Duplicate-free Generation of Alternatives in Transformation-based Optimizers

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 transformation rules may end up deriving the same element. The optimizer must detect and discard these duplicate elements generated by multiple paths.

In this paper we consider two questions: How bad is the overhead of duplicate generation? And then, how can it be avoided? We use a restricted class of join reordering to illustrate the problem.

For the first question, our analysis shows that as queries get larger, the number of duplicates 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.

Keywords Query optimization, Transformation-based optimization, Exhaustive search.

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 is crucial.

To generate the complete space of alternatives, the general algorithm is to maintain a set of *visited* plans. All transformation rules are applied on visited plans, 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 that it took to generate it and then find it in the set of visited plans is part of the overhead of a generic transformation-based search.

How expensive is this overhead? How frequently are we generating duplicates? We refine our analysis of duplicates later on, for a more realistic optimization framework. For now, 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 neighbor 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.

In particular, in a query joining 5 relations, each operator tree has 4 to 7 neighbors, using the "standard" associativity and commutativity rules to generate all "bushy" join orders. If we explore the space of alternatives for this query exhaustively, detecting and discarding duplicates, then we'll be discarding between 75% and 86%

Proceedings of the Fifth International Conference on Database Systems for Advanced Applications, Melbourne, Australia, April 1–4, 1997.

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 restricted case 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 3 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 4.

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. The Volcano system consists of a single optimization layer to create a truly flexible optimizer which can easily be adapted to new data-models and algebras. The optimizer's choices are represented as algebraic equivalences — transformation rules — and can be changed easily. One of the Volcano's search strategies is an exhaustive generation of alternatives, this is achieved by applying transformations until no new solutions are generated.

During this exhaustive generation, information about the alternatives explored is stored in a *look-up* table. This table is an efficient storage structure for storing (partial) search spaces [GCD+94]. The following sections describes this look-up table and its use in more detail.

2.1 Look-up table

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 evaluation orders 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 look-up table 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".

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

Figure 1: The complete look-up table for $\{A - B, A - C, B - C\}$.

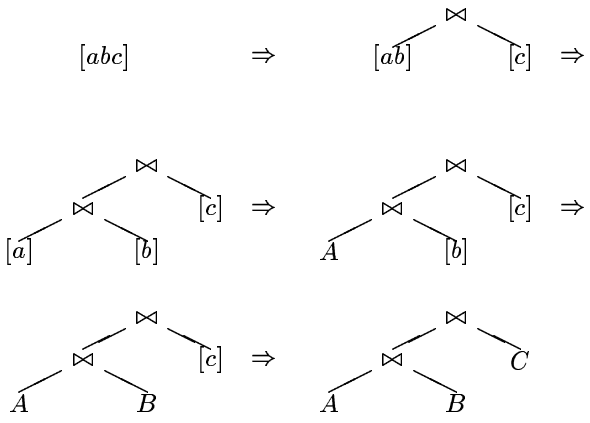

Figure 2: Operator tree extraction from a look-up table.

For example, the look-up table 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 look-up table by choosing a specific operator at each level. For example, the tree from Figure 2 is extracted from the look-up table in Figure 1 by *always selecting the first operator from a class.*

In comparison to the total number of feasible evaluation orders the look-up table is an efficient way of storing the explored alternatives. However, the size of the look-up table is still considerable. Theorem 1 considers the completely connected

graph to determine the upper bound on the size of the look-up table in case bushy evaluation orders are allowed. The look-up table is the most important and largest data structure used, so Theorem 1 is also an upper bound on the memory requirements of such optimizers.

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

Proof. The upper bound on the size of the look-up table 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 roots 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 $2^k - 2$. Now the number of operators in the look-up table is the sum of all elements of all classes, which is: $\sum_{k=2}^{n} \left(\begin{array}{c} n \\ k \end{array} \right) (2^k - 2) + n = 3^n - 2^{n+1} + n + 1$. ∎

Also [OL90] considered the number of feasible join orders for completely connected query graphs when bushy evaluation orders are allowed. They found the maximum number of join operators to be $(3^n - 2^{n+1} + 1)/2$. The difference is caused by the fact that the look-up table distinguishes between the left and right input of join operators and generates a class for each of the n base relations.

Relations	Join orders	Operators	Duplicates
2	2	4	1
3	12	15	10
4	120	54	71
5	1680	185	416
6	30240	608	2157
7	665280	1939	10326

Figure 3: Number of operators needed to represent all alternative evaluation orders for a completely connected query of n relations.

For completely connected queries of several sizes, Figure 3 shows the number of alternative join orders and the number of operators needed to encode those trees using a look-up table. The last column shows the number of duplicates generated and is explained in Section 2.4

2.2 Exploration process

To generate all alternative join evaluation orders, starting from a single join tree, we need two basic transformation rules, namely: commutativity $(x \bowtie y) \rightarrow (y \bowtie x)$ and associativity $((x \bowtie y) \bowtie z) \rightarrow (x \bowtie (y \bowtie z))$. This rule set is known to be complete, i.e. these rules are sufficient to generate all possible join evaluation orders for a given query.

A complete look-up table — 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 general approach as described in Section 1.

Figure 4 shows the exploration algorithm. The initial look-up table 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.

```
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
        look-up table;
        the root of ê is e; and
        ê matches the pattern of R
      x̂ := apply R on ê;
    if x̂ ∉ look-up table
      add x̂ to lookup-table;
      (place the root of x̂ in the
      same class as e)
  }
}
```

Figure 4: Exploration algorithm

In general, the application of a transformation rule can generate an operator which is already

present in the look-up table. For example, applying the commutativity rule twice reproduces the original operator. So, before inserting a new operator into the look-up table we have to make sure it is not already present. A hash table is used to speed-up the detection of duplicates.

2.3 Exploration example

To see how duplicates are generated an action trace is described in detail. For the completely connected query on the relations "a, b, c," Figure 5 shows the look-up tables before and after exploring operator $[ab] \bowtie [c]$. In the "before" look-up table all children, "ab" and "c", have been fully explored. The transformation rules (see Section 2.2) 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 look-up table in class "abc". The subexpression ($[b] \bowtie [c]$) starts a new class "bc" since it didn't appear in the look-up table.
- $[b] \bowtie [a]$**:** Second tree (($[b] \bowtie [a]$) $\bowtie$ $[c]$) is extracted. It matches the rule, so it is applied. The new tree generated is ($[b] \bowtie ([a] \bowtie [c])$) and is added to the look-up table. The subexpression $[a] \bowtie [c]$ starts a new class "ac".

Before	After
abc=$[ab] \bowtie [c]$.	abc=$[ab] \bowtie [c]$; $[c] \bowtie [ab]$; $[a] \bowtie [bc]$; $[b] \bowtie [ac]$; $[bc] \bowtie [a]$; $[ac] \bowtie [b]$.
ab= $[a] \bowtie [b]$; $[b] \bowtie [a]$.	ab= $[a] \bowtie [b]$; $[b] \bowtie [a]$.
	bc= $[b] \bowtie [c]$; $[c] \bowtie [b]$.
	ac= $[a] \bowtie [c]$; $[c] \bowtie [a]$.
a= $get(A)$.	a= $get(A)$.
b= $get(B)$.	b= $get(B)$.
c= $get(C)$.	c= $get(C)$.

Figure 5: look-up table 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.

To the new operators only the commutativity rule applies, which results in the operators $[ab] \bowtie [c]$, $[bc] \bowtie [a]$ and $[ac] \bowtie [b]$ of which $[ab] \bowtie [c]$ already exists. The new operators are added to the look-up table 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 look-up table. So, during the exploration of class "abc", 5 new operators and 7 duplicates were generated. In Figure 6 the operator generation graph shows which operators are generated by the commutativity rule (R_c) and the associativity rule (R_a). The bold operators are the duplicates.

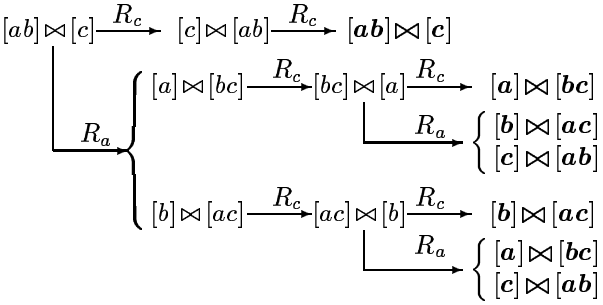

Figure 6: Operator generation graph.

2.4 Duplicates generated

We derived a factor of $b - 1$ of duplicate elements over new elements in the simple graph model of the introduction, where we also gave an example for a query of 5 relations. The naive model used there, however, did not take into account the look-up table used by the Volcano-type optimizers. The next Theorems deal with the details of the new structure.

Lemma 1. *The number of duplicates generated 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 $3^k - 2^{k+1} + 1$. The maximum 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 in the class is: $3^k - 2^{k+1} + 1 - (2^k - 2 - 1) = 3^k - 3 * 2^k + 4$. ■

Theorem 2. *The number of duplicates generated during the construction of a look-up table encoding the alternatives for a query with n relations, on a completely connected graph, is:* $4^n - 3^{n+1} + 2^{n+2} - n - 2$.

Proof. The look-up table consists of $\left(\begin{array}{c} n \\ k \end{array} \right)$ classes that combine k relations. In the class with only one relation no duplicates are generated since no transformation rule can be applied. Using Lemma 1 the total number of duplicates generated is $\sum_{k=1}^{n} \left(\begin{array}{c} n \\ k \end{array} \right) (3^k - 3 * 2^k + 4)$. Rewriting results in: $4^n - 3^{n+1} + 2^{n+2} - n - 2$. ■

Figure 3 shows concrete numbers for the size of the look-up table and the duplicates generated, as a function of the number of relations joined, for fully connected graphs. The second column gives the number of bushy join trees, without considering any sharing. The number of operators needed to encode these trees in the look-up table is given in column 3. The advantage of sharing common subtrees in the look-up table is clear. The last column shows the number of duplicate operators generated during exploration. Theorem 1 and 2 show that the ratio of duplicates over new elements is $O(2^{n \log(4/3)})$.

3 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 [GL95]. For example, the application of the commutativity rule will switch the commutativity rule off in the rule set of the resulting element.

In the next section, we present a set of transformation rules, together with an application schema, that generates all alternative "bushy" join trees for acyclic join queries, without generating duplicates.

3.1 Duplicate free transformation rules for acyclic queries

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

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. Sets 1,2 and 3 are generated using the initial element. Sets 4 and 5 are generated using the elements of set 1 and 2. Join $[wx] \bowtie [yz]$ must be a valid join tree (i.e. no Cartesian products) of an acyclic query graph.

Set 1: $[wx] \bowtie [yz] \stackrel{R_2}{\rightarrow} \{[w] \bowtie [xyz]\}$. Associativity produces 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 subquery 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 associativity rule used for Set 2 below.

Set 2: $[wx] \bowtie [yz] \stackrel{R_3}{\rightarrow} \{[wxy] \bowtie [z]\}$.

Set 3: $[wx] \bowtie [yz] \stackrel{R_1}{\rightarrow} [yz] \bowtie [wx]$.

Set 4: $\{[w] \bowtie [xyz]\} \stackrel{R_1}{\rightarrow} \{[xyz] \bowtie [w]\}$.

Set 5: $\{[wxy] \bowtie [z]\} \stackrel{R_1}{\rightarrow} \{[z] \bowtie [wxy]\}$.

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 per operator. With three transformation rules each operator then only needs 3 bits of memory to store the derivation history.

Theorem 3. *No duplicates are generated when applying the transformation rules R_1, R_2 and R_3.*

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.
Also, no two derivation paths can result in the same operator (elements of different sets). Suppose the application of rule R_2 (Set 1) generated the same element as rule $R_3; R_1$ (Set 5), then $[w] \bowtie [xyz]$ has to be equal to $[z] \bowtie [wxy]$. This can not be true since w, x, y, z are disjunct non-empty sets of relations, so $[w] \neq [z]$ and $[xyz] \neq [wxy]$. A similar argument can be given for any other combination of sets. ∎

Theorem 4. *For acyclic query graphs the transformation rules R_1, R_2 and R_3 generate all valid join orders.*

Proof. Since the query graph is acyclic, each join operator that can serve as root for a valid join tree corresponds 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 $\mathcal{A}(n) = 2 * (n-1)$, 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 $[L] \bowtie [R]$, the transformation rules generate the following new elements. Rule R_2 combines each *element* of class $[L]$ with $[R]$, 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 $\mathcal{A}(|L|)/2$ new operators, with $|L|$ denoting the number of operators of class L. Similarly rule R_3 generates $\mathcal{A}(|R|)/2$ new operators. Finally rule R_1 generates for each new operator and the initial operator a mirror image which results in $2*(1+\mathcal{A}(|L|)/2+\mathcal{A}(|R|)/2) = 2+\mathcal{A}(|L|)+\mathcal{A}(|R|)$ elements for the fully explored class.
Now, $2 + \mathcal{A}(|L|) + \mathcal{A}(|R|) = \mathcal{A}(|L| + |R|)$, which is the number of join operators for the fully explored class with $|L| + |R|$ relations. Since, by Theorem 1, no duplicate operators were produced, we must have generated all valid join orders. ∎

3.2 Acyclic example

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

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

Figure 7: Look-up table in which $[ab] \bowtie [cde]$ is about to be explored.

Applying the transformation rules R_1, R_2 and R_3 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: $[cde] \bowtie [ab], [bcde] \bowtie [a], [d] \bowtie [abce], [e] \bowtie [abcd]$

The fully explored class "abcde" contains all 8 elements. During the exploration process the new classes $[bcde]$, $[abce]$ and $[abcd]$ are created and, in turn, fully explored.

4 Conclusion

In this paper we described 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 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.

We showed that the number of duplicates exceeds the number of new elements even for small queries, 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 search is to keep track of the transformation rules that can still be applied without generating duplicates. The mechanism is described in detail, for the generation of all valid join trees for acyclic query graphs.[1]

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. For queries of 8 relations a performance improvement of a factor 3 to 5 has been achieved. For arbitrary sets of transformation rules it might be hard to transform them into an efficient set. However, the performance improvement gained by avoiding duplicate generation is significant in practice, and it should be used whenever possible.

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

[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.

[GL95] C. A. Galindo-Legaria. Dart V4.0 Optimizer: Join Reordering, August 1995. Manuscript.

[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. *Proceedings of the 9th International Conference on Data Engineering, Vienna, Austria*, pages 209–218, 1993.

[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.

[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.

[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.

[1] We have derived duplicate-free transformation rules for other spaces [PGLK96].