Centrum voor Wiskunde en Informatica
REPORT*RAPPORT*

Complexity of transformation-based optimizers and duplicate-free generation of alternatives

J. Pellenkoft, C.A. Galindo-Legaria and M.L. Kersten

Computer Science/Department of Algorithmics and Architecture

CS-R9639 1996

Report CS-R9639
ISSN 0169-118X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

Complexity of Transformation-based Optimizers and Duplicate-free Generation of Alternatives

Arjan Pellenkoft[1,2]
arjan@cwi.nl

César A. Galindo-Legaria[1]
cesarg@microsoft.com

Martin Kersten[2]
mk@cwi.nl

[1] *Microsoft*
One Microsoft Way, Redmond, WA 98052-6399 USA

[2] *CWI*
P. O. Box 94079, 1090 GB Amsterdam, The Netherlands

Abstract

Transformation-based optimizers that explore a search space exhaustively usually apply all possible transformation rules on each alternative, and stop when no new information is produced. In general, different sequences of transformations may end up deriving the same element. The optimizer must detect and discard these duplicate elements.

In this paper we consider two questions: How many duplicates are generated? And then, how can it be avoided?

For the first question, our analysis shows that as queries get larger, the number of duplicates encountered is several times that of the new elements. And even for small queries, duplicates are generated more often than new elements. For the second question, we describe a technique to avoid generating duplicates, based on keeping track of (a summary of) the derivation history of each element in the search space.

CR Subject Classification(1991): [H.2.4] Database Systems, Query Processing; [H.3.3] Information Search and Retrieval; [G.2.2] Graph Theory; [F.2.2] Non-numerical algorithms and problems.

Keywords and Phrases: Transformation-based optimization, Generation of duplicates, Complexity.

1. Introduction

Transformation-based query optimizers have been proposed as a modular, extensible tool to incorporate easily new operators and execution alternatives in the query optimization process [GCD+94]. But note that, in general, the same execution plan can be derived through different sequences of transformation rules, leading to *duplicates*. Duplicates are not an issue for strategies that explore only a small fragment of the search space, especially if the elements are generated probabilistically, because it is unlikely that the same element be generate twice [SG88, IK90, IK91, GLPK94] . However, for optimizers that generate the complete space of alternatives, dealing with duplicates becomes crucial.

To generate the complete space of alternatives, the naive algorithm is to maintain a set of *visited* plans. All transformation rules are applied on plans visited, adding the results to the set if they are new. When no new plans can be generated, we have explored the complete search space (provided the set of transformation rules is complete). Every time a duplicate plan is found, the time to generate it and then to find it in the set of visited plans is part of the overhead of a naive transformation-based search algorithm.

How expensive is this overhead? How frequently are we generating duplicates? Consider the following simple graph model to get a sense of the dimension of the problem. The number of duplicates generated depends on the the size of the search space, n, and the number of neighbors, b_i, of each state s_i (a state s_j is a neighbors of s_i if there is a transformation rule that generates s_j from s_i). Trying each transformation results in generating: $\sum_{i=1}^{n} b_i$ states. Assuming the number of neighbors for each state is the same ($b = b_1 = \ldots = b_n$) we get $b * n$ generated states. Since there are only n states, the number of duplicates generated is $n*(b-1)$. Only 1 out of every b plans generated is new, and $\left(1 - \frac{1}{b}\right)$ of the plans generated — i.e. most of them as b increases — are duplicates. A considerable efficiency improvement can be achieved by avoiding the generation of those duplicates. We refine our analysis of duplicates later on, for a more realistic optimization framework.

In particular, in a query joining 5 relations, each operator tree has 4 to 7 neighbors, using the "standard" associativity and commutativity rules. If we explore the space of alternatives for this query exhaustively, detecting and discarding duplicates, then we'll be discarding between 75% and 86% of all plans we generate! For larger queries, the number of neighbors of each plan increases, and so does the proportion of duplicates.

In this paper we show that it is possible to generate the space efficiently — i.e. without generating duplicates — within the framework of an extensible transformation-based optimizer. The technique described is based on conditioning the application of rules on the derivation history of an element. Each plan maintains a set of rules that can still be applied on it without generating duplicates. We illustrate the approach with a few restricted cases of join reordering.

The paper is organized as follows. In Section 2 we describe the generation of alternatives in Volcano [GM93], which is representative of transformation-based query optimizers. Section 5 describes how, within the same framework, alternative join orders can be generated without duplicates, for acyclic query graphs. Finally our conclusions are given in Section 6.

2. Optimizer Framework

To show how duplicate free, transformation based generation of alternatives can be incorporated into modern query optimizers, the exploration algorithm of Volcano-type optimizers is described in some detail. A key component in these optimizers is a *MEMO-structure* that efficiently stores information about all alternatives explored [GCD+94].

To generate a bushy space the following rule-set if used. This set was also used in [BMG93, IW87, IK91, Kan91].

Rule set **RS-B0**:

- Right Associativity: $(A \bowtie B) \bowtie C \rightsquigarrow A \bowtie (B \bowtie C)$.
- Left Associativity: $A \bowtie (B \bowtie C) \rightsquigarrow (A \bowtie B) \bowtie C$.
- Commutativity: $A \bowtie B \rightsquigarrow B \bowtie A$.

The set is redundant, because we can drop Right Associativity (or Left Associativity) and still generate the same space. We use here the minimal set **RS-B1**, which contains only Left Associativity and Commutativity.

For generating left linear join trees for completely connected queries a rule set that based on [SG88] is used.

Rule set **RS-L1**:

- Swap: $(A \bowtie B) \bowtie C \rightsquigarrow (A \bowtie C) \bowtie B$.
- Bottom Commutativity: $B_1 \bowtie B_2 \rightsquigarrow B_2 \bowtie B_1$, for base tables B_1, B_2.

2.1 MEMO-structure

A memory efficient representation of the search space is used to ameliorate the combinatorial explosion of alternative join orders. For a *completely connected* join query with n relations the number of alternative ordered bushy join trees is known to be $\frac{(2n-2)!}{(n-1)!}$ [LVZ93, GLPK95]. A completely connected query of 7 relations then already leads to 665280 alternative evaluation orders.

The main idea of the MEMO-structure is to avoid replication of subtrees by using *shared copies* only. It is organized as a network of *equivalence classes* (or simply classes). Each class is a set of operators which all generate the same (intermediate) result. The inputs for the operators are classes which can be interpreted as "any operator of that class can be used as input".

For example, the MEMO-structure for the 12 alternatives of a query whose fully connected graph is $Q = \{A-B, A-C, B-C\}$ is shown in Figure 1. It has 7 equivalence classes, namely "abc", "ab", "bc", "ac", "a", "b", "c", with the first class containing 6 join operators. For convenience, we name the classes with the relations that they combine. The first join operator of class "abc" has as input the classes "ab" and "c".

An operator tree is obtained from a MEMO-structure by choosing a specific operator at each level. For example, the tree to solve query Q, shown in Figure 2, is extracted from the MEMO-structure in Figure 1 by *always selecting the first operator from a class.*

$$
\begin{array}{ll}
\text{abc} & = [ab] \bowtie [c]; [a] \bowtie [bc]; [b] \bowtie [ac]; [c] \bowtie [ab]; [bc] \bowtie [a]; [ac] \bowtie [b] \\
\text{ab} & = [a] \bowtie [b] \ ; [b] \bowtie [a] \\
\text{bc} & = [b] \bowtie [c] \ ; [c] \bowtie [b] \\
\text{ac} & = [a] \bowtie [c] \ ; [c] \bowtie [a] \\
\text{a} & = A \\
\text{b} & = B \\
\text{c} & = C
\end{array}
$$

Figure 1: The complete MEMO-structure for $\{A - B, A - C, B - C\}$

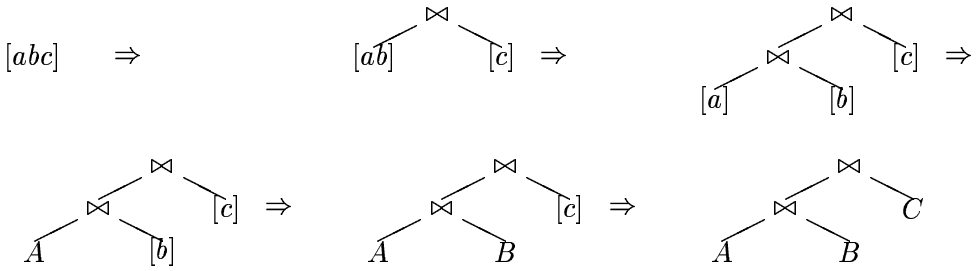

Figure 2: Operator tree extraction from a MEMO-structure.

2.2 Exploration process

A complete MEMO-structure — encoding a complete space — is constructed by recursively exploring the roots of operator trees, starting with an initial join evaluation order. Exploring an operator is done by exhaustively applying all transformation rules to generate all alternatives. This method is similar to the naive algorithm as described in Section 1.

Figure 3 shows the exploration algorithm. The initial MEMO-structure is created by walking down a join tree and creating a class for each join operator. This join tree is selected arbitrarily from the space of valid join trees. To start the exploration we call `EXPLORE-CLASS(`$\mathcal{C}$`)`, with $\mathcal{C}$ being the root class of the initial lookup table.

In general, the application of a transformation rule can generate an operator which is already present in the MEMO-structure. For example, applying the commutativity rule twice reproduces the original operator. So, before inserting a new operator into the MEMO-structure we have to make sure it is not already present. A hash table is used to speed-up the detection of duplicates.

3. Bounds on the size of the MEMO-structure

To determine the size of the MEMO-structure after a search space has been explored we consider a few well known query graph topologies, namely: *string, star* and *completely*

```
EXPLORE-CLASS(C) {
   while not all elements in C have been explored {
      pick an unexplored operator e ∈ C
      EXPLORE-OPERATOR(e);
      mark e explored;
   }
}

EXPLORE-OPERATOR(e) {
   EXPLORE-CLASS(left-child(e));
   EXPLORE-CLASS(right-child(e));
   for each rule R {
      for each partial tree ê such that
            ê is extracted from the MEMO-structure;
            the root of ê is e; and
            ê matches the pattern of R
         x̂ := apply R on ê;
      if x̂ ∉ MEMO-structure
         add x̂ to lookup-table;
         (place the root of x̂ in the same class as e)
   }
}
```

Figure 3: Exploration algorithm

connected queries. For these topologies the size of the MEMO-structure is determined in case *ordered bushy* or *left-linear* join orders are generated.

A string query on n relations consists of 2 *terminal* relations which are adjacent to only one other relation. The other $n-2$ relations are adjacent to 2 relations. In a star query on n relations there is one *central* relation and $n-1$ *terminal* relations. The terminal relations are only adjacent to the central relation and not to any other terminal relation.

3.1 Bushy join evaluation orders

Theorem 1 *The maximum number of join operators needed to encode all alternative evaluation orders for a query of n relations is:* $3^n - 2^{n+1} + n + 1$, *n¿1*

Proof. The upper bound on the size of the MEMO-structure is determined by considering a query topology with the largest number of alternative evaluation orders: A completely connected query on n relations. First, we compute the number of equivalence classes. Since each possible non-empty subset of base relations will occur as intermediate result the number of equivalence classes is: $\sum_{k=1}^{n} \left(\begin{array}{c} n \\ k \end{array} \right) = 2^n - 1$.

An equivalence class for k base relations describes all possible root operators for these k

relations. Every partition of the set of the k relations into left/right non-empty subsets corresponds to an operator in this class, so the number of elements in a class is $2^k - 2$, for $k > 1$. If $k = 1$ the class contains a single operator which fetches a base relation. So if there are n base relations each has a class containing a single fetch operator.
Now the number of operators in the MEMO-structure is the sum of all elements of all classes, which is: $\sum_{k=2}^{n} \begin{pmatrix} n \\ k \end{pmatrix} (2^k - 2) + n = 3^n - 2^{n+1} + n + 1, n > 1$. ■

Theorem 2 *The maximum number of join operators needed to encode all alternative bushy evaluation orders in case of acyclic queries of n relations is:* $(n-2)2^n + n + 2, n > 1$

Proof. In case the query is acyclic every sub query is also acyclic. The number of operators in a class with k, $k > 1$, relations is $2(k-1)$. If $k = 1$ the class contains one operator.
Since the exact topology of the acyclic query graph is unknown and bushy join trees are allowed we assume that all sub graphs exists in the MEMO-structure. This leads to the following summation for the total number of operators: $\sum_{k=2}^{n} \begin{pmatrix} n \\ k \end{pmatrix} 2(k-1) + n$.
Rewriting results in: $(n-2) * 2^n + n + 2, n > 1$ ■

For acyclic queries with a "regular" topology the size of the MEMO-structure can be made more precise. In the following Theorem we give the size of the MEMO-structure for the number of operators needed to encode all bushy trees for string queries.

Theorem 3 *The complete space of bushy join orders for string queries can be encoded in a MEMO-structure using* $(n^3 - n)/3 + n$ *operators.*

Proof. In a string query with n relations there are exactly $n-k+1$ possible substrings of size k. For each of these substrings there is a class that contains all the "root" operators for that class. If left and right input are distinguished there are $2(k-1)$ operators in a class that describes a substring of k operators ($k > 1$). If $k = 1$ the class contains one operator.
So the total number of operators in the MEMO-structure is $2\sum_{k=2}^{n}(n-k+1)(k-1)+n$.
Rewriting results in: $(n^3 - n)/3 + n$ ■

In [OL90] these query graphs and join tree topologies were also considered in the context of the Starburst join enumerator. Our results coincide with their findings, however there is a difference which is caused by the fact that we considered ordered join trees and counted n operators for the base relations. To map their results to ours the formulas have to be multiplied by two and a term n has to added.

3.2 Left-linear join trees

Now we will determine the size of the MEMO-structure if only left-linear join trees are generated. The query graph topologies considered are: *completely connected, string* and *star.*

When counting the number of left-linear join trees we assume that the join operators do not distinguish between the left and right input — i.e. we count unordered linear trees. However, the join operator at the "bottom" of a join tree

does distinguish between it's two inputs [LVZ93]. So, for a completely connected query on the relations a, b and c there are 6 different left-linear join trees, namely: $(c \bowtie b) \bowtie a, (b \bowtie c) \bowtie a, (a \bowtie c) \bowtie b, (c \bowtie a) \bowtie b, (a \bowtie b) \bowtie c, (b \bowtie a) \bowtie c$.

To determine the number of operators needed to represent all valid join trees for acyclic query graphs, we use the following observation. In each join operator of class $\mathcal{C}$ at least one of the inputs is a base relation. Also this base relation has to be a "terminal" in the subgraph associated with class $\mathcal{C}$, otherwise the join operator can not be valid.

Theorem 4 *The number of join operators needed to encode all left-linear join orders for a completely connected join query on n relations is* $n2^{n-1}$.

Proof. A class contains operators such that the right hand operand consists of a single base relation. So the number of operators in a class is k. The n classes referencing 1 relation contain only one operator.

The number of classes with k relations in the fully explored look-up table is $\left(\begin{array}{c} n \\ k \end{array} \right)$.

Summing the operators in each class we get $\sum_{k=2}^{n} \left(\begin{array}{c} n \\ k \end{array} \right) k+n$. Rewriting the summation results in: $n2^{n-1}$. ∎

Theorem 5 *The number of join operators needed to encode all left-linear join trees for a string queries is* n^2.

Proof. In each complete class that references at least two relations there are 2 operators. The two terminal relations appear once as right input of an operator. The classes with one relation contain only 1 operator.
Each sub-string of the complete string query appears as the "other" input, so summing the operators of each class gives the total number of operators. The number of possible sub-strings is $n-k+1$ with k the number of relations in the sub-string and n the number of relations in the complete string. Summing, we get: $\sum_{k=2}^{n} 2(n-k+1)+n = n^2$. ∎

Theorem 6 *The number of join operators needed to encode all left-linear join trees for star queries is* $(n-1) * 2^{n-2} + 2n - 1, n > 2$.

Proof. In a class which references k relations there are $k-1$ possible join operators, $k > 2$. Since, each of the $k-1$ "terminal" relations is once used as the right input for a join operator. The classes which reference two relations contain 2 operators, since the bottom operator distinguishes between the left and right input. The classes that reference one relation contain one operator.

The number of class which reference k relations is $\left(\begin{array}{c} n-1 \\ k-1 \end{array} \right)$. The "other" input is formed by the remaining $k-1$ relations which are selected from the $n-1$ original relations. Summing the number of operator in each class we get: $\sum_{k=3}^{n} \left(\begin{array}{c} n-1 \\ k-1 \end{array} \right) (k-1) + 2 \left(\begin{array}{c} n-1 \\ 1 \end{array} \right) + n$. Rewriting results in $(n-1) * 2^{n-2} + 2n - 1, n > 2$. ∎

In [OL90] also the number of operators needed to encode all left linear trees for string and star queries were derived. These results coincide with our findings except for a minor deviation. This is caused by the fact that we also count the n operators that are required to fetch the base relations and allowing the "bottom" join operator to distinguish between the left and right input.

Rel.	Bushy				Left-linear					
	cc		string		cc		string		star	
	trees	ops.	trees	ops.	trees	ops.	trees	ops.	trees	ops.
2	2	4	2	4	2	4	2	4	2	4
3	12	15	8	11	6	12	4	9	4	9
4	120	54	40	24	24	32	8	16	12	19
5	1680	185	224	45	120	80	16	25	48	41
6	30240	608	1344	76	720	192	32	36	240	91
7	665280	1939	8448	119	5040	448	64	49	1440	272

Figure 4: Number of join trees and operators for various query graphs and join trees.

The table in Figure 4 shows how many operators are needed to encode a specific search space using a MEMO-structure. All the combinations of query graphs and join trees discussed in this section are described. For computing the size of the search space — number of join trees — the formulas of Figure 5 were used. Note that for star query graphs there are no "real" bushy join trees. In the tables "cc" is short for completely connected.

	Bushy	Left-linear
cc	$\frac{(2n-2)!}{(n-1)!}$	$n!$
string	$\frac{(2n-2)!}{n!(n-1)!}2^{n-1}$	2^{n-1}
star	-	$2(n-1)!$

Figure 5: Formula's for the size of search spaces.

4. Duplicates

Before quantifying the effect of duplicate generation, we walk through the construction of a MEMO-structure for a completely connected query, on relations a, b and c. Even in this small example the number of duplicates is relatively large.

Example 1 For the completely connected query on the relations "a, b, c," Figure 6 shows the MEMO-structures before and after exploring operator $[ab] \bowtie [c]$. In the "before" MEMO-structure all children, "ab" and "c", have been fully explored. The transformation rules **RS-B1** (see Section 2) generate the following new operators, when applied to operator $[ab] \bowtie [c]$.

Commutativity: $([ab] \bowtie [c])$ creates $([c] \bowtie [ab])$ which is added to the class "abc".

Before	After
abc=$[ab] \bowtie [c]$	abc=$[ab] \bowtie [c]; [c] \bowtie [ab]; [a] \bowtie [bc]; [b] \bowtie [ac]; [bc] \bowtie [a]; [ac] \bowtie [b]$.
ab =$[a] \bowtie [b]; [b] \bowtie [a]$	ab =$[a] \bowtie [b]; [b] \bowtie [a]$
	bc =$[b] \bowtie [c]; [c] \bowtie [b]$
	ac =$[a] \bowtie [c]; [c] \bowtie [a]$

Figure 6: MEMO-structure before and after exploration.

Associativity: $([ab] \bowtie [c])$ does not match, the left child is a class and should be a tree. This is resolved by extracting a partial trees for the left class "ab."

$[a] \bowtie [b]$**:** First tree $(([a] \bowtie [b]) \bowtie [c])$ is extracted. Now the rule matches and is applied. The new tree $([a] \bowtie ([b] \bowtie [c]))$ is generated, and added to the MEMO-structure in class "abc". The subexpression $([b] \bowtie [c])$ starts a new class "bc" since it didn't appear in the earlier MEMO-structure.

$[b] \bowtie [a]$**:** Second tree $(([b] \bowtie [a]) \bowtie [c])$ is extracted. It matches the rule, so it is applied. The new tree $([b] \bowtie ([a] \bowtie [c]))$ is generated and added to the MEMO-structure. The subexpression $[a] \bowtie [c]$ starts a new class "ac".

The exploration process is continued by applying transformation rules to the newly created operators. Now, duplicates are generated. Before the new operators ($[c] \bowtie [ab]$, $[a] \bowtie [bc]$, and $[b] \bowtie [ac]$) of the root class "abc" can be explored, all their children ("a","b","c" "ab","bc" and "ac") must be fully explored. This results in two new operators, $[c] \bowtie [b]$ and$[c] \bowtie [a]$, which are added to the appropriate classes.

Commutativity on the new operators produces $[ab] \bowtie [c]$, $[bc] \bowtie [a]$ and $[ac] \bowtie [b]$, out of which $[ab] \bowtie [c]$ already exists. The new operators are added to the MEMO-structure and explored. Both associativity and commutativity can be applied to the operators $[bc] \bowtie [a]$ and $[ac] \bowtie [b]$, which results in 6 operators. All these operators were already stored in the MEMO-structure. So, during the exploration of class "abc", 5 new operators and 7 duplicates were generated. In Figure 7 the generation graph for class "abc" is shown. The bold operators are duplicates.

■

As illustrated by the previous example, the straight forward application of transformation rules results in the generation of operators which are already in the MEMO-structure — duplicates. The generation of duplicates affects the efficiency of the join enumeration process considerably. For each operator generated the MEMO-structure has to be searched to determine if it already exists. The search and the time needed to generate duplicates are part of the generation cost.

In the introduction, where a simple graph model was assumed, we derived a factor of $b - 1$ of duplicate elements over new elements. The naive model used there, however, did not take into account the MEMO-structure used by the Volcano-type optimizers. The next Theorems deal with the details of the new structure.

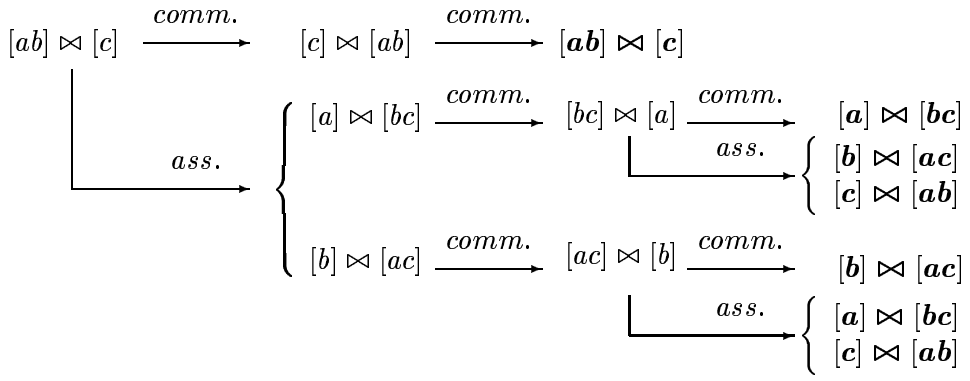

Figure 7: Operator generation graph.

4.1 Bushy join trees

The following theorem shows the number of duplicates generated when exploring the search space of bushy join trees, for completely connected query graphs. It assumes a minimal set of unidirectional join associativity and commutativity rules, See 2. If associativity is enabled in both direction, as is commonly suggested, we simply end up generating more duplicates.

Lemma 1 *The number of duplicates generated by* **RS-B1** *during the exploration of a class that combines k relations, on a completely connected graph, is:* $3^k - 3 * 2^k + 4$.

Proof. In a class that combines k relations, take an operator $[L] \bowtie [R]$, with l the number of relations in $[L]$ and $k-l$ the number of relations in $[R]$ $(2 \leq k \leq n)$. Applying commutativity and associativity on this operator we generate $(2^l - 2) + 1$ alternatives. So the total number of operators generated in the class is $\sum_{l=1}^{k-1} \left(\begin{array}{c} k \\ l \end{array} \right) (2^l - 1)$. Rewriting, the summation becomes $3^k - 2^{k+1} + 1$. But the number of unique operators in such class is $2^k - 2$ and of these elements the initial operator is given instead of being generated. Therefore the number of duplicates generated in the class is: $3^k - 2^{k+1} + 1 - (2^k - 2 - 1) = 3^k - 3 * 2^k + 4$. ∎

Theorem 7 *The number of duplicates generated by* **RS-B1** *during the construction of a MEMO-structure encoding all bushy join trees for a query with n relations, on a completely connected graph, is:* $4^n - 3^{n+1} + 2^{n+2} - n - 2$.

Proof. The MEMO-structure consists of $\left(\begin{array}{c} n \\ k \end{array} \right)$ classes that combine k relations, $2 \leq k \leq n$. Using Lemma 1 the total number of duplicates generated is $\sum_{k=2}^{n} \left(\begin{array}{c} n \\ k \end{array} \right) (3^k - 3 * 2^k + 4)$. Rewriting results in: $4^n - 3^{n+1} + 2^{n+2} - n - 2$. ∎

4.2 Linear join trees

The following lemma and theorem show how many duplicate join operators are generated when generating the MEMO-structure for all left linear join trees.

Lemma 2 *The number of duplicates generated by* **RS-L1** *during the exploration of a class that combines k relations, on a completely connected graph, is: $k^2 - 2k + 1$.*

Proof. In a class that combines k relations, take an operator $[L] \bowtie [R]$, with l the number of relations in $[L]$ and r the number of relations in $[R]$, $r + l = k$ and $2 \leq k \leq n$. Since we are considering left linear join trees for each operator $l = k - 1$ and $r = 1$.
For a class with more than two relations, $k > 2$, only the Swap rule applies to the join operators. For each join operator this rule generates $k - 1$ alternative operators. Since a class contains k unique operators, $k(k-1)$ operators are generated in a class. Of the k unique operators one is already given, the initial one, so the number of duplicates generated is: $k(k-1) - (k-1) = k^2 - 2k + 1$.
For the class with exactly two relations, $k = 2$, only the Bottom Commutativity rule applies. This rule behaves the same as the Swap rule so the formula $k(k-1) - (k-1) = k^2 - 2k + 1$ also holds for $k = 2$. ∎

Theorem 8 *The number of duplicates generated by* **RS-L1** *during the construction of a MEMO-structure encoding the left linear join trees for a query with n relations, on a completely connected graph, is: $(n^2 - 3n + 4)2^{n-2} - 1$.*

Proof. The MEMO-structure consists of $\binom{n}{k}$ classes that combine k relations, $2 \leq\leq n$. Using the previous lemma, the total number of duplicates generated during the construction of the MEMO-structure is: $\sum_{k=2}^{n} \binom{n}{k} (k^2 - 2k + 1)$. Rewriting results in: $(n^2 - 3n + 4)2^{n-2} - 1$. ∎

	Bushy join trees		Left-linear join trees	
Relations	Operators	Duplicates	Operators	Duplicates
2	4	1	4	1
3	15	10	12	7
4	54	71	32	31
5	185	416	80	111
6	608	2157	192	351
7	1939	10326	448	1023

Figure 8: Number of duplicates generated during the exploration of a MEMO-structure.

Figure 8 shows concrete numbers for the size of the MEMO-structure and the duplicates generated (both bushy and linear trees), as a function of the number of relations joined, for fully connected graphs. The second column gives the number of operators needed to

encode all bushy trees using the MEMO-structure. The number of duplicates generated during the exploration process is given in column 3. For linear join trees the size of the MEMO-structure and the number of duplicates generated is given in column 4 and 5.

Combining the results form Theorem 1 and 7 shows that for bushy join trees the ratio of duplicates over new elements is $O(2^{n \log(4/3)})$. For left linear join trees the ratio of duplicates over new elements is $O(n)$, Theorem 4 and 8.

5. Duplicate-free join order generation

To avoid the generation of duplicates, information about the behaviour of transformation rules is used. For example, if an element was generated by applying the commutativity rule, there is no point in applying that rule again, because it will result in the original element.

To determine by which rule a join operator has been generated a "derivation history" is recorded for each element. This is done by keeping track of rules that are still worth applying. For example, the application of the commutativity rule will switch the commutativity rule off in the rule set of the resulting element.

Keeping track of the derivation history requires only a few bits per operator. The applicability of a rule can be encoded using a single bit. So each operator needs as many bits as there are transformation rules. Using compression techniques this could even be reduced further.

In the next sections, we present sets of duplicate-free transformation rules, together with an application schema. The join trees generated are either *left-linear* or *bushy*. The query graph topologies we consider are *acyclic* and *completely connected*. For each combination of join tree and query graph there is a different set of transformation rules. The next table shows in which section which combination is described.

Query graph:	Completely connected	Acyclic
Bushy join tree	5.1	5.2
Linear join tree	5.3	5.4

5.1 Bushy trees for completely connected queries

Generating all valid join trees for completely connected queries is similar to generating all join trees for arbitrary query graphs (cyclic or acyclic) which contain both Cartesian products and valid join operators. Since each query graph can be transformed into a completely connected query graph by adding the "missing" predicates. These added predicates are set to *true*, forcing the join operators to compute a Cartesian product. The following rule set generates all bushy join trees for completely connected queries without duplicates.

Rule set **RS-B**$_{cc}$

R_1 : **Commutativity** $x \bowtie_0 y \rightarrow y \bowtie_1 x$
Add $\bowtie_1$ to the class of $\bowtie_0$.
Disable all rules R_1, R_2, R_3, R_4 for application on $\bowtie_1$.

R_2 : **Right associativity** $(x \bowtie_0 y) \bowtie_1 z \rightarrow x \bowtie_2 (y \bowtie_3 z)$
Add $\bowtie_2$ to the class of $\bowtie_1$.
Disable rules R_2, R_3, R_4 for application on $\bowtie_2$.
Start new class with $\bowtie_3$ (new operator) with all rules enabled.

R_3 : **Left associativity** $x \bowtie_0 (y \bowtie_1 z) \rightarrow (x \bowtie_2 y) \bowtie_3 z$
Add $\bowtie_3$ to the class of $\bowtie_0$.
Disable rules $R_2, R_3 R_4$ for application on $\bowtie_3$.
Start new class with $\bowtie_2$ (new operator) with all rules enabled.

R_4 : **Exchange** $(w \bowtie_0 x) \bowtie_1 (y \bowtie_2 z) \rightarrow (w \bowtie_3 y) \bowtie_4 (x \bowtie_5 z)$
Add $\bowtie_4$ to the class of $\bowtie_1$.
Disable all rules R_1, R_2, R_3, R_4 for application on $\bowtie_4$.
Start new classes with $\bowtie_3$ and $\bowtie_5$ (new operators) with all rules enabled.

Each transformation rule generates a non-overlapping partition of the space of join operators, and each rule generates each operator within such a partition exactly once. Furthermore the complete space of join operators is the conjunction of all partitions, see Figure 9 .

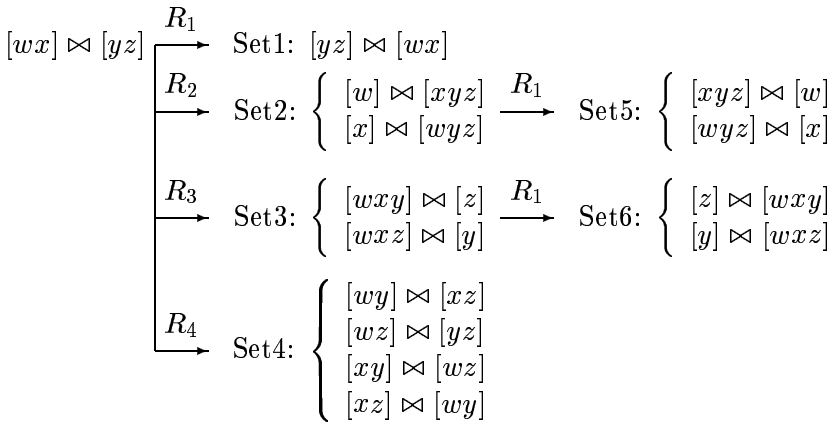

Figure 9: Partitions generated by the transformation rules

For an initial join operator, $\mathcal{L} \bowtie \mathcal{R}$, with $\mathcal{L} = [wx]$ and $\mathcal{R} = [yz]$ rule R_2 generates the join operators that combine each *alternative* of $\mathcal{L} : \{w, x\}$ with $\mathcal{R}$ so the elements $[w] \bowtie [xyz]$ and $[x] \bowtie [wyz]$ are generated. Rule R_3 and R_4 work in a similar fashion. R_1 generates the mirror images for the original operator and all operators generated by rule R_2 and R_3 (since $\mathcal{L}$ and $\mathcal{R}$ contain all alternatives including the mirror images, rule R_4 automatically generates the mirror images).

Theorem 9 *If the transformation rules of rule set* **RS-B**$_{cc}$ *are applied to* $\mathcal{L} \bowtie \mathcal{R}$ *then the newly generated elements will be duplicate free if the child classes* $\mathcal{L}$ *and* $\mathcal{R}$ *are duplicate free.*

Proof. Two operators can not be identical if they are both generated by the same rule (elements of the same set). Namely, rule R_1 is used to generate mirror images of operators, since the left and right operand will never be identical a duplicate can not

be generated. Rule R_2 combines the unique operators of the left child with the right operand of the initial operator resulting in only unique operators. The same holds for rule R_3 and R_4.
Also, no two derivation paths can result in the same operator (elements of different sets). Suppose the application of rule R_2 (Set 2) generated the same element as rule $R_3; R_1$ (Set 6), then $[w] \bowtie [xyz]$ or $[x] \bowtie [wyz]$ has to be equal to $[z] \bowtie [wxy]$ or $[y] \bowtie [wxz]$. This can not be true since w, x, y, z are disjunct non-empty sets of relations, so $[w] \neq [z] \neq [x] \neq [y]$ and $[xyz] \neq [wxy] \neq [wyz] \neq [wxz]$. A similar argument can be given for any other combination of rules. ■

Theorem 10 *For completely connected queries the rules of rule set* **RS-B**$_{cc}$ *generate all valid bushy join orders.*

Proof. In a fully explored class that references n relations the number of join operators is $B_{cc}(n) = 2^n - 2$ (See proof of theorem 1). From the initial operator of a class, say $\mathcal{L} \bowtie \mathcal{R}$ the transformation rules generate the following elements. Rule R_2 combines each *element* of class "$\mathcal{L}$" with $\mathcal{R}$ resulting in $B_{cc}(|\mathcal{L}|)$ new operators, with $|\mathcal{L}|$ denoting the number of relations in class "$\mathcal{L}$". Similarly rule R_3 generates $B_{cc}(|\mathcal{R}|)$ new elements. Rule R_4 combines each *element* of class "$\mathcal{L}$" with each *element* of class "$\mathcal{R}$" which results in $B_{cc}(|\mathcal{L}|)B_{cc}(|\mathcal{R}|)$ new elements. Finally rule R_1 generates the mirror images for the initial operator and the operators generated by rule R_2 and R_3. Adding all the newly created operators and the initial operator we get: $2 + 2B_{cc}(|\mathcal{L}|) + 2B_{cc}(|\mathcal{R}|) + B_{cc}(|\mathcal{L}|)B_{cc}(|\mathcal{R}|)$. Rewriting shows that $2+2B_{cc}(|\mathcal{L}|)+2B_{cc}(|\mathcal{R}|)+B_{cc}(|\mathcal{L}|)B_{cc}(|\mathcal{R}|) = B_{cc}(|\mathcal{L}|+|\mathcal{R}|)$. From theorem 9 we know that no duplicates are generated, so all valid bushy join trees are generated. ■

Example 2 For a completely connected query on five relations $\{a, b, c, d, e\}$, Figure 10 shows a partial MEMO-structure in which the children of operator $[ab] \bowtie [cde]$ have been fully explored. Applying the transformation rules of rule set **RS-B**$_{cc}$ results in the generation of the following elements.

Rule R$_2$: Each split of the left subtree $[ab]$ is combined with the right subtree $[cde]$ to obtain $[a] \bowtie [bcde], [b] \bowtie [acde]$.

Rule R$_3$: Each split of the right subtree $[cde]$ is combined with the left subtree $[ab]$ to obtain $[abc] \bowtie [de], [abde] \bowtie [c], [abd] \bowtie [ce], [abce] \bowtie [d], [abe] \bowtie [cd], [abcd] \bowtie [e]$.

Rule R$_4$: The splittings of the left subtree are combined with the splittings of the right subtree resulting in the elements $[ac] \bowtie [bde], [ade] \bowtie [bc], [ad] \bowtie [bce], [ace] \bowtie [bd], [ae] \bowtie [bcd], [acd] \bowtie [be]$ and the mirror images $[bde] \bowtie [ac], [bc] \bowtie [ade], [bce] \bowtie [ad], [bd] \bowtie [ace], [bcd] \bowtie [ae], [be] \bowtie [acd]$.

Rule R$_1$: The mirror images of the initial element and the elements generated by the rules R_2 and R_3.

During the exploration process 20 new classes were generated and, in turn, fully explored. The complete explored class "*abcde*" contains 30 ($= B_{cc}(5)$) elements. ■

$$
\begin{array}{ll}
\text{abcde} & = [ab] \bowtie [cde] \\
\text{cde} & = [c] \bowtie [de]; [de] \bowtie [c]\ ; [d] \bowtie [ce]; \\
 & \quad [ce] \bowtie [d]; [e] \bowtie [cd]; [cd] \bowtie [e] \\
\text{ab} & = [a] \bowtie [b]; [b] \bowtie [a] \\
\text{cd} & = [c] \bowtie [d]; [d] \bowtie [c] \\
\text{ce} & = [c] \bowtie [e]; [e] \bowtie [c] \\
\text{de} & = [d] \bowtie [e]; [e] \bowtie [d]
\end{array}
$$

Figure 10: Partial MEMO-structure with bushy trees for a completely connected query on five relations.

5.2 Bushy trees for Acyclic queries

To generate all alternative bushy join operators, without duplicates, for acyclic query graphs the following transformation rules and application schema is used:

Rule set **RS-B**$_{ac}$

R_1 : **Commutativity** $x \bowtie_0 y \rightarrow y \bowtie_1 x$
Disable all rules R_1, R_2, R_3 for application on the new operator $\bowtie_1$.

R_2 : **Right associativity** $(x \bowtie_0 y) \bowtie_1 z \rightarrow x \bowtie_2 (y \bowtie_3 z)$
Disable rules R_2, R_3 for application on the new operator $\bowtie_2$.
Start new class with new operator $\bowtie_3$, with all rules enabled.

R_3 : **Left associativity** $x \bowtie_0 (y \bowtie_1 z) \rightarrow (x \bowtie_2 y) \bowtie_3 z$
Disable rules R_2, R_3 for application on the new operator $\bowtie_3$.
Start new class with new operator $\bowtie_2$ with all rules enabled.

For example, consider a query with predicates between relations $(w, x), (x, y), (y, z)$. Using the initial element $[wx] \bowtie [yz]$ of a class and the fully explored classes of "wx" and "yz" the three transformation rules generate the following five sets of elements (See Figure 11). Sets 1,2 and 3 are generated using the initial element. Sets 4 and 5 are generated using the elements of set 2 and 3. Join $[wx] \bowtie [yz]$ must be a valid join tree (i.e. no Cartesian products) of an acyclic query graph.

Set 2 is generated by the right associativity rule and contains only one valid result. Since the graph is connected and acyclic, there must be a predicate between yz and either w or x, but not both; say it is between yz and x. A sub-query combining tables wyz would have to use a Cartesian product, which is invalid. Therefore, R_2 generates only one valid alternative. The same argument applies on the left associativity rule used for Set 3.

Theorem 11 *No duplicates are generated when rule set* **RS-B**$_{ac}$ *is applied.*

Proof.
The proof is analogous to the proof of Theorem 9

■

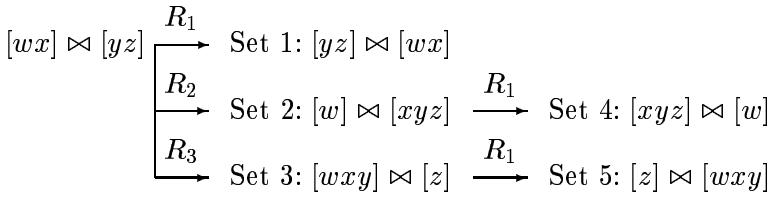

Figure 11: Partitions generated by the transformation rules

Theorem 12 *For acyclic query graphs rule set* **RS-B**$_{ac}$ *generates all valid bushy join operators.*

Proof. Since the query graph is acyclic, each join operator that can serve as root for a valid join tree corresponds one-to-one to an edge of the query graph. So for a query graph with n relations the number of join operators in the root class is $B_{ac}(n) = 2(n-1)$, when the mirror images are included. Note that each explored class describes all valid roots of the corresponding acyclic sub-graph.
Using the initial element of a class, say $\mathcal{L} \bowtie \mathcal{R}$, the transformation rules generate the following new elements. Rule R_2 combines each *element* of class $\mathcal{L}$ with $\mathcal{R}$, of these new combinations only half is valid since a class contains mirror images and only one can lead to a new valid join operator. This means that rule R_2 generates $B_{ac}(|\mathcal{L}|)/2$ new operators, with $|\mathcal{L}|$ denoting the number of operators of class $\mathcal{L}$. Similarly rule R_3 generates $B_{ac}(|\mathcal{R}|)/2$ new operators. Finally rule R_1 generates for each new operator and the initial operator a mirror image which results in $2(1 + B_{ac}(|\mathcal{L}|)/2 + B_{ac}(|\mathcal{R}|)/2) = 2 + B_{ac}(|\mathcal{L}|) + B_{ac}(|\mathcal{R}|)$ elements for the fully explored class.
Now, $2 + B_{ac}(|\mathcal{L}|) + B_{ac}(|\mathcal{R}|) = B_{ac}(|\mathcal{L}| + |\mathcal{R}|)$, which is the number of join operators for the fully explored class with $|\mathcal{L}| + |\mathcal{R}|$ relations. Since, by Theorem 11, no duplicate operators were produced, all valid join orders have been generated. ■

Example 3 Given a query $G = \{\{a, b, c, d, e\}, \{a - b, b - c, c - d, c - e\}\}$ and a look-up table as shown in Figure 12, where the class "abcde" is about to be explored. Of the initial operator of the root class "abcde" the child classes have been explored exhaustively.

Applying the transformation rules of rule set **RS-B**$_{ac}$ to $[ab] \bowtie [cde]$ results in generating the following elements.

Rule R$_2$: $[a] \bowtie [bcde]$.
The element $[b] \bowtie [acde]$ is considered by the associativity rule, but rejected, because there is no valid join tree for "acde" (we would be force to introduce a Cartesian product because there is no predicate between a and any of c, d, e).

Rule R$_3$: $[abce] \bowtie [d]$, $[abcd] \bowtie [e]$.
The elements $[abd] \bowtie [ce]$ and $[abe] \bowtie [cd]$ are considered but rejected because there are no valid join trees for "abd" and "abe".

Rule R$_1$: $[cde] \bowtie [ab]$, $[bcde] \bowtie [a]$, $[d] \bowtie [abce]$, $[e] \bowtie [abcd]$

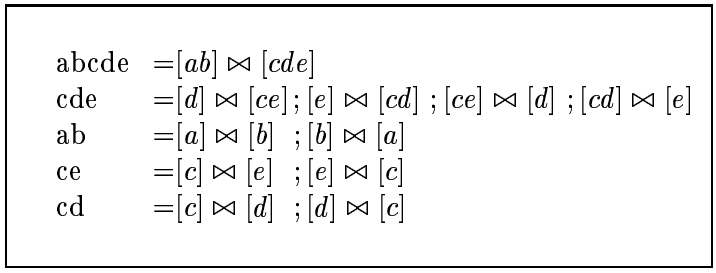

Figure 12: Partial MEMO-structure with bushy trees for an acyclic query.

The fully explored class "abcde" contains all 8 ($= B_{ac}(5)$) elements. During the exploration process the new classes $[bcde], [abce]$ and $[abcd]$ are created and, in turn, fully explored. ■

5.3 Linear trees for completely connected queries

Some systems limit the join evaluation orders to linear trees. This section and the following section describe transformation rules which generate linear join evaluation orders for acyclic query graphs and completely connected query graphs. To generate all the linear join trees for completely connected queries the following rule set is used.

Rule set **RS-L**$_{cc}$

R_1 : **Swap** $(x \bowtie_0 y) \bowtie_1 z \rightarrow (x \bowtie_2 z) \bowtie_3 y$.
Disable rule R_1 for application on operator $\bowtie_3$.
x, y and z are classes which reference one or more relations.

R_2 : **Bottom Commutativity** $(Table_1 \bowtie_0 Table_2) \rightarrow (Table_2 \bowtie_1 Table_1)$.
Disable rule R_2 for application on operator $\bowtie_1$.
$Table_1$ and $Table_2$ are base relation.

All valid join operators in the MEMO-structure have a single base relation as the right operand. For classes with more than two relations the swap rule generates all valid join operators if the class of the left operand is fully explored. For a class which references two relations the bottom commutativity rule generates a mirror image. This is needed to ensure that in larger classes all base relations appear once as the right input of a join operator.

Alternatively the bottom commutativity rule could be omitted and and exception for classes with two relations could be added to the swap rule. This reduces the size of the look-up table but makes it harder to describe how all valid join operators are generated.

Theorem 13 *The transformation rules of rule set RS-L$_{cc}$ do not generate duplicates if applied to an initial operator $\mathcal{L} \bowtie \mathcal{R}$, where $\mathcal{R}$ consists of a single base relation and the class for $\mathcal{L}$ does not contain duplicates.*

Proof. The application schema defines two distinct derivation paths. Either $\mathcal{L}$ and $\mathcal{R}$ are both base relations and only rule R_2 applies, or $\mathcal{L}$ references more than one relation and $\mathcal{R}$ is a base relation so only rule R_1 applies. For each case a proof similar to the proof for Theorem 9 can be given. ■

$$[wx] \bowtie [y] \xrightarrow{R_1} \text{Set 2: } \{[wy] \bowtie [x], [xy] \bowtie [w]\}$$

$$[Table_1] \bowtie [Table_2] \xrightarrow{R_2} \text{Set 1: } [Table_2] \bowtie [Table_1]$$

Figure 13: Partitions generated by the transformation rules of rule set RS-L_{cc}.

Theorem 14 *For completely connected queries the transformation rules of rule set RS-L_{cc} generate all valid linear join operators.*

Proof. For a class which references n relations, $n > 2$, there are $L_{cc}(n) = n$ join operators. Each of the n relations appears once as the right operand of a join operator. For the initial operator $\mathcal{L} \bowtie \mathcal{R}$ class "$\mathcal{R}$" is a base relation, $|\mathcal{R}| = 1$. Then rule R_1 generates $L_{cc}(|\mathcal{L}|)$ new operators. So the total number of operators in the class is $1 + L_{cc}(|\mathcal{L}|)$. Rewriting results in $L_{cc}(|\mathcal{L}| + |\mathcal{R}|)$. Since no duplicates are generated all join operators must have been generated. ■

Example 4 For a completely connected query on the five relations a, b, c, d, e, Figure 14 shows a partial MEMO-structure in which the children of the initial operator $[a] \bowtie [bcde]$ have been fully explored. Applying the transformation rules of rule set RS-$L_{cc}$to the initial element $[abcd] \bowtie [e]$ results in the generation of the following elements:

Rule R_1: $[bcde] \bowtie [a]$, $[acde] \bowtie [b]$, $[abde] \bowtie [c]$ and $[abce] \bowtie [d]$.

Rule R_2: This rule is only active in classes that references two base relations. For instance when class "ae" is generated rule R_2 is applicable and will generate a mirror image.

During the exploration process four new classes are created and, in turn, fully explored. After exploration the total number of operators in the root class is 5 ($= C_{cc}(5)$).

■

5.4 Linear trees for Acyclic queries

The rule set used to generate all linear join trees for acyclic queries is similar to rule set **RS-L_{cc}**. The only difference is that the operators generated by the swap rule are not necessarily valid so a test is added to the transformation rule.

Rule set **RS-L_{ac}**

R_1 : **Swap** $(x \bowtie_0 y) \bowtie_1 z \rightarrow (x \bowtie_2 z) \bowtie_3 y$.
Disable rule R_1 for application on operator $\bowtie_3$.
x, y and z are classes which reference one or more relations.
Add $\bowtie_3$ only to the class of $\bowtie_1$ if it is a valid operator.

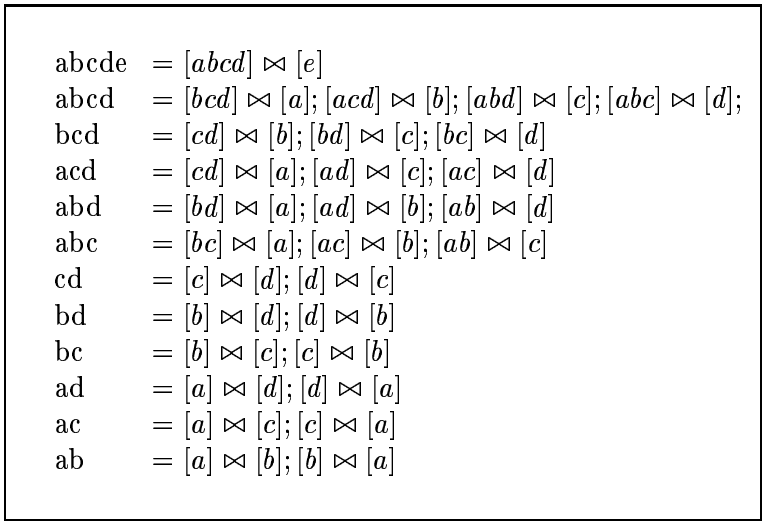

Figure 14: Partial MEMO-structure containing linear trees for a completely connected query on five relations.

R_2 : **Bottom Commutativity** $(Table_1 \bowtie_0 Table_2) \rightarrow (Table_2 \bowtie_1 Table_1)$.
Disable rule R_2 for application on operator $\bowtie_1$.
$Table_1$ and $Table_2$ are base relation.

Theorem 15 *The transformation rules of rule set RS-L_{ac} do not generate duplicates if applied on an initial operator $\mathcal{L} \bowtie \mathcal{R}$, where $\mathcal{R}$ consists of a single base relation and the class for $\mathcal{L}$ does not contain duplicates.*

Proof. The application schema defines two distinct derivation paths. Either $\mathcal{L}$ and $\mathcal{R}$ are both base relations and only rule R_2 applies, or $\mathcal{L}$ references more than one relation and $\mathcal{R}$ is a base relation so only rule R_1 applies. For each case a proof similar to the proof for Theorem 9 can be given. ■

Theorem 16 *The transformation rules of rule set RS-L_{ac} generate all valid linear join trees if applied on an initial operator $\mathcal{L} \bowtie \mathcal{R}$, where $\mathcal{R}$ references a single relation.*

Proof. All valid root operators of a class have a single base relation as right input. This base relation is a "terminal" in the associated query graph, otherwise the operator is not be valid.
Assume $\mathcal{R}$ is a single relation and class "$\mathcal{L}$" has been explored completely. Now rule R_1 will combine $\mathcal{R}$ with each operator of class "$\mathcal{L}$" resulting a number of new join operators. Only the operators that are valid will be added to class that is being explored. ■

Example 5 Given a query $G = \{\{a, b, c, d, e\}, \{a - b, b - c, c - d, c - e\}\}$ and the MEMO-structure as shown in Figure 15, where the class "abcde" is about to be explored. Of the initial operator in class "abcde" the child classes have been explored exhaustively.

Applying the transformation rules of rule set RS-L_{ac} results in generating the following elements.

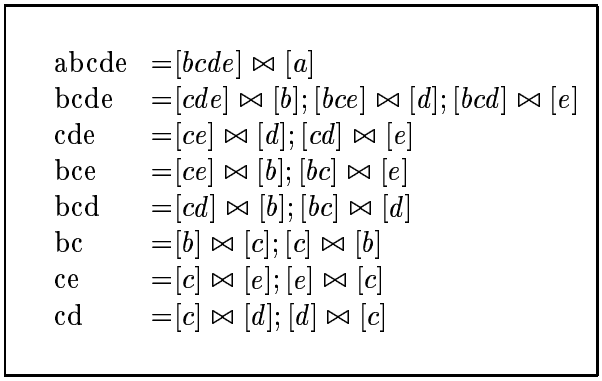

Figure 15: Partial MEMO-structure with linear trees for acyclic queries

Rule $\mathbf{R_1}$: $[acde] \bowtie [b], [abce] \bowtie [d], [abcd] \bowtie [e]$.
The element $[acde] \bowtie [b]$ is discarded because there is no predicate that connects relation a to either c, d or e.

Rule $\mathbf{R_2}$: This rule can not be applied for class "abcde" since is is only active for classes with exactly two relations.

The fully explored class "abcde" contains 3 elements. During the exploration process the new classes $[abce]$ and $[abcd]$ are created and, in turn, fully explored.

■

6. Summary

In this paper we analyse the problem of generation of duplicate plans, for transformation-based optimizers that explore a space exhaustively. Despite the exponential size of the space, exhaustive search is often used in practice. We are aware of at least two commercial DBMSs under development that are using a Volcano-type optimizer based on exhaustive search one at Tandem in their *NonStop SQL* product and one at Microsoft in their *SQL Server* product [Gra95].

We have shown that duplicates are a serious problem in transformation based optimizers. Even for small queries the number of duplicates exceeds the number of new elements, and it increases dramatically with the size of the query. In particular, for the Volcano-type optimizers the ratio of duplicates over new elements is $O(2^{n \log(4/3)})$. The detailed complexity analysis developed here is the first that we are aware of, for this type of optimizers.

Our approach to an efficient generation algorithm is to keep track of the transformation rules that can still be applied without generating duplicates at any point in the search space. The overhead of the method consists of a few bits per operator. It is described in detail, for several classes of query topologies and both bushy and linear join trees.

The conditioned application of rules can be incorporated easily in the existing framework of modern query optimizers, and preliminary tests corroborate that considerable performance improvements result from the large reduction of generated elements. The performance improvement gained by avoiding duplicate generation is significant in practice, and it should be used whenever possible.

For arbitrary sets of transformation rules it might be hard to transform them into an efficient set. Determining which set of arbitrary transformation rules can be converted into a duplicate-free set, and the interaction with other rules is our current focus of research.

REFERENCES

[BMG93] J.A. Blakeley, W. J. McKenna, and G. Graefe. Experiences bulding the open oodb query optimizer. *Proceedings of the ACM SIGMOD Conf on Management of Data, Washington DC*, 1993.

[GCD+94] G. Graefe, R. L. Cole, D . L. Davison, W. J. McKenna, and R. H. Wolniewicz. *Query Processing for Advanced Database Systems.* Morgan Kaufmann, 1994.

[GLPK94] C. A. Galindo-Legaria, A. Pellenkoft, and M. L. Kersten. Fast, randomized join-order selection —Why use transformations? In *Proceedings of the Twentieth International Conference on Very Large Databases, Santiago*, 1994. Also CWI Technical Report CS-R9416.

[GLPK95] C. A. Galindo-Legaria, A. Pellenkoft, and M. L. Kersten. Uniformly-distributed random generation of join orders. In *Proceedings of the International Conference on Database Theory, Prague*, pages 280–293, 1995. Also CWI Technical Report CS-R9431.

[GM93] G. Graefe and W. J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. *Procedings of the 9th International Conference on Data Engineering, Vienna, Austria*, pages 209–218, 1993.

[Gra95] G. Graefe. The cascades framework for query optimization. *IEEE Data Engineering Bulletin*, 18(3):19 – 29, September 1995.

[IK90] Y. E. Ioannidis and Y. C. Kang. Randomized algorithms for optimizing large join queries. *Proc. of the ACM-SIGMOD Conference on Management of Data*, pages 312–321, 1990.

[IK91] Y. E. Ioannidis and Y. C. Kang. Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization. *Proc. of the ACM-SIGMOD Conference on Management of Data*, pages 168–177, 1991.

[IW87] Y. E. Ioannidis and E. Wong. Query optimization by simulated annealing. *Proc. of the ACM-SIGMOD Conference on Management of Data*, pages 9–22, 1987.

[Kan91] Y. C. Kang. *Randomized Algorithms for Query Optimization.* PhD thesis, University of Wisconsin-Madison, 1991. Technical report #1053.

[LVZ93] R. S. G. Lanzelotte, P. Valduriez, and M. Zaït. On the effectiveness of optimization search strategies for parallel execution spaces. *Proc. of the 19th VLDB Conference, Dublin, Ireland*, pages 493–504, 1993.

[OL90] K. Ono and G. M. Lohman. Measuring the complexity of join enumeration in query optimization. *Proc. of the 16th VLDB Conference, Brisbane, Australia*, pages 314–325, 1990.

[SG88] A. N. Swami and A. Gupta. Optimization of large join queries. *Proc. of the ACM-SIGMOD Conference on Management of Data*, pages 8–17, 1988.