator.**

I. Introduction

In a recent paper, Shanbhag and Sudarshan pointed out that the biggest disadvantage of DP-based plan generators (PG) (like [1]) is that they are not capable of reordering grouping and joins [2]. Here, we show how to overcome this deficit. Further, we fill another gap in the literature: the lack of equivalences useful to push grouping down non-inner joins. Consider the following query against the TPC-H schema:

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

The main point here is that since only reorderings between grouping and inner joins are known [3], [4], [5], [6], [7], [8], the outer join constitutes a barrier to any reordering with grouping. This has severe consequences. On HyPer [9], the execution time is 2140 ms. The plan produced by the DP-based PG presented in this paper reduces it to 1.51 ms. (Similar results were obtained for two major commercial systems: for system 1 we got 8480 ms vs. 47 ms and for system 2 64900 ms vs. 210 ms.) Since in general, reordering grouping and outer joins is not a correct rewrite, we eliminate the barrier by generalizing the definition of outer joins (Sec. III).

After this preliminary step, we show how to integrate these equivalences into a modern DP-based PG. Since free placement of grouping extends the search space substantially, we also present one optimality-preserving pruning technique and two heuristics.

The rest of the paper is organized as follows. The next section presents some preliminaries. Then, we discuss the new equivalences. Sec. IV starts by introducing the basic DP-based PG. Then, we show how to extend it such that the search space enlarged by the new equivalences is systematically explored. Since the enlarged search space may become quite large, we discuss two heuristics and an optimality-preserving pruning technique to make the approach feasible. Sec. V contains experimental results. We first evaluate the gain of our approach, then its costs. Sec. VI summarizes the achievements and proposes directions for future research.

II. Preliminaries

A. 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 F of aggregate functions. If F_1 and F_2 are two vectors of aggregate functions, we denote by $F_1 \circ F_2$ their concatenation.

As usual, the set of attributes provided by some expression e (e.g. a base relation) is denoted by $\mathcal{A}(e)$ and the set of attributes referenced by some expression e (e.g. a predicate) is denoted by $\mathcal{F}(e)$.

The following definitions of properties of aggregate functions will be illustrated by some examples in the next section. There, it also becomes clear why they are needed.

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 arbitrary expressions e_1 and e_2 if $F = F_1 \circ F_2$, $\mathcal{F}(F_1) \cap \mathcal{A}(e_2) = \emptyset$ and $\mathcal{F}(F_2) \cap \mathcal{A}(e_1) = \emptyset$.

In this case, we can evaluate F_1 on e_1 and F_2 on e_2. A special case S1 occurs for `count(*)`, which accesses no attributes and can thus be added to both, F_1 and F_2.

2) Decomposability: One property of aggregate functions that is of particular interest for the considerations in this paper is *decomposability* [10]:

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 be seen easily:

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

In contrast to their duplicate preserving counterparts, sum(distinct) and count(distinct) are not decomposable.

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

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

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 by count^{NN}. We can then use this to decompose avg as follows:

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

For special case S1, the two counts have to be multiplied.

3) Duplicate Sensitive and Agnostic: 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 [4].

For SQL aggregate functions, we have that

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

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

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

with

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

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

B. Algebraic Operators

One operator that plays an important role in the following sections is the grouping operator, which we denote by Γ. The grouping operator can be defined as

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

for some set of grouping attributes G, a single attribute g, an aggregate function f, and a comparison operator $\theta \in \{=, \neq, \leq, \geq, <, >\}$. We denote by $\Pi^D_A(e)$ the duplicate-removing projection onto the set of attributes A, applied to the expression e. The resulting relation only contains values for those attributes that are contained in A and no duplicate values. The function f is then applied to groups of tuples taken from this relation. The groups are determined by the comparison operator θ. Afterwards, a new tuple consisting of the grouping attribute's values and an attribute g holding the corresponding value calculated by the aggregate function f is constructed.

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

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

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

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

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

As usual, selection is defined as

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

The join operators we consider are the (inner) join ($\bowtie$), left semijoin $\ltimes$, left antijoin ($\triangleright$), left outerjoin ($\mathbin{⟕}$), full outerjoin ($\mathbin{⟗}$), and groupjoin ($\mathbin{⟕^{\Gamma}}$). The definitions of these join operators are given in Figure 1. There, $\circ$ denotes tuple concatenation. Most of these operators are rather standard. However, both the left and the full outerjoin are generalized such that for tuples not finding a join partner, default values can be provided instead of null padding. More specifically, let $D_i = d^i_1 : c^i_1, \ldots, d^i_k : c^i_k (i = 1, 2)$ be two vectors assigning constants c_j to attributes d^i_j. The definitions of the left and full outerjoin with defaults are given in 7 and 8, respectively. Fig. 2 provides examples.

The last row defines the left groupjoin $\mathbin{⟕^{\Gamma}}$, introduced by von Bültzingsloewen [11]. First, for a given tuple $t_1 \in e_1$, it determines the sets of all join partners for t_1 in e_2 using the join predicate p. Then, it applies the aggregate function f to these tuples and extends t_1 by a new attribute g containing the result of this aggregation. Figure 2 gives an example.

$$
\begin{aligned}
e_1 \times e_2 &:= \{r \circ s | r \in e_1, s \in e_2\} && (1)\\
e_1 \bowtie_p e_2 &:= \{r \circ s | r \in e_1, s \in e_2, p(r,s)\} && (2)\\
e_1 \ltimes_p e_2 &:= \{r | r \in e_1, \exists s \in e_2, p(r,s)\} && (3)\\
e_1 \triangleright_p e_2 &:= \{r | r \in e_1, \nexists s \in e_2, p(r,s)\} && (4)\\
e_1 \rtimes_p e_2 &:= (e_1 \bowtie_p e_2) \cup ((e_1 \triangleright_p e_2) \times \{\bot_{\mathcal{A}(e_2)}\}) && (5)\\
e_1 \rtimes\!\!\ltimes_p e_2 &:= (e_1 \bowtie_p e_2)\\
&\quad \cup ((e_1 \triangleright_p e_2) \times \{\bot_{\mathcal{A}(e_2)}\})\\
&\quad \cup (\{\bot_{\mathcal{A}(e_1)}\} \times (e_2 \triangleright_p e_1)) && (6)\\
e_1 \bowtie_p^{D_2} &:= (e_1 \bowtie_p e_2) \cup\\
&\quad ((e_1 \triangleright_p e_2) \times \{\bot_{\mathcal{A}(e_2)\setminus\mathcal{A}(D_2)} \circ [D_2]\}) && (7)\\
e_1 \rtimes\!\!\ltimes_p^{D_1;D_2} e_2 &:= (e_1 \bowtie_p e_2) \cup\\
&\quad ((e_1 \triangleright_p e_2) \times \{\bot_{\mathcal{A}(e_2)\setminus\mathcal{A}(D_2)} \circ [D_2]\} \cup\\
&\quad (\{\bot_{\mathcal{A}(e_1)\setminus\mathcal{A}(D_1)} \circ [D_1]\} \times (e_2 \triangleright_p e_1)) && (8)\\
e_1 \bowtie\!\!\!\!\Gamma_{p;g:f} e_2 &:= \{r \circ [g:G] | r \in e_1,\\
&\quad G = f(\{s | s \in e_2, p(r,s)\})\} && (9)
\end{aligned}
$$

Fig. 1. Join operators

e_1

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

e_2

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

$e_1 \bowtie_{e_1.b=e_2.d} e_2$

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

$e_1 \triangleright_{e_1.a=e_2.e} e_2$

a	b	c
3	2	3

$e_1 \ltimes_{e_1.b=e_2.d} e_2$

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

$e_1 \bowtie^{e:7}_{e_1.a=e_2.e} e_2$

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	-	7	-

$e_1 \rtimes\!\!\ltimes^{b:7;e:7}_{e_1.a=e_2.e} e_2$

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	-	7	-
-	7	-	3	4	2

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

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

Fig. 2. Examples of different join operators

C. Keys

We use information about keys to avoid unnecessary grouping operators. Thus, if the key information is incomplete, unnecessary grouping operators are introduced. However, it is important to note that this does *not* affect the correctness of plans, but only their efficiency.

The keys for base relations are specified in the database schema and therefore given. By using them and the join operators and predicates contained in a given plan it is possible to compute the keys for all intermediate results.

In the following paragraphs, we denote by $\kappa(e)$ the set of keys for a relation resulting from an expression e. Note that a single key is a set of attributes. Therefore, κ is a set of sets. Each of the following paragraphs covers one of the join operators specified in Sec. II-B.

1) Inner Join: We have to distinguish three cases:

- In case A_1 is a key of e_1 and A_2 is a key of e_2, we have
$$\kappa(e_1 \bowtie_{A_1=A_2} e_2) = \kappa(e_1) \cup \kappa(e_2).$$
That is, each key from one of the input expressions is again a key for the join result.
- In case A_1 is a key, but A_2 is not, we have
$$\kappa(e_1 \bowtie_{A_1=A_2} e_2) = \kappa(e_2).$$
The reverse case is handled analogously.
- Without any assumption on the A_i or the join predicate, we have
$$\kappa(e_1 \bowtie_q e_2) = \bigcup_{k_1 \in \kappa(e_1), k_2 \in \kappa(e_2)} k_1 \cup k_2.$$
In other words, every pair of keys from e_1 and e_2 forms a key for the join result.

2) Left Outerjoin: Here, we have only two possible cases. If A_2 is a key of e_2, then
$$\kappa(e_1 \rtimes_{A_1=A_2} e_2) = \kappa(e_1).$$
Otherwise, we have to combine two arbitrary keys from e_1 and e_2 to form a key:
$$\kappa(e_1 \rtimes_q e_2) = \bigcup_{k_1 \in \kappa(e_1), k_2 \in \kappa(e_2)} k_1 \cup k_2,$$
where q is an arbitrary predicate.

3) Full Outerjoin: 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:
$$\kappa(e_1 \rtimes\!\!\ltimes_q e_2) = \bigcup_{k_1 \in \kappa(e_1), k_2 \in \kappa(e_2)} k_1 \cup k_2,$$
where q is an arbitrary join predicate.

4) Left Semijoin/Left Antijoin/Left Groupjoin: Since the attributes from the right input are no longer present in the join result and the result is duplicate-free by definition, we always have
$$\kappa(e_1 \circ e_2) = \kappa(e_1)$$
for $\circ \in \{\ltimes, \triangleright, \bowtie\!\!\!\!\Gamma\}$.

Using these basic rules, the keys for every subtree of an operator tree can be computed bottom-up. Note that the keys resulting from the full and left outerjoin contain null values. We therefore assume that null values are treated as suggested in [12], i.e., two attributes are equal if they agree in value or they are both null.

III. EQUIVALENCES

This section is organized into two parts. The first part shows how to push down/pull up a grouping operator, the second part shows how to eliminate an unnecessary top grouping operator.

A. Pushing Group-By

Since the work by Yan and Larson [5], [6], [8], [7], [4] is the most general one, we take it as the basis for our work. Figure 3 shows all known and new equivalences. The nine equivalences already known from Yan and Larson's work can be recognized by the inner join on their left-hand side. The different section headings within the figures are those proposed by Yan and Larson (except for *Others*). A special case of Eqv. 20 occured in [13]. The proofs of all equivalences are provided in our technical report [14].

Within the equivalences, a couple of simple abbreviations as well as some conventions occur. We give them in this short paragraph and illustrate them by means of two examples afterwards. By G we denote the set of grouping attributes, by F a vector of aggregation functions, and by q 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., $J_i = \mathcal{A}(e_i) \cap \mathcal{F}(q)$. The union of the grouping and join attributes from e_i are denoted by $G_i^+ = G_i \cup J_i$. If F_1 and/or F_2 occur in some equivalence, then the equivalence assumes that F is splittable into F_1 and F_2. If F_1 or F_2 do not occur in some equivalence, they are 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.

1) Example 1: Join: Fig. 4 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. In order to do so, 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. 4 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. For our join example, it consists simply 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 parts in e_2. This gives us $F_1 = k : \text{count}(*), b_1 : \text{sum}(a_1)$, where 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 by applying the insights of Sec. II-A. This gives us $F_1^1 = k' : \text{count}(*), b_1' : \text{sum}(a_1)$ and

e_1

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

e_2

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

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

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

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

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

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

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

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

g_1	j_1	k'/c_1	b_1'
1	1	1	2
1	2	2	12
1	3	1	7

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

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

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

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

$e_6' := e_5 \bowtie^{F_1^1(\{\perp\}),c_1:1;-}_{j_1=j_2} e_2$

g_1	j_1	k'/c_1	b_1'	g_2	j_2	a_2
1	1	1	2	1	1	2
1	1	1	2	1	1	4
1	2	2	12	1	2	8
1	3	1	7	-	-	-
-	-	1^*	-	1	4	9

where

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

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

Fig. 4. Example for Eqvs. 10 and 12

$F_1^2 = k : \text{sum}(k'), b_1 : \text{sum}(b_1')$. 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 count$(*)$ the result of which is stored in k', we keep only one of them in Fig. 4 and call it k'/c_1. This finishes our preprocessing on the aggregation functions of the inner grouping operator. Its result is given as relation e_5 in Fig. 4. It consists of two tuples. 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 Sec. II-A, 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. 4. The final result of the left-hand side of Eqv. 10, calculated as $e_7 = \Gamma_{g_1,g_2;F_Y}(e_6)$, is given in Fig. 4. Note that this is the same as the result of the right-hand side (e_4).

2) Example 2: Full Outerjoin: The second example reuses the relations e_1 and e_2 given in Fig. 4. But this time, we calculate the full outerjoin instead of the inner join, and we

Eager/Lazy Groupby-Count

$$\Gamma_{G;F}(e_1 \bowtie_q e_2) \equiv \Gamma_{G;(F_2 \otimes c_1) \circ F_1^2}(\Gamma_{G_1^+;F_1^1 \circ (c_1:\text{count}(*))}(e_1) \bowtie_q e_2) \quad (10)$$

$$\Gamma_{G;F}(e_1 \mathbin{⟗}_q e_2) \equiv \Gamma_{G;(F_2 \otimes c_1) \circ F_1^2}(\Gamma_{G_1^+;F_1^1 \circ (c_1:\text{count}(*))}(e_1) \mathbin{⟗}_q e_2) \quad (11)$$

$$\Gamma_{G;F}(e_1 \mathbin{⟕}_q e_2) \equiv \Gamma_{G;(F_2 \otimes c_1) \circ F_1^2}(\Gamma_{G_1^+;F_1^1 \circ (c_1:\text{count}(*))}(e_1) \mathbin{⟕}_q^{F_1^1(\{\bot\}),c_1:1;-} e_2) \quad (12)$$

$$\Gamma_{G;F}(e_1 \bowtie_q e_2) \equiv \Gamma_{G;(F_1 \otimes c_2) \circ F_2^2}(e_1 \bowtie_q \Gamma_{G_2^+;F_2^1 \circ (c_2:count(*))}(e_2)) \quad (13)$$

$$\Gamma_{G;F}(e_1 \mathbin{⟗}_q e_2) \equiv \Gamma_{G;(F_1 \otimes c_2) \circ F_2^2}(e_1 \mathbin{⟗}_q^{F_2^1(\{\bot\}),c_2:1} \Gamma_{G_2^+;F_2^1 \circ (c_2:\text{count}(*))}(e_2)) \quad (14)$$

$$\Gamma_{G;F}(e_1 \mathbin{⟕}_q e_2) \equiv \Gamma_{G;(F_1 \otimes c_2) \circ F_2^2}(e_1 \mathbin{⟕}_q^{-;F_2^1(\{\bot\}),c_2:1} \Gamma_{G_2^+;F_2^1 \circ (c_2:\text{count}(*))}(e_2)) \quad (15)$$

Eager/Lazy Group-by

$$\Gamma_{G;F}(e_1 \bowtie_q e_2) \equiv \Gamma_{G;F_1^2}(\Gamma_{G_1^+;F_1^1}(e_1) \bowtie_q e_2) \quad (16)$$

$$\Gamma_{G;F}(e_1 \mathbin{⟗}_q e_2) \equiv \Gamma_{G;F_1^2}(\Gamma_{G_1^+;F_1^1}(e_1) \mathbin{⟗}_q e_2) \quad (17)$$

$$\Gamma_{G;F}(e_1 \mathbin{⟕}_q e_2) \equiv \Gamma_{G;F_1^2}(\Gamma_{G_1^+;F_1^1}(e_1) \mathbin{⟕}_q^{F_1^1(\{\bot\});-} e_2) \quad (18)$$

$$\Gamma_{G;F}(e_1 \bowtie_q e_2) \equiv \Gamma_{G;F_2^2}(e_1 \bowtie_q \Gamma_{G_2^+;F_2^1}(e_2)) \quad (19)$$

$$\Gamma_{G;F}(e_1 \mathbin{⟗}_q e_2) \equiv \Gamma_{G;F_2^2}(e_1 \mathbin{⟗}_q^{F_2^1(\{\bot\})} \Gamma_{G_2^+;F_2^1}(e_2)) \quad (20)$$

$$\Gamma_{G;F}(e_1 \mathbin{⟕}_q e_2) \equiv \Gamma_{G;F_2^2}(e_1 \mathbin{⟕}_q^{-;F_2^1(\{\bot\})} \Gamma_{G_2^+;F_2^1}(e_2)) \quad (21)$$

Eager/Lazy Count

$$\Gamma_{G;F}(e_1 \bowtie_q e_2) \equiv \Gamma_{G;(F_2 \otimes c_1)}(\Gamma_{G_1^+;c_1:\text{count}(*)}(e_1) \bowtie_q e_2) \quad (22)$$

$$\Gamma_{G;F}(e_1 \mathbin{⟗}_q e_2) \equiv \Gamma_{G;(F_2 \otimes c_1)}(\Gamma_{G_1^+;(c_1:\text{count}(*))}(e_1) \mathbin{⟗}_q e_2) \quad (23)$$

$$\Gamma_{G;F}(e_1 \mathbin{⟕}_q e_2) \equiv \Gamma_{G;(F_2 \otimes c_1)}(\Gamma_{G_1^+;(c_1:\text{count}(*))}(e_1) \mathbin{⟕}_q^{c_1:1;-} e_2) \quad (24)$$

$$\Gamma_{G;F}(e_1 \bowtie_q e_2) \equiv \Gamma_{G;(F_1 \otimes c_2)}(e_1 \bowtie_q \Gamma_{G_2^+;c_2:\text{count}(*)}(e_2)) \quad (25)$$

$$\Gamma_{G;F}(e_1 \mathbin{⟗}_q e_2) \equiv \Gamma_{G;(F_1 \otimes c_2)}(e_1 \mathbin{⟗}_q^{c_2:1} \Gamma_{G_2^+;c_2:\text{count}(*)}(e_2)) \quad (26)$$

$$\Gamma_{G;F}(e_1 \mathbin{⟕}_q e_2) \equiv \Gamma_{G;(F_1 \otimes c_2)}(e_1 \mathbin{⟕}_q^{-;c_2:1} \Gamma_{G_2^+;(c_2:\text{count}(*))}(e_2)) \quad (27)$$

Double Eager/Lazy

$$\Gamma_{G;F}(e_1 \bowtie_q e_2) \equiv \Gamma_{G;(F_1^2 \otimes c_2)}(\Gamma_{G_1^+;F_1^1}(e_1) \bowtie_q \Gamma_{G_2^+;c_2:\text{count}(*)}(e_2)) \quad (28)$$

$$\Gamma_{G;F}(e_1 \mathbin{⟗}_q e_2) \equiv \Gamma_{G;(F_1^2 \otimes c_2)}(\Gamma_{G_1^+;F_1^1}(e_1) \mathbin{⟗}_q^{c_2:1} \Gamma_{G_2^+;c_2:\text{count}(*)}(e_2)) \quad (29)$$

$$\Gamma_{G;F}(e_1 \mathbin{⟕}_q e_2) \equiv \Gamma_{G;(F_1^2 \otimes c_2)}(\Gamma_{G_1^+;F_1^1}(e_1) \mathbin{⟕}_q^{F_1^1(\{\bot\});c_2:1} \Gamma_{G_2^+;(c_2:\text{count}(*))}(e_2)) \quad (30)$$

$$\Gamma_{G;F}(e_1 \bowtie_q e_2) \equiv \Gamma_{G;(F_2^2 \otimes c_1)}(\Gamma_{G_1^+;c_1:\text{count}(*)}(e_1) \bowtie_q \Gamma_{G_2^+;F_2^1}(e_2)) \quad (31)$$

$$\Gamma_{G;F}(e_1 \mathbin{⟗}_q e_2) \equiv \Gamma_{G;(F_2^2 \otimes c_1)}(\Gamma_{G_1^+;c_1:\text{count}(*)}(e_1) \mathbin{⟗}_q^{F_2^1(\{\bot\})} \Gamma_{G_2^+;F_2^1}(e_2)) \quad (32)$$

$$\Gamma_{G;F}(e_1 \mathbin{⟕}_q e_2) \equiv \Gamma_{G;(F_2^2 \otimes c_1)}(\Gamma_{G_1^+;(c_1:\text{count}(*))}(e_1) \mathbin{⟕}_q^{c_1:1;F_2^1(\{\bot\})} \Gamma_{G_2^+;F_2^1}(e_2)) \quad (33)$$

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

$$\Gamma_{G;F}(e_1 \bowtie_q e_2) \equiv \Gamma_{G;(F_1^2 \otimes c_2) \circ (F_2^2 \otimes c_1)}(\Gamma_{G_1^+;F_1^1 \circ (c_1:\text{count}(*))}(e_1) \bowtie_q \Gamma_{G_2^+;F_2^1 \circ (c_2:\text{count}(*))}(e_2)) \quad (34)$$

$$\Gamma_{G;F}(e_1 \mathbin{⟗}_q e_2) \equiv \Gamma_{G;(F_1^2 \otimes c_2) \circ (F_2^2 \otimes c_1)}(\Gamma_{G_1^+;F_1^1 \circ (c_1:\text{count}(*))}(e_1) \mathbin{⟗}_q^{F_2^1(\{\bot\}),c_2:1} \Gamma_{G_2^+;F_2^1 \circ (c_2:\text{count}(*))}(e_2)) \quad (35)$$

$$\Gamma_{G;F}(e_1 \mathbin{⟕}_q e_2) \equiv \Gamma^2(\Gamma_{G_1^+;F_1^{1,1} \circ (c_1:\text{count}(*))}(e_1) \mathbin{⟕}_q^{F_1^{1,1}(\{\bot\}),c_1:1;F_2^{1,1}(\{\bot\}),c_2:1} \Gamma_{G_2^+;F_2^{1,1} \circ (c_2:\text{count}(*))}(e_2)) \quad (36)$$

Others

$$\Gamma_{G;F}(e_1 \ltimes_q e_2) \equiv \Gamma_{G;F}(e_1) \ltimes_q e_2 \quad (\mathcal{F}(q) \cap \mathcal{A}(e_1)) \subseteq G \quad (37)$$

$$\Gamma_{G;F}(e_1 \triangleright_q e_2) \equiv \Gamma_{G;F}(e_1) \triangleright_q e_2 \quad (\mathcal{F}(q) \cap \mathcal{A}(e_1)) \subseteq G \quad (38)$$

$$\Gamma_{G;F}(e_1 \mathbin{⟖}_{J_1 \theta J_2;\overline{F}} e_2) \equiv \Gamma_{G;(F_2 \otimes c_1) \circ F_1^2}(\Gamma_{G_1^+;F_1^1 \circ (c_1:\text{count}(*))}(e_1) \mathbin{⟖}_{J_1 \theta J_2;\overline{F}} e_2)). \quad (39)$$

$$\Gamma_{G;F}(e_1 \mathbin{⟖}_{J_1 \theta J_2;\overline{F}} e_2) \equiv \Gamma_{G;F_1^2}(\Gamma_{G_1^+;F_1^1}(e_1) \mathbin{⟖}_{J_1 \theta J_2;\overline{F}} e_2)) \quad (40)$$

$$\Gamma_{G;F}(e_1 \mathbin{⟖}_{J_1 \theta J_2;\overline{F}} e_2) \equiv \Gamma_{G;(F_2 \otimes c_1)}(\Gamma_{G_1^+;(c_1:\text{count}(*))}(e_1) \mathbin{⟖}_{J_1 \theta J_2;\overline{F}} e_2)) \quad (41)$$

Fig. 3. Equivalences

apply Eqv. 12. The according expressions are now 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 $e_1 \fullouterjoin_{j_1=j_2} e_2$ is given in e_3', where we denote NULL by '-'. We can reuse all the different aggregation vectors derived in the previous example. The only new calculation that needs to be done is the one for the default values for the full outerjoin on the right-hand side of Eqv. 12. Eqv. 12 defines default values in case a tuple t from e_2 does not find a join partner from the other side. All c_1 values of orphaned e_2 tuples become 1. Further, $F_1^1(\{\bot\})$ evaluates to 1 for k (count(*) on a relation with a single element), and NULL for a_2, since SQL's sum returns NULL for sets which contain only NULL values. Thus prepared, we can calculate the left-hand side of Eqv. 12 via e_5, which is the same as in the previous example e_6', which now uses a full outerjoin with default, and, finally, e_7', which shows the same result as e_4'.

3) Remarks: The main equivalences are those under the heading Eager/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 like inner join and full outerjoin, deriving one from the other is simple. For non-commutative operators like the left outerjoin, an additional proof is necessary. Now, instead of pushing the grouping operator only into one argument, we can combine both equivalences to push it into both arguments. The resulting equivalences are given under the heading Eager/Lazy Split. The equivalences between these two blocks are specializations in case an aggregation vector F only accesses attributes from 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/Lazy Group-By, Eager/Lazy Count and Double Eager/Lazy. The last block of equivalences, 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 $\overline{F}$. 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.

B. Eliminating the Top Grouping

We wish to eliminate a top grouping from some expression of the form $\Gamma_{G,F}(e)$ for some aggregation vector $F = (b_1 : \text{agg}_1(a_1), \ldots, b_k : \text{agg}_k(a_k))$. Clearly, this is only possible if G is a key 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

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

if we define $\hat{F}$ to calculate the result of some aggregate function for single values, i.e., $\hat{F} := (b_1 : \text{agg}_1(\{a_1\}), \ldots, b_k : \text{agg}_k(\{a_k\}))$, and $C = G \cup \{b_1, \ldots, b_k\}$.

Remark. In SQL, a declaration of a primary key or a uniqueness constraint implies not only a key but also that the relation is duplicate-free.

IV. Dynamic Programming

A. Plan Generation Basics

We briefly repeat the basics of a bottom-up plan generator. Fig. 5 shows the basic structure of a typical dynamic programming-based plan generator. Its input consists of three major pieces: the set of relations to be joined, the set of operators to be used for this, and a hypergraph representing the query graph. Clearly, the relations and the operators are derived from the initial SQL query in a straightforward manner. The hypergraph is constructed by a conflict detector [1]. It encodes possible reordering conflicts as far as possible into the hypergraph. This is necessary since inner joins and outer joins are not freely reorderable.

The major data structure used is the $DPTable$, which stores (an) optimal plan(s) for a given set of relations. The basic algorithm in Fig. 5 uses a single plan per $DPTable$ entry. Later on, multiple plans exists per $DPTable$ entry.

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

Definition 3: Let $H = (V, E)$ be a hypergraph and S_1, S_2 two subsets of V. (S_1, S_2) is a *csg-cmp-pair* (*ccp* for short) if the following three conditions hold:

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

An efficient enumerator for csg-cmp-pairs has been proposed in [15].

The third component (Line 5) is an applicability test for operators. It builds upon the conflict representation and checks whether some operator $\circ_p$ can be safely applied. This is necessary since it is not possible to exactly cover all conflicts within a hypergraph representation of the query [1].

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

Subsequently, we will see that all components except for the last one can remain unmodified if we introduce the capability of pushing grouping operators down. Thus, our approach is minimally invasive.

B. Applying Eager Aggregation

Before we continue with the first plan generator, we introduce the routine OpTrees (Fig. 6) that is utilized by all plan generators in this paper. Its arguments are two join trees T_1 and T_2, and a join operator $\circ_p$. The result consists of a set of at most four trees which join T_1 and T_2, including all possible variants of eager aggregation.

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

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

Fig. 5. Basic DP Algorithm

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

One situation that requires some care is when we create a join tree containing all the relations in our query, which is equivalent to $S = R$, where R is the set of all relations. Then we have to add another grouping on top of $\circ_p$ if and only if the grouping attributes do not comprise a key (cf. Sec. III-B). This is checked by calling NEEDSGROUPING, which is listed in Figure 7.

The next tree is the one that groups the left argument before the join. In order to do so, we have to make sure that the corresponding transformation is valid. This check is accomplished by calling the subroutine VALID, which implements the equivalences presented in Sec. III. Additionally, we have to avoid the case in which the grouping attributes G_i^+ form a key for the set S_i, with $i \in \{1, 2\}$, because then the grouping would be a waste. And again, if necessary, we have to add a grouping on top.

Once the routine terminates, the returned set *Trees* contains up to four different join trees which are depicted in Fig. 8. Note that the introduction of OPTREES only serves the purpose of increasing the readability of the following algorithms and should not be included in a real implementation since it produces plans that not all of the subsequent algorithms will need.

```
OPTREES(T_1, T_2, ∘_p)
1  S_1 = T(T_1)
2  S_2 = T(T_2)
3  S = S_1 ∪ S_2
4  Trees = ∅
5  NewTree = (T_1 ∘_p T_2)
6  if S == R ∧ NEEDSGROUPING(G, NewTree)
7     NewTree = (Γ_G(NewTree))
8  Trees.insert(NewTree)
9  NewTree = Γ_{G_1^+}(T_1) ∘_p T_2
10 if VALID(NewTree) ∧ NEEDSGROUPING(G_1^+, NewTree)
11    if S == R ∧ NEEDSGROUPING(G, NewTree)
12       NewTree = (Γ_G(NewTree))
13    Trees.insert(NewTree)
14 NewTree = T_1 ∘_p Γ_{G_2^+}(T_2)
15 if VALID(NewTree) ∧ NEEDSGROUPING(G_2^+, NewTree)
16    if S == R ∧ NEEDSGROUPING(G, NewTree)
17       NewTree = (Γ_G(NewTree))
18    Trees.insert(NewTree)
19 NewTree = Γ_{G_1^+}(T_1) ∘_p Γ_{G_2^+}(T_2)
20 if VALID(NewTree)
      ∧ NEEDSGROUPING(G_1^+, NewTree)
      ∧ NEEDSGROUPING(G_2^+, NewTree))
21    if S == R ∧ NEEDSGROUPING(G, NewTree)
22       NewTree = (Γ_G(NewTree))
23    Trees.insert(NewTree)
24 return Trees
```

Fig. 6. OPTREES

```
NEEDSGROUPING(G, T)
1  if ∃k ∈ κ(T), k ⊆ G ∧ the result of T is duplicate-free
2     return FALSE
3  else
4     return TRUE
```

Fig. 7. NEEDSGROUPING

C. Enumerating the Complete Search Space

Our goal is to make use of eager aggregation and the equivalences presented in the previous section in a plan generator like the basic DP-algorithm described above.

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.

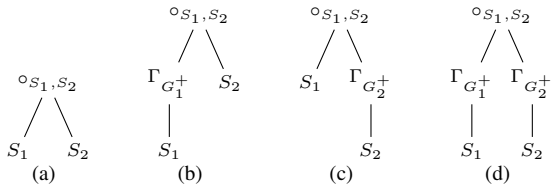


Fig. 8. Possible trees for grouping and join

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

INSERTTOPLEVELPLAN(S, T)
1  if DPTable[S] == ∅ ∨ COST(T) < COST(DPTable[S])
2    DPTable[S] = ∅
3    DPTable.APPEND(T)
```

Fig. 9. BUILDPLANSALL

To do this, we change the dynamic programming table in such a way that it can not only contain one optimal join tree for every set $S \subseteq R$, but also a list of possible trees. Figure 9 shows the routine BUILDPLANSALL, which is derived from the routine BUILDPLANS depicted in Figure 5 and illustrates the necessary modifications.

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

Eventually, we face the situation where $S = R$ holds and we need to build a join tree for the complete query. At this point, we call another subroutine named INSERTTOPLEVELPLAN. Inside this routine, we compare the join trees for S to find the one with minimal costs because there are no subsequent join operators that need to be taken into account. Before we can do this, we have to decide whether we need a top-level grouping by calling NEEDSGROUPING (cf. Figure 7). In contrast to the other relation sets, we do not have to keep a list of trees for R, but only the best tree found so far and replace it if a better one is found.

Obviously, the runtime complexity of this algorithm is $O(2^{2n-1}\#\text{ccp})$ for n relations if #ccp denotes the number of csg-cmp-pairs for the query.

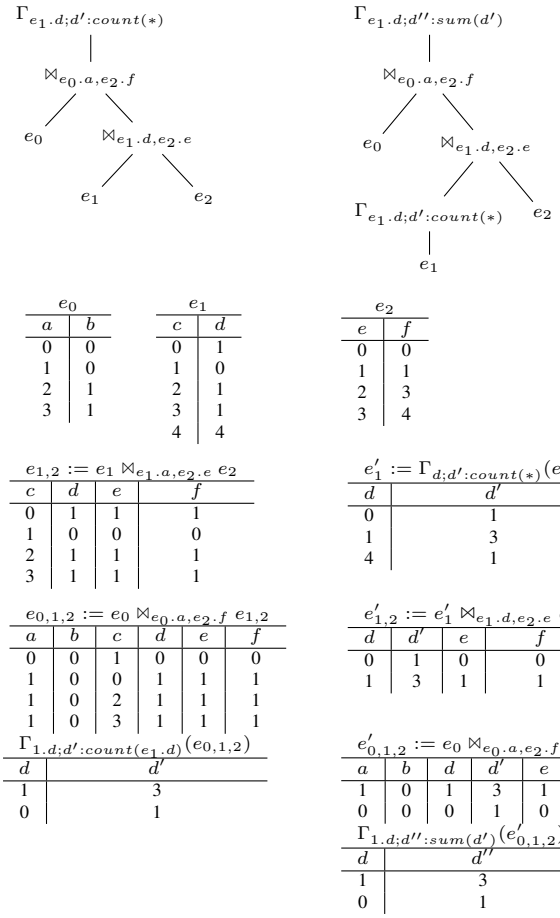
D. A First Heuristic

In this subsection, we present a plan generator that is capable of applying eager aggregation without the exponential overhead induced by the algorithm depicted in Figure 9. The downside of this is the fact that the new less complex plan generator does not guarantee an optimal solution any longer.

The major difficulty we face in incorporating eager aggregation into a DP-based plan generator is that Bellman's Principle of Optimality is no longer valid. If we push a grouping operator into one or both arguments of a join operator, this can influence the costs of subsequent join operations. This means that it might be necessary to use a suboptimal join tree for a set S_1 to construct an optimal solution for some set $S \supset S_1$.

```
BUILDPLANSH1(S1, S2, ∘p)
1  for each T ∈ OPTREES(DPTable[S1], DPTable[S2], ∘p)
2    if COST(T) < COST(DPTable[S1 ∪ S2])
3      DPTable[S1 ∪ S2] = T
```

Fig. 10. BUILDPLANSH1



e_0	
a	b
0	0
1	0
2	1
3	1

e_1	
c	d
0	1
1	0
2	1
3	1
4	4

e_2	
e	f
0	0
1	1
2	3
3	4

$e_{1,2} := e_1 \bowtie_{e_1.a,e_2.e} e_2$

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

$e'_1 := \Gamma_{d;d':count(*)}(e_1)$

d	d'
0	1
1	3
4	1

$e_{0,1,2} := e_0 \bowtie_{e_0.a,e_2.f} e_{1,2}$

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

$e'_{1,2} := e'_1 \bowtie_{e_1.d,e_2.e} e_2$

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

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

d	d'
1	3
0	1

$e'_{0,1,2} := e_0 \bowtie_{e_0.a,e_2.f} e'_{1,2}$

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

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

d	d''
1	3
0	1

Fig. 11. Exemplary query with alternative join trees

That is because the higher costs of the non-optimal subplan under certain circumstances are compensated by cost savings for the subsequent joins.

Figure 10 again shows a modified version of BUILDPLANS. We refer to the resulting plan generator as our first heuristic or H1. The modified routine is called BUILDPLANSH1. It serves to demonstrate the problems that arise from the violation of Bellman's Principle of Optimality.

The only difference to the basic version of BUILDPLANS we presented in Section IV-A is that the new algorithm makes use of OPTREES to find all possible trees for the current csg-cmp-pair. For each of them the cost function is called to compute the combined costs for the join and the groupings contained in the tree, if any. If the costs are lower than those of an existing plan or if this is the first plan for the current set of relations, the plan is added to the DP-table. In summary, H1 records only the single cheapest plan for every plan class.

To clarify why this approach can lead to problems, Fig. 11 provides a sample query. At the top of the figure there are two equivalent operator trees. Both of them involve a grouping operation. The left one does not make use of eager aggregation, so the grouping remains at the top of the tree and is evaluated after all join operations. In the tree on the right side, a grouping operator has been pushed down into the left argument of $\bowtie_{e_1.d,e_2.e}$. Note how the aggregation vector of the original grouping operator at the top of the tree is adjusted according to our observations from Section III. That is, we now have to sum up the values created by the other grouping operator to get the originally intended $count(*)$. Below the two operator trees, there are instances of the three relations e_0, e_1 and e_2, and all the intermediate results for both operator trees.

$C_{out}(e_0) = C_{out}(e_1) = C_{out}(e_2) = 0$	
$C_{out}(e_{1,2}) = 4$	$C_{out}(e'_1) = 3$
$C_{out}(e_{0,1,2}) = 8$	$C_{out}(e'_{1,2}) = 5$
$C_{out}(\Gamma(e_{0,1,2})) = 10$	$C_{out}(e'_{0,1,2}) = 7$
	$C_{out}(\Gamma(e'_{0,1,2}) = 9$

TABLE I. COSTS FOR INTERMEDIATE RESULTS

Table I contains the costs of all subexpressions of both operator trees, where, for simplicity, we used the cost function C_{out}, which simply sums up the intermediate result sizes:

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

According to the definition of C_{out}, scanning the base relations does not cause any costs at all, which is reflected in the first line of Table I. (Note that the scan costs would be the same constant in both plans anyway.) Beginning on the second line, the left (right) column contains the cost of the intermediate results of the left (right) plan of Fig. 11.

Let us now go through our heuristic. It will decide against early aggregation of relation e_1 because the combined costs for the grouping and the following join operation are higher than the costs for joining without prior grouping. Taking a closer look at the following lines in our table, we see that the costs for joining $e_{1,2}$ with e_0 amount to 8, whereas the right column states a value of 7 for the join between $e'_{1,2}$ and e_0. For the total costs of the query, we notice the same cost difference: the left tree causes costs of 10, the right one only 9. This means the tree that is eliminated by our naive plan generator is in fact less expensive than the other one.

The reason for this behaviour can be seen in Figure 11. The early grouping of relation e_1 causes additional costs of 3, but it also reduces the cardinality of the following expressions $e'_{1,2}$ and $e'_{0,1,2}$ compared to $e_{1,2}$ and $e_{0,1,2}$. The additional costs for the first grouping operation are therefore compensated by the reduced cardinalities and costs of the following expressions. Considering only the costs of expression $e'_{1,2}$, this benefit is not obvious because it becomes visible only further up in the tree.

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

Looking back at the values for $e'_{0,1,2}$ in Figure 11, we can see that the final grouping is not necessary to produce the same result as the left join tree. Instead, a projection on the attribute set $\{e_1.d, d'\}$ suffices because the functional dependency $e_1.d \rightarrow \mathcal{A}(e'_{0,1,2})$ holds, i.e., $e_1.d$ is a key for $e'_{0,1,2}$ and the attribute d' already contains the correct value for the original aggregate function $count(*)$. We can therefore leave out the final grouping and replace it by a much cheaper duplicate-preserving projection $\Pi_{e_1.d,d'}$. As our cost function does not take projection costs into account, we end up with a cost value of 7 for the tree applying eager aggregation, in contrast to a value of 10 for the other tree.

These findings lead to the conclusion that it is not sufficient to "locally" assess the profitability of eager aggregation for one join operation, as described above, if we want to consider the whole search space. Still, this approach can be used as a simple heuristic producing only a moderate overhead on top of dynamic programming and at the same time exploiting at least some of the potential benefits of eager aggregation.

E. Improving the Heuristic

As we have seen in the previous subsection, the routine BUILDPLANSH1 tends to discard trees applying eager aggregation even in cases where it might be beneficial because the accumulated costs of the aggregation and the join are higher than those of the join alone.

It is therefore possible to improve the heuristic by making this cost comparison less strict and thereby enabling the plan generator to prefer plans that are "more eager" even though they might cause slightly higher costs locally. For this purpose, we introduce the simple notion of the *eagerness* of a plan, which is defined as follows:

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

The eagerness of a join tree T is simply defined as the number of grouping operators that are a direct child of the topmost join operator. Figure 12 shows the pseudocode for the routine BUILDPLANSH2, which exploits eagerness.

The main difference to BUILDPLANSH1 is the new subroutine COMPAREADJUSTEDCOSTS, which is called from line 2. It takes two join trees and compares the costs of the two, whereby it adjusts the costs of the less eager tree using a constant factor F. The value of F determines the degree to which more eager plans are preferred when compared to less eager plans. If the eagerness of the two join trees passed to COMPAREADJUSTEDCOSTS is equal or if the trees form a plan for the whole query, no cost adjustment is applied. In the evaluation, we experiment with different values for F.

F. Optimality Preserving Pruning

In Subsection IV-D we showed that it is not possible to decide whether or not a particular subtree is part of the final solution solely based on its costs. Instead, there are some

```
BUILDPLANSH2(S1, S2, ∘p)
1  for each T ∈ OPTREES(DPTable[S1], DPTable[S2], ∘p)
2    if COMPAREADJUSTEDCOSTS(T, DPTable[S1 ∪ S2])
3      DPTable[S1 ∪ S2] = T

COMPAREADJUSTEDCOSTS(T1, T2)
1  if T is top-level plan ∨
     EAGERNESS(T1) == EAGERNESS(T2)
2    return COST(T1) < COST(T2)
3  if (EAGERNESS(T1) < EAGERNESS(T2))
4    return (F × COST(T1)) < COST(T2)
5  elseif (EAGERNESS(T1) > EAGERNESS(T2))
6    return COST(T1) < (F × COST(T2))
```

Fig. 12. BUILDPLANSH2 and COMPAREADJUSTEDCOSTS

more properties we have to check before we can safely discard suboptimal trees.

As we have seen in Subsection IV-C, keeping all possible trees in the solution table guarantees an optimal solution but, on the other hand, causes such a big overhead that it is impractical for most queries. This leads us to the question if we can find a way to reduce the number of DP-table entries and still preserve the optimality of the resulting solution.

The first observation we made for the example query shown in Figure 11 was that we have to take the cardinalities of intermediate results into account. That is because subtrees with suboptimal costs caused by the introduction of additional grouping operators can in turn produce smaller results and thereby lower the costs of subsequent operations.

In addition to that, we discovered that the functional dependencies that hold for the result of an intermediate join expression can influence the costs of the final join tree. These functional dependencies are in turn influenced by the grouping operators present in the expression.

As a result of these findings, we can define *dominance* of one tree over another tree by means of three criteria:

Definition 4: A join tree T_1 dominates another join tree T_2 for the same set of relations if all of the following conditions hold:

- $Cost(T_1) \leq Cost(T_2)$
- $|T_1| \leq |T_2|$
- $FD^+(T_1) \supseteq FD^+(T_2)$.

We can safely discard any join tree T_2 that is dominated by another join tree T_1. Note that the third item makes use of the closure of the functional dependencies, denoted by FD^+, that hold in T_1 and T_2, respectively. Since the computation and comparison of these two sets is expensive, this condition can be weakened in an actual implementation by comparing the sets of candidate keys instead. Figure 13 shows the routine PRUNEDOMINATEDPLANS, which checks these three conditions.

The routine expects as arguments a set of relations S and a join tree T for this set. The loop beginning in line 1 iterates through the existing join trees for S taken from the DP-table

```
PRUNEDOMINATEDPLANS(S, T)
1  for T_old ∈ DPTable[S]
2    if COST(T_old) ≤ COST(T) ∧ |T_old| ≤ |T|
       ∧ FD+(T_old) ⊇ FD+(T)
3      return
4    if COST(T_old) ≥ COST(T) ∧ |T_old| ≥ |T|
       ∧ FD+(T_old) ⊆ FD+(T)
5      discard T_old
6  DPTable[S].APPEND(T)
```

Fig. 13. PRUNEDOMINATEDPLANS

```
BUILDPLANSPRUNE(S1, S2, ∘p)
1  S = S1 ∪ S2
2  for each T1 ∈ DPTable[S1]
3    for each T2 ∈ DPTable[S2]
4      for each T ∈ OPTREES(T1, T2, ∘p)
5        if S == R
6          INSERTTOPLEVELPLAN(S, T)
7        else
8          PRUNEDOMINATEDPLANS(S, T)
```

Fig. 14. BUILDPLANSPRUNE

and compares each of them with the new tree T. If there is an existing tree T_{old} with lower or equal costs and lower or equal cardinality than T and the functional dependencies holding for the new tree are a subset of the ones for T_{old}, T cannot result in a better solution than T_{old}. Therefore, the routine returns without adding T to the tree list for S.

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

This pruning routine is called by BUILDPLANSALL for every new join tree found for a set $S \neq R$. Note that there is no need to prune in case of $S = R$ because then the list contains only one tree anyway. Summarizing, this gives us the routine BUILDPLANSPRUNE depicted in Figure 14.

V. EVALUATION

We evaluate the algorithms experimentally with respect to runtime and plan optimality. For our experiments, we extended the DP-based plan generator DPhyp [16] and generated 10,000 operator trees each for a certain number of relations from three to twenty. Therefore, we first generated random binary trees using the unranking procedure proposed by Liebehenschel [17]. Next, we randomly attached join operators to the internal node and relations to the leaves. Then, the attributes for equality join predicates and grouping are randomly selected. Finally, random cardinalities and selectivities are generated.

A. The Gain

First of all, we demonstrate the potential benefit that arises from the application of eager aggregation in terms of plan quality. Fig. 15 shows the average total plan cost achieved without

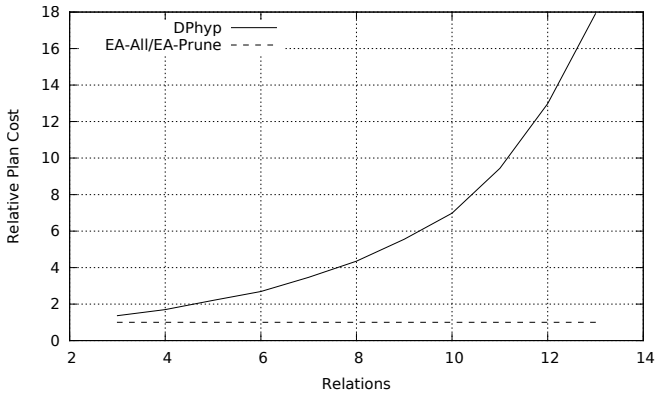


Fig. 15. Plan Cost DPhyp and EA-Prune

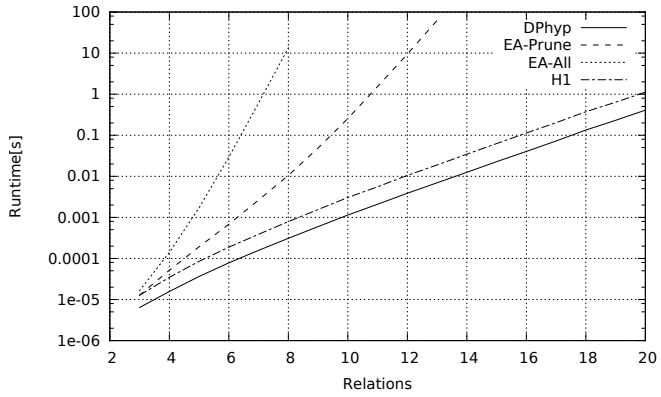


Fig. 16. Runtime EA-Prune, EA-All and H1

eager aggregation in relation to the values produced by EA-Prune/EA-All, i.e., the complete plan generators with/without pruning. As we have stated in the previous section, our pruning criterion does not affect plan optimality. The values of the two algorithms are therefore identical.

As can be seen in the graphic, the plan quality for queries with three relations is nearly equal for the two plan generators, but EA-Prune is already slightly ahead. As the number of relations increases, the cost difference also increases. The curves stop at 13 relations, where the plans produced by DPhyp are on average 18 times as expensive as the ones produced by EA-Prune. However, there are some extreme outliers. The biggest cost difference was observed for a query with 10 tables where the plan produced by DPhyp was 17,500 times as expensive as the one achieved with eager aggregation.

B. The Price

These gains come at the price of increased runtime and memory usage. Fig. 16 shows the runtime for DPhyp, the two complete enumeration algorithms EA-Prune and EA-All and our first heuristic H1. Note that the y-axis is scaled logarithmically. The curves for EA-Prune and EA-All stop at 8 and 13 relations, respectively, since running them with 10,000 different input queries for up to 20 relations was not feasible because of their extremely long runtimes. As can be seen in the figure, EA-Prune takes more than one second for a query with 11 relations. If pruning is not applied, this threshold is reached with only 7 relations. DPhyp, on the other hand, stays below one second even for 20 relations. H1 differs from DPhyp by an almost constant factor of 2.6 on average. This leads us to the conclusion that the complete enumeration of the search space including eager aggregation is only practical for small to medium queries, even if pruning is applied.

C. The Details

Now we take a closer look at the heuristic algorithms. First of all, we are interested in how close to the optimal solution they actually get. Fig. 17 compares the total cost achieved by H1, H2 and the complete algorithm with pruning. Again, all values are relative to the ones produced by EA-Prune. For the second (improved) heuristic H2, the figure contains four curves, each with a different value for the tolerance factor F. These values are drawn from a wide range of alternative

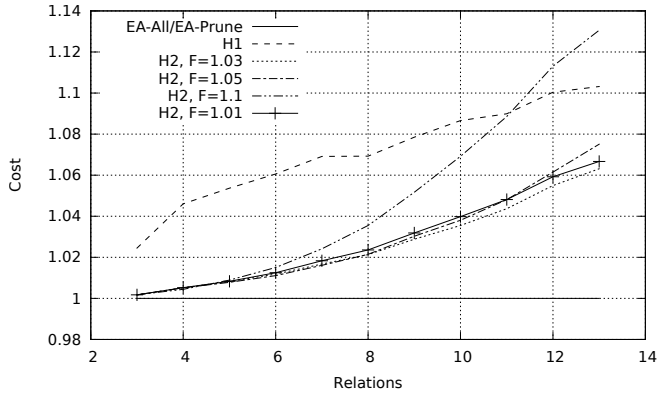


Fig. 17. Plan Cost Heuristics and EA-Prune

tolerance factors that we used in the course of our experiments. They serve to show the influence of different factors on the resulting plan quality.

None of the heuristic plan generators produces optimal costs for every query, but all of them are significantly closer to optimality than DPhyp. Out of the plan generators that were run for this experiment, H2 with a tolerance factor of 1.03 is the best as its plan quality is closest to that produced by EA-Prune. For 13 relations, the plans produced by H2 are on average only 7 percent more expensive than the optimal solution. The largest factor we observed for H1 is 10.3, and for H2 it is 9.7 ($F = 1.03$), both resulting from queries with 13 relations.

The runtimes of H1 and H2 are given in Fig. 18. In many cases H2 is slightly faster. The reason for this is that H2 has a tendency to apply eager aggregation more often than H1, which has an influence on the key constraints that hold in the produced subplans. While H2 has to do more work for every plan it considers because it has to determine the eagerness of the plan and calculate the adjusted cost, it considers fewer plans because pushing a grouping often makes a group-by further up in the operator tree obsolete. The latter is due to the fact that the grouping attributes become a key.

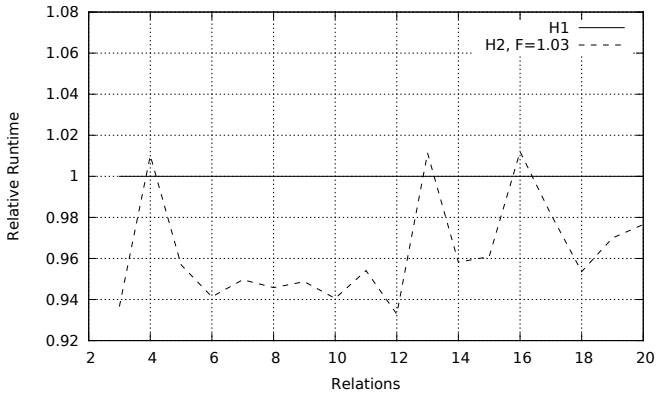


Fig. 18. Runtime H1 and H2

D. TPC-H Queries

Table II shows a comparison of DPhyp and our new algorithms with respect to optimization times and optimized plan costs for the example query from Section I (Ex) and three selected TPC-H queries (Q3, Q5, Q10). Query statistics were taken from a scale factor 1 instance of TPC-H. Since only Ex contains an outer join, it is important to stress that the presence of outer joins does not increase the complexity of EA-Prune or any of the algorithms presented in this paper.

Among the listed queries, Ex benefits most from eager aggregation, which is also reflected by the execution times we observed on different existing systems (see Section I). TPCH-Q5, on the other hand, provides the smallest possible gain.

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

TABLE II. OPTIMIZATION TIME AND PLAN COST FOR TPC-H QUERIES

VI. CONCLUSION

We presented a complete set of equivalences that allows us to push grouping into inner joins, left outerjoins, full outerjoins, semijoins, antijoins, and groupjoins. Further, we introduced four novel algorithms to integrate the exploitation of these equivalences within a state-of-the-art dynamic programming-based plan generator. Both, a simple complexity analysis and the experiments indicate that the complete enumeration of the extended search space is possible for only up to 7 relations. A newly introduced optimality preserving pruning technique allows to extend this bound to 10. Beyond that, only heuristic approaches are possible. One of them, H2, produces competitive plans which are on average only 7% worse than the optimal plan. However, some extreme outliers exist where the plan produced by H2 is a factor of 9.7 worse than the optimal plan. Thus, two directions for future research are to discover better heuristic algorithms and to develop even more effective optimality preserving pruning techniques.

ACKNOWLEDGMENT

We thank Simone Seeger for her help preparing the manuscript and the anonymous referees for their helpful comments.

REFERENCES

[1] G. Moerkotte, P. Fender, and M. Eich, "On the correct and complete enumeration of the core search space," in *ACM SIGMOD*, 2013, pp. 493–504.

[2] A. Shanbhag and S. Sudarshan, "Optimizing join enumeration in transformation-based query optimizers," *Proc. of the VLDB Endowment (PVLDB)*, vol. 7, no. 12, pp. 1243–1254, 2014.

[3] S. Chaudhuri and K. Shim, "Including group-by in query optimization," in *VLDB*, 1994, pp. 354–366.

[4] W. Yan, "Rewriting optimization of sql queries containing group-by," Ph.D. dissertation, University of Waterloo, 1995.

[5] W. Yan and P.-A. Larson, "Performing group-by before join," Dept. of Computer Science, University of Waterloo, Canada, Technical Report CS 93-46, 1993.

[6] ——, "Performing group-by before join," in *IEEE ICDE*, 1994, pp. 89–100.

[7] ——, "Eager aggregation and lazy aggregation," in *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 1995, pp. 345–357.

[8] ——, "Interchanging the order of grouping and join," Dept. of Computer Science, University of Waterloo, Canada, Technical Report CS 95-09, 1995.

[9] A. K. et al., "Processing in the hybrid OLTP & OLAP main-memory database system HyPer," *IEEE Data Engineering Bulletin*, vol. 36, no. 2, pp. 41–47, 2013.

[10] S. Cluet and G. Moerkotte, "Efficient evaluation of aggregates on bulk types," in *Int. Workshop on Database Programming Languages*, 1995.

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

[12] G. Paulley, "Exploiting functional dependence in query optimization," Ph.D. dissertation, University of Waterloo, 2000.

[13] C. Galindo-Legaria and M. Joshi, "Orthogonal optimization of subqueries and aggregation," in *Proc. of the ACM SIGMOD Conf. on Management of Data*, 2001, pp. 571–581.

[14] M. Eich and G. Moerkotte, "Dynamic programming: The next step," University of Mannheim, Tech. Rep., 2014.

[15] G. Moerkotte and T. Neumann, "Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products," in *VLDB*, 2006, pp. 930–941.

[16] ——, "Dynamic programming strikes back," in *ACM SIGMOD*, 2008, pp. 539–552.

[17] J. Liebehenschel, "Lexicographical generation of a generalized dyck language," University of Frankfurt, Tech. Rep. 5/98, 1998.