The Complexity of Transformation-Based Join Enumeration

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

Query optimizers that explore a search space exhaustively using transformation rules usually apply all possible rules on each alternative, and stop when no new information is produced. A memoizing structure was proposed in [McK93] to improve the re-use of common subexpression, thus improving the efficiency of the search considerably. However, a question that remained open is, what is the complexity of the transformation-based enumeration process? In particular, with n the number of relations, does it achieve the $O(3^n)$ lower bound established by [OL90]?

In this paper we examine the problem of duplicates, in transformation-based enumeration. In general, different sequences of transformation rules may end up deriving the same element, and the optimizer must detect and discard these duplicate elements generated by multiple paths. We show that the usual commutativity/associativity rules for joins generate $O(4^n)$ duplicate operators. We then propose a scheme —within the generic transformation-based framework— to avoid the generation of duplicates, which does achieve the $O(3^n)$ lower bound on join enumeration. Our experiments show an improvement of up to a factor of 5 in the optimization of a query with 8 tables, when duplicates are avoided rather than detected.

Proceedings of the 23rd VLDB Conference
Athens, Greece, 1997

1 Introduction

Ono and Lohman [OL90] gave a lower bound of $O(3^n)$, with n the number of relations, on the complexity of join enumeration, by counting how many join operators have to be considered in the bottom-up, dynamic programming enumeration algorithm of System R and Starburst.[1] This algorithm is efficient for join enumeration, and the code has been extensively tested and tuned over the years. But there is an advantage to reordering other operators, and choosing among new alternatives based on cost estimation. Although the algorithm has been extended to deal with other operators (e.g. aggregates [CS96], outerjoins [GLR96], expensive functions [HN96]), there is no precise characterization of the properties of operators that can be handled by the enumeration algorithm. There is no general technique for these extensions, and no guarantee that the next extension will be possible. For example, to deal with subqueries, Seshadri et al suggest a clever iterative process in which an extended bottom-up enumeration module is called

[1]A careful analysis by Vance [VM96, Van96a] shows that the bottom-up enumeration algorithm doesn't achieve this lower bound in all cases, having a complexity of $O(4^n)$.

multiple times [SHP+96] —but we are forced beyond the bottom-up enumeration framework.

Proponents of "rule-based, extensible optimizers" would say that the problem is a "lack of extensibility," and they would point to a number of research prototypes that have been developed over the past 10 years. But the common view is that the tradeoff of extensible optimizers is efficiency.

Is transformation-based enumeration as efficient as bottom-up enumeration? The question is relevant in practice as at least two companies —Tandem and Microsoft— are adopting a transformation-based optimization engine for new releases of their DBMSs. The goal is to have cost-based selection of alternatives that are generated via transformation rules, without performance penalties on join reordering.

In this paper we show that join enumeration can be done in the lower bound given by Ono and Lohman [OL90], in a transformation-based system. The two key components for our result are the *memoizing structure* proposed in Volcano [GM93, McK93], and a novel technique to *avoid generating duplicate expressions*.

Duplicates are a major problem in transformation-based enumeration. To generate a complete space, the general algorithm is to apply all possible transformation rules, until no new information is produced. To get a sense of the problem, model the search space as a graph, where each solution is a node, and transformation rules provide edges between those nodes. Now, the number of duplicates generated depends on the size of the number of nodes in the space, n, and the number of neighbors, b_i of each node. The naive application of transformation rules until no new elements are generated results in the generation of $\sum_{i=1}^{n} b_i$ alternatives. For simplicity, assume that the number of neighbors for each alternative is the same ($b = b_1 = b_2 = ... = b_n$), then we get $b * n$ alternatives generated. Since the are only n nodes in the space, the number of duplicates is $n * (b - 1)$. Only 1 out of every b trees generated is new, and $(1 - 1/b)$ of the plans generated —i.e. *most of them* as b increases— are duplicates.

A note on terminology.

Some terms in the "rule-based optimizer literature" are applied loosely to systems with different characteristics. In the context of this paper we contrast *bottom-up join enumeration* with *transformation-based enumeration*. Transformation-based enumeration consists of generating all alternatives reachable from an initial algebraic expression by a set of transformation rules; so that their estimated cost can be used in choosing one of them.

The paper is organized as follows. In Section 2 we describe the memoizing structure of Volcano [GM93, McK93], and analyze its complexity. Section 3 identifies and quantifies the problem of duplicates. Section 4 describes how to enumerate join orders without generating duplicates. Section 5 shows experimentally the performance improvement of avoiding duplicates. Finally our conclusions are given in Section 6.

2 Memoizing

Transformation-based enumeration of a space proceeds as follows. Keep a set of *visited* plans, which starts by containing a single input expression. Apply all transformation rules to visited plans, adding the results to the set if they are new. When no new plans can be generated, the complete search space for this set of transformations has been explored.

For join reordering, the transformations commonly used (to generate a bushy space) are [BMG93, IW87, IK91, Kan91]:

Rule set **RS-B0**:

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

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 left linear trees, a minimal rule set based on [SG88] is: Rule set **RS-L1**:

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

Since there are likely to be many common subexpression among the alternatives generated, Volcano introduced a memory-efficient representation of the search space, inspired by the idea of memoizing [McK93].

2.1 The MEMO-structure in Volcano

The MEMO structure minimizes memory requirements by maximizing the sharing of common sub-trees. The main idea behind 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 produce 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 more details about this structure, see [McK93]. In this paper we used a sightly simplified version of the MEMO-structure described there.

$$\begin{aligned}
\text{abcd} &= [a] \bowtie [bcd]; [b] \bowtie [acd]; [c] \bowtie [abd];\\
&\quad [d] \bowtie [abc]; [ab] \bowtie [cd]; [ac] \bowtie [bd];\\
&\quad [ad] \bowtie [bc]; [bcd] \bowtie [a]; [acd] \bowtie [b];\\
&\quad [abd] \bowtie [c]; [abc] \bowtie [d]; [cd] \bowtie [ab];\\
&\quad [bd] \bowtie [ac]; [bc] \bowtie [ad].\\
\text{abc} &= [a] \bowtie [bc]; [b] \bowtie [ac]; [c] \bowtie [ab];\\
&\quad [bc] \bowtie [a]; [ac] \bowtie [b]; [ab] \bowtie [c].\\
\text{abd} &= [a] \bowtie [bd]; [b] \bowtie [ad]; [d] \bowtie [ab];\\
&\quad [bd] \bowtie [a]; [ad] \bowtie [b]; [ab] \bowtie [d].\\
\text{acd} &= [a] \bowtie [cd]; [c] \bowtie [ad]; [d] \bowtie [ac];\\
\text{acd} &= [cd] \bowtie [a]; [ad] \bowtie [c]; [ac] \bowtie [d].\\
\text{bcd} &= [b] \bowtie [cd]; [c] \bowtie [bd]; [d] \bowtie [bc];\\
&\quad [cd] \bowtie [b]; [bd] \bowtie [c]; [bc] \bowtie [d].\\
\text{ab} &= [a] \bowtie [b]; [b] \bowtie [a].\\
\text{ac} &= [a] \bowtie [c]; [c] \bowtie [a].\\
\text{ad} &= [a] \bowtie [d]; [d] \bowtie [a].\\
\text{bc} &= [b] \bowtie [c]; [c] \bowtie [b].\\
\text{bd} &= [b] \bowtie [d]; [d] \bowtie [b].\\
\text{cd} &= [c] \bowtie [d]; [d] \bowtie [c].
\end{aligned}$$

Figure 1: The complete MEMO-structure with bushy join orders for the completely connected query on a, b, c, d.

Figure 1 depicts the MEMO-structure encoding the search space of a four-table join query. For convenience, the classes are labeled with the relations that are being joined. The memo structure for the join space corresponds closely to the structures kept by Vance and Maier in their work [VM96].

The MEMO-structure has 11 equivalence classes, namely "abcd", "abc", "abd", "acd", "bcd", "ab", "ac", "ad", "bc", "bd", "cd", with the first class containing 14 join operators. An operator tree is obtained from a MEMO-structure by choosing a specific operator at each level.

Base relations are not shown as operators, although they are operators in an implementation. Children of join operators must be classes, so the base table operator is contained in a one-operator class.

The MEMO-structure helps ameliorate the combinatorial explosion of alternative join orders. For a *completely connected* join query with n relations, the number of alternative ordered bushy and linear join trees is known to be $\frac{(2n-2)!}{(n-1)!}$ and $n!$, respectively [LVZ93, GLPK95]. A completely connected query of 7 relations then already leads to 5040 alternative linear join trees and 665280 bushy join trees, see Figure 2 for the number of join trees and operators for both linear and bushy evaluation orders at several query sizes.

	Linear join trees		Bushy join trees	
Rel	JT	Op	JT	Op
2	2	1	2	2
3	6	6	12	12
4	24	22	120	50
5	120	65	1680	180
6	720	171	30240	602
7	5040	420	665280	1932

Figure 2: Number of ordered bushy and linear join trees for a completely connected query of n relations.

2.2 Size of the MEMO-structure

In comparison to the total number of feasible evaluation orders, the MEMO-structure is an efficient way of encoding the seach space. The following two theorems give the number of join operators in the MEMO-structure to encode all bushy or left linear join trees. The query graph is assumed to be completely connected. The proofs are omitted due to lack of space, but they can be found in [PGLK96], which contains results for other query graph topologies.

Theorem 1 *The MEMO-structure requires* $3^n - 2^{n+1} + 1$ *operators to encode the space of bushy join trees for a completely connected query of* n *relations.*

Theorem 2 *The MEMO-structure requires* $n2^{(n-1)} - n(n+1)/2$ *operators to encode the space of linear join trees for a completely connected query of* n *relations, for* $n > 2$.

In [OL90] a lower bound was determined for these combinations of query graph topologies and join tree shapes by counting how many join operators had to be considered by their dynamic programming algorithm. Our findings coincide with their lower bounds.[2] Also the other cases coverd by Ono and Lohman coincide with our findings and are described in [PGLK96].

[2]There is a factor of 2 difference, due to the fact that Ono and Lohman do not count $A \bowtie B$ and $B \bowtie A$ as distinct operators, i. e. they use unordered trees. Transformation-based enumeration can generate unordered trees as well [PGLK96], but we use ordered trees here for consistency with conventional rule sets (see rule set **RS-B0**, it requires ordered trees to work). In practice the difference is minor, because unordered trees will most likely consider both children as candidates for, say, the build input of hash join, in a sense delaying the commutativity application.

2.3 Exploration process

A complete MEMO-structure — encoding a complete space — is constructed by recursively exploring the roots of operator trees, starting with an initial expression. Exploring an operator is done by exhaustively applying all transformation rules to generate all alternatives. A detailed description of the exploration algorithm is given in [McK93].

In general, the application of a transformation rule can generate an operator that is already present in the MEMO-structure. The simplest example is the commutativity rule, which, when applied a second time, 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 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 3 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".

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".

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

Figure 3: MEMO-structure before and after exploration.

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 4 the complete "abc" class is shown together with the duplicates generated. The duplicates are positioned such that they are next to the non-duplicate operator from which they originated.

Class abc	Duplicates in class abc
$[ab] \bowtie [c]$	
$[c] \bowtie [ab]$	$[ab] \bowtie [c]$
$[a] \bowtie [bc]$	
$[b] \bowtie [ac]$	
$[bc] \bowtie [a]$	$[a] \bowtie [bc]; [b] \bowtie [ac]; [c] \bowtie [ab]$
$[ac] \bowtie [b]$	$[b] \bowtie [ac]; [a] \bowtie [bc]; [c] \bowtie [ab]$

Figure 4: Fully explored class "abc" and the duplicates generated.

■

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 search cost.

3.1 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 minumal set of unidirectional join associativity and commutativity rules, See 2.3. 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 $[R]$ $(0 < k < n)$. 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 $\sum_{l=1}^{k-1} \binom{k}{l} (2^l - 1)$. 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 3 *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 $\binom{n}{k}$ classes that combine k relations. In the class with only one relation no duplicates are generated since no transformation rules are applied. Using Lemma 1 the total number of duplicates generated is $\sum_{k=1}^{n} \binom{n}{k} (3^k - 3 * 2^k + 4)$. Rewriting results in: $4^n - 3^{n+1} + 2^{n+2} - n - 2$. ■

3.2 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:* $k^2 - k + 1$, *with* $k > 2$.

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 $0 < k < n$. Since we are considering linear join trees for each operator either $l = 1$ or $r = 1$. If $l = 1$ only the commutativity rule can be applied, if $r = 1$ also the swap rules applies and generates $k - 1$ operators. Both cases happen k times, so in a class $k * 1 + k * (1 + k - 1) = k^2 + k$ operator are generated.
In a class there are only $2k$ unique operators and one of these, the initial operator, is already given. This brings the number of duplicates per class to $k^2 + k - (2k - 1) = k^2 - k + 1$. ■

Theorem 4 *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:* $2^n + n(n-1)2^{n-2} - 1 - n^2$, $n > 2$.

Proof. The MEMO-structure consists of $\binom{n}{k}$ classes that combine k relations. In the class with only one relation no duplicates are generated since no transformation rule can be applied. In the class with two relations only the commutativity rule can be applied and results in one duplicate. Since there are $\binom{n}{2}$ such classes, $\frac{n(n-1)}{2}$ duplicates are generated.
For $k > 2$ we use Lemma 2 so the total number of duplicates generated is $\sum_{k=3}^{n} \binom{n}{k} (k^2 - k + 1)$. Rewriting and adding the duplicates generated by the classes with two relations results in: $2^n + n(n-1)2^{n-2} - 1 - n^2$. ■

	Linear join trees		Bushy join trees	
Rel	Op	Duplicates	Op	Duplicates
2	1	1	2	1
3	6	10	12	10
4	22	47	50	71
5	65	166	180	416
6	171	517	602	2157
7	420	1422	1932	10326

Figure 5: Number of duplicates generated during the exploration of a MEMO-structure.

Figure 5 shows concrete numbers for the size of the MEMO-structure and the duplicates generated (both buhsy 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 Section 2.2 and Theorem 3 shows that for buhsy join trees the ratio of duplicates over new elements is $O(2^{n\log(4/3)})$. Also for linear join trees the number of duplicates outgrows the number of unique operators quickly.

4 Duplicate-free join order generation

Although the space complexity of transformation-based join enumeration is $O(3^n)$, by Theorem 1, the enumeration process itself takes $O(4^n)$, by Theorem 2. In this Section we show how to avoid generating duplicates. This makes the enumeration process as efficient as the lower bound of $O(3^n)$ given by Ono and Lohman.

To avoid the generation of duplicates, information about the behavior of transformation rules is used. The simplest example is commutativity: 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.[3]

In general, we need to keep track of the "derivation history" of each operator, or conversely, which rules 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 operator.

4.1 Duplicate free transformation rules for completely connected queries

To generate all alternative bushy join trees for completely connected query graphs we use the following transformation rules:

Rule set **RS-B2**:

R_1 : **Commutativity** $x \bowtie_0 y \to y \bowtie_1 x$
Disable all rules R_1, R_2, R_3, R_4 for application on the new operator $\bowtie_1$.

R_2 : **Right associativity**
$(x \bowtie_0 y) \bowtie_1 z \to x \bowtie_2 (y \bowtie_3 z)$
Disable rules R_2, R_3, R_4 for application on the new operator $\bowtie_2$.

R_3 : **Left associativity**
$x \bowtie_0 (y \bowtie_1 z) \to (x \bowtie_2 y) \bowtie_3 z$
Disable rules R_2, R_3, R_4 for application on the new operator $\bowtie_3$.

R_4 : **Exchange**
$(w \bowtie_0 x) \bowtie_1 (y \bowtie_2 z) \to (w \bowtie_3 y) \bowtie_4 (x \bowtie_5 z)$
Disable all rules R_1, R_2, R_3, R_4 for application on $\bowtie_4$.

For example, consider a completely connected query on the relations w, x, y and z. Using the initial element $[wx] \bowtie [yz]$ of a class and the fully explored classes of "wx" and "yz" the four transformation rules generate six sets of elements as shown in Figure 6.

$$\begin{tikzcd}
{[wx] \bowtie [yz]} \arrow[r, "R_1"] \arrow[dr, "R_2"] \arrow[dddr, "R_3"] \arrow[dddddr, "R_4"] & {[yz] \bowtie [wx]} & \\
 & {[w] \bowtie [xyz]} \arrow[r, "R_1"] & {[xyz] \bowtie [w]} \\
 & {[x] \bowtie [wyz]} & {[wyz] \bowtie [x]} \\
 & {[wxy] \bowtie [z]} \arrow[r, "R_1"] & {[z] \bowtie [wxy]} \\
 & {[wxz] \bowtie [y]} & {[y] \bowtie [wxz]} \\
 & {[wy] \bowtie [xz]} & \\
 & {[wz] \bowtie [yz]} & \\
 & {[xy] \bowtie [wz]} & \\
 & {[xz] \bowtie [wy]} &
\end{tikzcd}$$

Figure 6: Sets generated by the transformation rules

Rule R_2 combines each operator of class "wx" with the right operand $[yz]$. Rule R_3 does a similar thing for the operators of class "yz". Rule R_4 combines the operators of class "wx" with the operators of class "yz" and rule R_1 generates the mirror images for the elements generated by rule R_2, R_3 and the initial element.

Keeping a summary of the derivation history for each operator increases the memory requirements. However, the applicability of a rule can be encoded using a single bit. With four transformation rules each operator needs 4 extra bits of memory to store the derivation history. Alternatively, rules R_2 and R_3 could be modified so they generate two substitutes instead of only one, for each pattern, and skip a later application of R_1.

[3]This observation, for commutativity, was made early on in [GD87], where it led to the idea of "unidirectional transformation rules." An idea that has taken several forms in follow up projects. Cycles of length 2 in the search space are easy to avoid, but the advantage of excluding only those short cycles is marginal. In terms of complexity, it is easy to see that they introduce a constant factor slowdown, so their removal does not affect the complexity. In practice, we ran experiments where those cycles were avoided, but this never made a difference of more than 2% in performance, with queries of up to 8 tables. *Most* duplicates are generated by larger cycles in the search graph.

Theorem 5 *No duplicates are generated when the transformation rules, R_1, R_2, R_3 and R_4, are applied as described.*

Proof. Two operators can not be identical if they are both generated by the same rule — i.e. 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. Rule R_4 combines the unique operators of the left and right operand resulting in only unique operators.
Also, no two derivation paths can result in the same operator — i.e. elements of different sets. Suppose the application of rule R_2 generated the same element as rule $R_3; R_1$, 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. A similar argument can be given for any other combination of sets. ∎

Theorem 6 *For completely connected queries the transformation rules R_1, R_2, R_3 and R_4 generate all valid bushy join orders.*

Proof. In a fully explored class that references n relations the number of join operators is $\mathcal{A}(n) = 2^n - 2$ (See proof of lemma 1). Using the initial element of a class, say $[L] \bowtie [R]$ the transformation rules generate the following elements. Rule R_2 combines each *element* of class $[L]$ with $[R]$ resulting in $\mathcal{A}(|L|)$ new operators. Similarly rule R_3 generates $\mathcal{A}(|R|)$ new elements. Rule R_4 combines each *element* of class $[L]$ with each *element* of class $[R]$ which results in $\mathcal{A}(|L|) * \mathcal{A}(|R|)$ 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: $2 + 2 * \mathcal{A}(|L|) + 2 * \mathcal{A}(|R|) + \mathcal{A}(|L|) * \mathcal{A}(|R|)$. Rewriting shows that $2 + 2 * \mathcal{A}(|L|) + 2 * \mathcal{A}(|R|) + \mathcal{A}(|L|) * \mathcal{A}(|R|) = \mathcal{A}(|L| + |R|)$ which is the number of elements for the fully explored class with $|L| + |R|$ relations. Since, by Theorem 3, no duplicates are generated we must have generated all valid bushy join orders. ∎

4.2 Example

Given a completely connected query on five relations $\{a, b, c, d, e\}$ and the MEMO-structure as shown in Figure 7, where class "abcde" is about to be explored. Of the initial operator of class "abcde", $[ab] \bowtie [cde]$, the child classes have been explored exhaustively.

abcde	$= [ab] \bowtie [cde]$
cde	$= [c] \bowtie [de]; [de] \bowtie [c]; [d] \bowtie [ce];$ $[ce] \bowtie [d]; [e] \bowtie [cd]; [cd] \bowtie [e]$
ab	$= [a] \bowtie [b]; [b] \bowtie [a]$
cd	$= [c] \bowtie [d]; [d] \bowtie [c]$
ce	$= [c] \bowtie [e]; [e] \bowtie [c]$
de	$= [d] \bowtie [e]; [e] \bowtie [d]$

Figure 7: MEMO-structure in which $[ab] \bowtie [cde]$ is about to be explored.

The exploration process starts by applying rules R_2, R_3 and R_4 to $[ab] \bowtie [cde]$, then R_1 is applied to generate the mirror images. This results in the following elements.

R_2: Each operator of the left subtree $[ab]$, is combined with the right subtree $[cde]$ to obtain $[a] \bowtie [bcde], [b] \bowtie [acde]$, which are added to class "abcde".

R_3: Combines each operator of the right subtree $[cde]$ with the left subtree $[ab]$ to obtain: $[abc] \bowtie [de], [abde] \bowtie [c], [abd] \bowtie [ce], [abce] \bowtie [d], [abe] \bowtie [cd], [abcd] \bowtie [e]$, which are also added to class "abcde".

R_4: Combines the operators of the left subtree with each split of the right subtree, so we obtain: $[ac] \bowtie [bde], [ade] \bowtie [bc], [ad] \bowtie [bce], [ace] \bowtie [bd], [ae] \bowtie [bcd], [acd] \bowtie [be]$ and their mirror images $[bde] \bowtie [ac], [bc] \bowtie [ade], [bce] \bowtie [ad], [bd] \bowtie [ace], [bcd] \bowtie [ae], [be] \bowtie [acd]$.

R_1: Generates all the mirror images of the original join operator and the operator generated by R_2 and R_3.

Now the fully explored class "abcde" contains 30 operators. During the exploration 20 new classes were created and, in turn, fully explored.

Linear join trees

To generate all linear trees efficiently — without duplicates — we use the following two transformation rules and application schema. The proofs for completeness and efficiency are omitted but are similar to proofs for the case of bushy join trees.

Rule $\mathbf{R_1}$:

$(A \Join_0 B) \Join_1 C \to (A \Join_2 C) \Join_3 B$.
Disable rule R_1 for application on operator $\Join_3$.
A, B and C are classes which reference one or more relations.

Rule $\mathbf{R_2}$:

$(Table_1 \Join_0 Table_2) \to (Table_2 \Join_1 Table_1)$.
Disable rule R_2 for application on operator $\Join_1$.
$Table_1$ and $Table_2$ are classes which reference exactly one relation.

5 Experiments

In this section we experimentally verify the efficiency improvement of the duplicate free join enumeration process. For completely connected queries, from 3 to 8 relations, we generated all bushy and linear join trees using both sets of transformation rules — the naive transformation rules and the duplicate-free rules.

The measurements have been performed on a 90 MHz Pentium PC running Windows NT. It's main memory was 64Mb which was more than enough to contain the largest MEMO-structure — all bushy trees for a query of 8 relations. The measurements have been performed using the Cascades optimizer, which is a descendent of the Volcano optimizer. However no feature was used that wasn't already present in Volcano. The one modification on the domain-independent, transformation-rule kernel was to add the ability to disable transformation rules. The remainder of the logic is done completely within the domain-specific set of transformation rules.

For each unique "logical" join operator we also generated a single "physical" operator (Nested Loop). For each physical operator some cost estimation was done — i.e. cardinality — which makes the generation of a physical operator more expensive than the generation of a logical operator. However, the estimation cost are constant per physical operator and is only performed for *unique* operators and not for duplicates. Avoiding the generation of physical plans would make the improvement factor even bigger.

Each experiment has been performed several times and the graphs represent the averages over these runs. The variation amongst the runs was very small, less then 0.5%,

Bushy join trees.

For the naive generation of bushy join trees we used the commutativity and associativity rules as described in Section 2.3. In [GD87] it was already observed that the performance of the join enumerator could be improved by applying the commutativity rule only once. This avoids all generation cycles of length two — i.e. cycles like $(a \Join b) \to (b \Join a) \to (a \Join b)$. However the improvement is very small, for 8 relations the improvement is less than 2%. When generating all bushy trees using the naive set of rules, cycles of length 2 were avoided.

Figure 8 shows the (scaled) time required to generate all bushy trees for completely connected queries from 3 to 8 relations. The scaling of the graphs is done using the time to generate all bushy trees for a query of tree relations, as a reference.

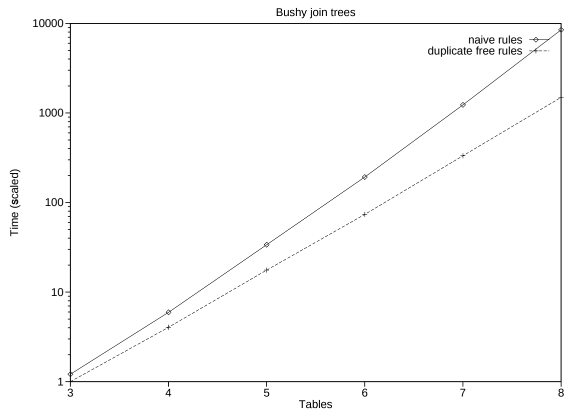

Figure 8: Exhaustive generation of bushy trees for completely connected queries.

The experiments show that duplicate free generation of join orders is always faster than generating and discarding duplicates. The performance gain increases from a factor 1.22 for three relations to a factor 5.67 for eight relations. Based on the complexity analysis of the generation algorithms the improvement factor will increase further as queries get larger.

Linear join trees.

The naive method for generating the complete space of linear join trees uses Swami's [SG88] "Swap" rule — i.e. $(A \Join B) \Join C \to (A \Join C) \Join B$ —and the commutativity rule. As in the naive generation of bushy join trees the commutativity rule is applied only once to avoid cycles of length 2.

Figure 9 shows the experimental results for generating all linear trees using the naive method, in which duplicates are generated, and the efficient method that avoids the generation of duplicates. The time for generating all linear join trees for a query of tree relations, using the duplicate-free rules, was used as reference for scaling the graphs. For linear trees, avoiding the generation of duplicates shows a performance improvement of a factor 1.33 to 3.67 for queries from three to eight

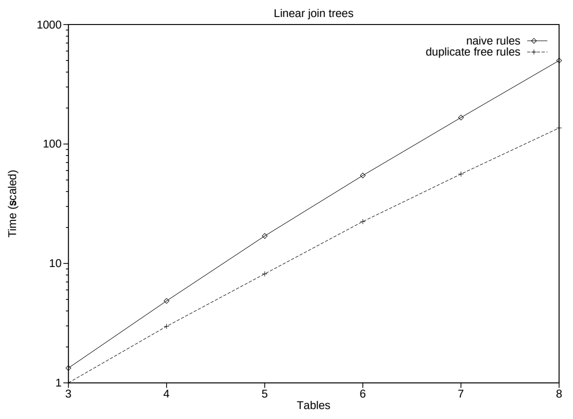

Figure 9: Exhaustive generation of linear trees for completely connected queries.

relations. Again the improvement factor will keep increasing with the number of relations.

6 Conclusion

In this paper we showed that the join enumeration process of transformation-based optimizers can be made as efficient as the lower bound of $O(3^n)$ given for the problem by Ono and Lohman [OL90], as long as we avoid generating duplicates. We showed that the number of duplicates is $O(4^n)$, and it exceeds the number of new elements even for small queries.

Our approach to an efficient search is to keep track of the transformation rules that can still be applied without generating duplicates. We descibed the mechanism in detail, for the generation of bushy and linear join trees. Our experiments demonstrated a significant improvement in optimization time, as large as a factor of 5 for 8-table joins.

The implementation of our approach was relatively simple, requiring only a minor extension to the transformation-rule engine —the ability to turn off transformation rules. Of course, most of the work went into devising the appropriate set of transformation rules, which is a very important task, seldom emphasized in the rule-base optimization literature.

There is considerable work left on the general problem of duplicates and how to avoid them, and we believe this is a promising area of research. For an arbitrary set of transformation rules, it might be hard to transform it into an efficient, duplicate-avoiding set. But we suspect there are useful, tractable classes.

Acknowledgements

Bennet Vance considered the problem of duplicates and arrived at the Exchange transformation rule independently of us, using it in a slightly different way [Van96b]. We are grateful to him for estimulating discussions, sharing of his results [Van96a], and comments on this paper.

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.

[CS96] S. Chaudhuri and K. Shim. Optimizing queries with aggregate vieuws. *International Conference on Extending Database Technology, Avignon, France*, pages 167–182, 1996.

[GD87] G. Graefe and D. J. DeWitt. The exodus optimizer generator. *Proc. of the ACM-SIGMOD Conference on Management of Data*, pages 160–172, 1987.

[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.

[GLR96] C. A. Galindo-Legaria and Arnon Rosenthal. Outerjoin simplification and reordering for query optimization. *To appear in ACM Transactions on Database Systems*, 1996.

[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.

[HN96] J.M. Hellerstein and J.F. Naughton. Query execution techiques for caching expensive methods. *Proceedings of the ACM SIGMOD Conf on Management of Data, Montreal*, pages 423 – 434, 1996.

[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.

[McK93] W. J. McKenna. *Efficient Search in Extensible Database Query Optimization: The Volcano Optimizer Generator*. PhD thesis, University of Colorado, Boulder, 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.

[PGLK96] A. Pellenkoft, G.A. Galindo-Legaria, and M.L. Kersten. Complexity of transformation based optimizers and duplicate free generation of alternatives. Technical Report CS-R9639, CWI, 1996.

[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.

[SHP$^+$96] P. Seshadri, J.M. Hellerstein, H. Pirahesh, T.Y.C. Leung, R. Ramakrishnan, D. Srivastava, P.J. Stuckey, and S. Sudarshan. Cost-based optimization for magic: Algebra and implementation. *Proceedings of the ACM SIGMOD Conf on Management of Data, Montreal*, pages 435–446, 1996.

[Van96a] B. Vance. Complexity of join enumeration in starburst, 1996. Manuscript.

[Van96b] B. Vance. Personal communication, 1996.

[VM96] B. Vance and D. Maier. Rapid bushy join-order optimization with cartesian products. In *Proceedings of the ACM SIGMOD Conf on Management of Data, Montreal*, pages 35–46, 1996.