Exploiting Upper and Lower Bounds in Top-Down Query Optimization

Leonard Shapiro*
Portland State University
len@cs.pdx.edu

David Maier**, Paul Benninghoff
Oregon Graduate Institute
{maier, benning}@cse.ogi.edu

Keith Billings
Informix Corporation
kgb@informix.com

Yubo Fan
ABC Technologies, Inc
yubof@abctech.com

Kavita Hatwal
Portland State University
kavitah@cs.pdx.edu

Quan Wang
Oracle Corporation
Quan.wang@oracle.com

Yu Zhang
IBM
jennyz@us.ibm.com

Hsiao-min Wu
Systematic Designs, Inc.
hswu@cs.pdx.edu

Bennet Vance

Abstract

System R's bottom-up query optimizer architecture forms the basis of most current commercial database managers. This paper compares the performance of top-down and bottom-up optimizers, using the measure of the number of plans generated during optimization. Top down optimizers are superior according to this measure because they can use upper and lower bounds to avoid generating groups of plans. Early during the optimization of a query, a top-down optimizer can derive upper bounds for the costs of the plans it generates. These bounds are not available to typical bottom-up optimizers since such optimizers generate and cost all subplans before considering larger containing plans. These upper bounds can be combined with lower bounds, based solely on logical properties of groups of logically equivalent subqueries, to eliminate entire groups of plans from consideration. We have implemented such a search strategy, in a top-down optimizer called Columbia. Our performance results show that the use of these bounds is quite effective, while preserving the optimality of the resulting plans. In many circumstances this new search strategy is even more effective than heuristics such as considering only left deep plans.

1. Introduction

The first generation of commercial query optimizers consisted of variations on System R's dynamic programming, bottom-up approach [SAC+79]. This generation had limited extensibility. For example, adding a new operator, such as aggregation, required myriad changes to the optimizer. Approximately ten years ago, researchers proposed two ways to build extensible optimizers. Lohman [Loh88] proposed using *rules* to generate plans in a bottom-up optimizer; Graefe and DeWitt [GrD87] proposed using *transforms* (the top-down version of rules) to generate new plans using a top-down approach. Lohman's generative rules were implemented in Starburst[HCL90]. Several Starburst projects have demonstrated Starburst's extensibility, from incremental joins [CSL90] to distributed heterogeneous databases [HKW97]. Since there is a huge commercial investment in engineering bottom-up optimizers like Starburst, there seems to be little motivation for investigating top-down optimizers further. It is the purpose of this paper to demonstrate a significant benefit of top-down optimizers, namely their performance, as measured by the number of plans generated during optimization.

* Supported by NSF IRI-9119446, IRI-9610013, DARPA (BAAB07-91-C-Q513) subcontract from Oregon Graduate Institute to Portland State University.

** Supported by NSF IRI-9509955, IRI-9619977, DARPA (BAAB07-91-C-Q513)

Early during the optimization of a query, a top-down optimizer can derive upper bounds for the costs of the plans it generates. For example, if the optimizer determines that a single plan for executing $A \bowtie B \bowtie C$ has cost 7, then any subplan that can participate in an optimal plan for the execution of $A \bowtie B \bowtie C$ will cost at most 7. If the optimizer can infer a lower bound greater than 7 for a group of plans, which are about to be generated, then the plans need not be generated – the optimizer knows that they cannot participate in an optimal solution. For example, suppose the optimizer determines that $A \bowtie C$, a Cartesian product, is extremely large, and the cost of just passing this huge output to the next operator is 8. Then it is unnecessary to generate any of the plans for executing $A \bowtie C$ – such plans could never participate in an optimal solution. Such upper bounds are not available to typical bottom-up optimizers since such bottom-up optimizers generate and cost all subplans before considering larger containing plans.

As we have illustrated, top-down optimizers can use upper and lower bounds to avoid generating entire groups of plans, which the bottom-up strategy would have produced. We have implemented, in an optimizer we call Columbia, a search strategy that uses this technique to decrease significantly the number of plans generated, especially for acyclic connected queries.

In Section 2 we survey related work. Section 3 describes the optimization models we will use. Section 4 describes the core search strategy of Cascades, the predecessor of Columbia. Section 5 describes Columbia's search strategy and our analysis of cases in which this strategy will most likely lead to a significant decrease in the number of plans generated. Section 6 describes our experimental results, and Section 7 is our conclusion.

2. Previous work

Figure 1 outlines the System R, bottom-up, search strategy for finding an optimal plan for the join of N tables.

```
(1)    For i = 1, …, N
(2)        For each set S containing exactly i of the N tables
(3a)           Generate all appropriate plans for joining the tables in S,
(3b)           considering only plans with optimal inputs, and
(3c)           retaining the optimal generated plan for each set of interesting physical properties.
```

Figure 1: System R's Bottom-up Search Strategy for a Join of N Tables

This dynamic programming search strategy generates O (3^N) distinct plans [OnL90]. Because of this exponential growth rate, bottom-up commercial optimizers use heuristics such as postponing Cartesian products or allowing only left-deep trees, or both, when optimizing large queries [GLS93].

Vance and Maier [VaM96] show that bottom-up optimization can be effective for up to 20 relations without heuristics. Their approach is quite different from ours. Instead of minimizing the number of plans generated, as we do, Vance and Maier develop specialized data structures and search strategies that allow the optimizer to process plans much more quickly. In their model, plan cost computation is the primary factor in optimization time. In our model, plan creation is the primary factor. Their approach is also somewhat different from Starburst's in that their outer loop (line (1) of Figure 1) is driven by carefully chosen subsets of relations, not by the size of the subsets. Vance and Maier's technique of plan-cost thresholds is similar to ours in that they use a fixed upper bound on plan costs, to prune plans. They choose this threshold using some heuristics and if it is not effective, they reoptimize. Our upper bounds are based on previously constructed plans rather than externally determined thresholds. Furthermore, our upper bounds can differ for each subplan being optimized.

Top-down optimization began with the Exodus optimizer generator [GrD87], whose primary purpose was to demonstrate extensibility. Graefe and collaborators subsequently developed Volcano [GrM93] with the primary goal of improving efficiency with memoization. Volcano's efficiency was hampered by its search strategy, which generated all logical expressions before generating any physical expressions. This ordering meant that Volcano generated O (3^N) expressions, like Starburst.

Recently, a new generation of query optimizers has emerged that uses object-oriented programming techniques to greatly simplify the task of constructing or extending an optimizer, while maintaining efficiency and making search strategies even more flexible. Examples of this third generation of optimizers are the OPT++ system from Wisconsin [KaD96] and Graefe's Cascades system [Gra95].

OPT++ compared the performance of top-down and bottom-up optimizers. But it used Volcano's $O(3^N)$ generation strategy for the top-down case, which yielded poor performance in OPT++ benchmarks. Cascades was developed to demonstrate both the extensibility of the object-oriented approach and the performance of top-down optimizers. It proposed numerous performance improvements, mostly based on more flexible control over the search process, but few of these were implemented. We have implemented a top-down optimizer, Columbia, which includes a particular optimizer implementation of the Cascades framework. This optimizer supports the optimization of relational queries, such those of TPC-D, and includes such transforms as aggregate pushdowns and bit joins [Bil97]. Columbia also includes the performance-oriented techniques described here.

Three groups have produced hybrid optimizers with the goal of achieving the efficiency of bottom-up optimizers and the extensibility of top-down optimizers. The EROC system developed at Bell Labs and NCR [MBH96] combines top-down and bottom-up approaches. Region based optimizers developed at METU [ONK95] and at Brown University [MDZ93] use different optimization techniques for different phases of optimization in order to achieve increased efficiency.

Commercial systems from Microsoft [Gra96] and Tandem [Cel96] are based on Cascades. They include techniques similar to those we present here, but to our knowledge these are the first analyses and testing of those techniques.

3. Optimization fundamentals

3.1 Operators

In this study we will consider only join operators and file retrieval operators, for two reasons. First, it is possible to describe the Columbia search strategy with just these operators. Second, the classic performance study by Ono and Lohman [OnL90] uses only these operators, and we will use the methodology of that study to compare the performance of top-down and bottom-up optimizers.

A *logical operator* is a function from the operator's inputs to its outputs. A *physical operator* is an algorithm mapping inputs to outputs.

The logical equijoin operator is denoted $\bowtie$ It maps its two input streams into their join. In this study we consider two physical join operators, namely sort-merge join, denoted $\bowtie_M$, and nested-loops join, denoted $\bowtie_N$. For simplicity we will not display join conditions [Ram00] .

We denote the logical file retrieval operator by GET(A), where A is the scanned table. The file A is actually a parameter of the operator, which has no input. Its output is the tuples of A. GET(A) has two implementations, or physical operators, namely FILE_SCAN(A) and INDEX_SCAN(A). For simplicity we will not specify the index used in the index scan.

Physical properties, such as being sorted or being compressed, play an important part in optimization. For example, a sort-merge join requires that its inputs be sorted on the joining attributes.

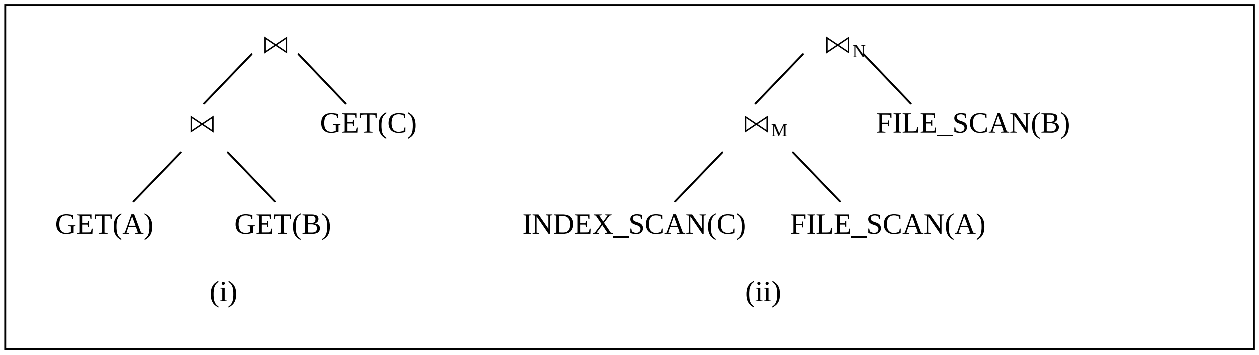


Figure 2: Two logically equivalent operator expressions

An *operator expression* is a tree of operators in which the children of an operator produce the operator's inputs; Figure 2 displays two operator expressions. An expression is *logical* or *physical* if its top operator is logical or physical, respectively. A *plan* is an expression made up entirely of physical operators. An example plan is Figure 2 (ii). We say that two operator expressions are *logically equivalent* if they produce identical results over any legal database state.

3.2 Optimization, multiexpressions, and groups

A query optimizer's input is an expression consisting entirely of logical operators, e.g., Figure 2(i) and, optionally, a set of requested physical properties on its output. The optimizer's goal is to produce an optimal plan, which might be Figure 2 (ii). An optimal plan is one that has the requested physical property, is logically equivalent to the original query, and is least costly among all such plans. (Cost is calculated by a cost model which we shall assume to be given.) Optimality is relative to that cost model.

The search space of possible plans is huge, and naïve enumeration is not likely to be successful for any but the simplest queries. Bottom-up optimizers use dynamic programming [Bel75], and top-down optimizers since Volcano use a variant of dynamic programming called memoization [Mic68, RuN95], to find an optimal plan. Both dynamic programming and memoization achieve efficiency by using the *principle of optimality:* every subplan of an optimal plan is itself optimal (for the requested physical properties). The power of this principle is that it allows an optimizer to restrict the search space to a much smaller set of expressions: we need never consider a plan containing a subplan p1 with greater cost than an equivalent plan p2 having the same physical properties. Figure 1, line (3c) is where a bottom-up optimizer exploits the principle of optimality.

The principle of optimality allows bottom-up optimizers to succeed while testing fewer alternative plans. Top-down optimization uses an equivalent technique, namely a compact representation of the search space. Beginning with Volcano, the search space in top-down optimizers has been referred to as a MEMO[McK93]. A MEMO consists primarily of two mutually recursive data structures, which we call groups and multiexpressions. A *group* is an equivalence class of expressions producing the same output. Figure 3 shows the group representing all expressions producing the output $A\bowtie B$. [1] In order to keep the search space small, a group does not explicitly contain all the expressions it represents. Rather, it represents all those expressions implicitly through multiexpressions: A *multiexpression* is an operator having groups as inputs. Thus all expressions with the same top operator, and the same inputs to that operator, are represented by a single multiexpression. In Figure 3, the multiexpression $[B]\bowtie_N[A]$ represents all expressions whose top operator is a nested loops join $\bowtie_N$ and whose left input produces the tuples of B and whose right input produces the tuples of A.

In general, if S is a subset of the tables being joined in the original query, we denote by [S] the group of multiexpressions that produces the join of the tables in S.

A *logical (physical, respectively) multiexpression* is one whose top operator is logical (physical). During query optimization, the query optimizer generates groups and for each group it finds the cheapest plans in the group satisfying the requested physical properties. It stores these cheapest plans, which we call *winners*, along with their costs and the requested properties, in the group, in a structure we call the *winner's circle*. The process of generating winners for requested physical properties is called *optimizing* the group. Figure 5 contains several groups (at an early stage in their optimization, before any winners have been found). The multiexpression $[AB] \bowtie [C]$ in Figure 5 represents (among others) the expression in Figure 2(i).

Multiexpressions: $[A]\bowtie[B]$, $[A]\bowtie_N[B]$, $[A]\bowtie_M[B]$, $[B]\bowtie[A]$, $[B]\bowtie_N[A]$, $[B]\bowtie_M[A]$
Winner's Circle:
The optimal plan, when no property is required, is $[A]\bowtie_N[B]$, and its estimated cost is 127.
There are no other winners at this time.

Figure 3: An example group [AB]

3.3 Bottom-up Optimizers: group contents and enumeration order

Bottom-up optimizers generate structures analogous to multiexpressions [Loh88]. There, the inputs are pointers to optimal plans for the properties sought. We will also use the term multiexpression, and notation like $[A]\bowtie[B]$, to denote the structures used in bottom-up optimization in which [A] and [B] are pointers to optimal plans for producing the tuples of A and B. The crucial difference between top-down and bottom-up optimizers is the order in which multiexpressions are enumerated: A bottom-up optimizer enumerates such multiexpressions from one group at a time, in the order of the number of tables in the group, as in Figure 1, lines (3a-c). If a bottom-up optimizer is optimizing the join of tables A, B and C, it will optimize groups in this order:

[A], [B], [C]; [AB], [AC], [BC]; [ABC]

[1] The costs in Figures 3 and 6 are from an arbitrary example, chosen just to illustrate the search strategies.

where the semicolons denote iterations of Figure 1, line (1). Between the semicolons, the order is controlled by line (2) and depends on the generation rules used in line (2). Note that before a single multiexpression in [ABC] is generated, all the subqueries (such as [AC]) are *completely* optimized, i.e. all optimal plans for all physical properties that are anticipated to be useful are found. Thus there is no chance to avoid generating any multiexpressions in groups such as [AC] on the basis of information gleaned from [ABC]. We will see that top-down optimizers optimize groups in a different order and may be able to use information from the optimization of [ABC] to avoid optimizing some groups such as [AC].

```
// OptimizeGroup( ) returns the cheapest physical multiexpression in Grp,
//    with property Prop, and with cost less than the upper bound UB.
// It returns NULL if there is no such multiexpression.
// It also stores the returned multiexpression in Grp’s winner’s circle.

Multiexpression* OptimizeGroup(Group Grp, Properties Prop, Real UB)
{
     // Does the winner’s circle contain an acceptable solution?
(1)  If there is a winner in the winner’s circle of Grp, for Properties Prop {
         If the cost of the winner is less than UB, return the winner
         else return NULL
     }

     // The winner’s circle does not hold an acceptable solution, so enumerate
     //   multiexpressions in Grp, using transforms, and compute their costs.
     WinnerSoFar = NULL
(2)  For each enumerated physical multiexpression, denoted MExpr {
(3)      LB = cost of root operator of MExpr
(4)      If UB <= LB then go to (2)
(5)      For each input of MExpr {
             input-group = group of current input
             input-prop = properties necessary to produce Prop from current input
(6)          InputWinner = OptimizeGroup(input-group, input-prop, UB - LB)
(7)          If InputWinner is NULL then go to (2)
(8)          LB += cost of InputWinner
         }
(9)      Use the cost of MExpr to update WinnerSoFar and UB
     }
(10) Place WinnerSoFar in the winner’s circle of Grp, for Property Prop
     Return WinnerSoFar
}
```

Figure 4: The core of Cascades’ search strategy, OptimizeGroup( )

4. Cascades’ search strategy

Figure 4 displays a simplified version of the function OptimizeGroup( ) that is at the core of Cascades’ search strategy. The goal of OptimizeGroup( ) is to optimize the group in question, by searching for an optimal physical multiexpression in Grp with the requested properties Prop and having cost less than UB.

It is nontrivial to define the cost of a multiexpression. A multiexpression’s root operator has a cost, but its inputs are groups, not expressions, and it is not clear how to calculate the cost of a group. We will see that the Cascades search strategy searches for winners – optimal solutions – by recursively searching input groups for winners. The cost of a multiexpression is thus calculated recursively, by summing the costs of the root operators of each of the winners from each of the recursive calls at line (5) of the search strategy. Let us examine the search strategy in more detail.

Line (1) checks the winner’s circle, where winners from previous OptimizeGroup( ) calls have been stored. If there is no acceptable winner in the winner’s circle, then the eventual solution WinnerSoFar is initialized and line (2) uses transforms to generate all candidate logically equivalent physical multiexpressions, corresponding to all plans generated at line (3a) of Figure 1. Line (3) calculates a lower bound LB for the cost of the multiexpression. So far LB includes only the cost of the root operator. At line (8) it will be incremented by the cost of each optimal input.

Line (6) recursively seeks a winner for each input of the candidate multiexpression. This recursive call uses as its upper bound, UB - LB, because some of the allowed cost has been used up by the cost of the root operator of the parent multiexpression, at line (3), and some by the cost of previous input winners, at line (8). For example, if OptimizeGroup( ) is seeking a multiexpression with a cost at most UB=53, and the optimizer is considering a candidate multiexpression whose root operator costs 13, then the first input must cost at most 40 to be acceptable. If the winner for the first input costs 15, then the next input can cost at most 53-28 = 25, etc.

The loop at line (5) is trying to construct acceptable inputs for the multiexpression chosen at line (2). Because the typical database operator has from 0 to 2 inputs, it typically executes at most twice. The loop can exit in two ways. First, it can exit from (4) with failure because the root operator alone costs more than the upper bound, or it can exit from (7) with failure because no acceptable winner could be found for that input (because of the bound or because of the property). Note that in this case, line (6) will not be invoked for subsequent inputs, so the groups for subsequent inputs will not be optimized. It is possible that these groups are never optimized, so none of the multiexpressions in the group will be generated. We call this *group pruning* and discuss it in Section 5 below.

Second, the loop at line (5) can exit with success, with control passing to line (9) where the resulting multiexpression is compared with WinnerSoFar. If the multiexpression has a lower cost it replaces WinnerSoFar, and the upper bound UB is set equal to the lower cost of the newly found multiexpression. This continual adjusting of upper bounds is essential to the success of our approach.

How does Cascades use OptimizeGroup( )? Cascades begins the optimization of a query by calling a function CopyIn( ) to create a separate group for each subexpression of the original input query, including leaves (see Figure 5). Then it calls OptimizeGroup( ), using as parameters the top group, whatever output properties are requested by the original query, and an infinite limit. When OptimizeGroup( ) returns, it will return the output of the query optimization, or NULL if there is no plan that is logically equivalent to the input query and that satisfies the requested properties. Since OptimizeGroup( ) returns only a multiexpression and not an actual expression, another search is necessary, using a function CopyOut( ) to retrieve winners from the winner's circles of input groups recursively, to construct the actual optimal expression from the returned multiexpression. (In fact, OptimizeGroup( ) only needs to know about the success of the recursive call in line (6), and the cost of InputWinner, so in the actual implementation we only return the cost.)

In contrast to Figure 1, Figure 4 is a top-down search strategy. It begins with the input query and, at Figure 4 line (6), proceeds top-down, using recursion on the inputs of the current multiexpression MExpr. However, plan costing actually proceeds bottom-up, based on the order of the returns from the top-down recursive calls.

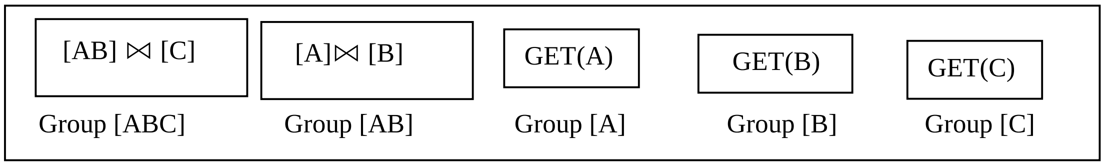


Figure 5: Cascades search space (MEMO), after initialization

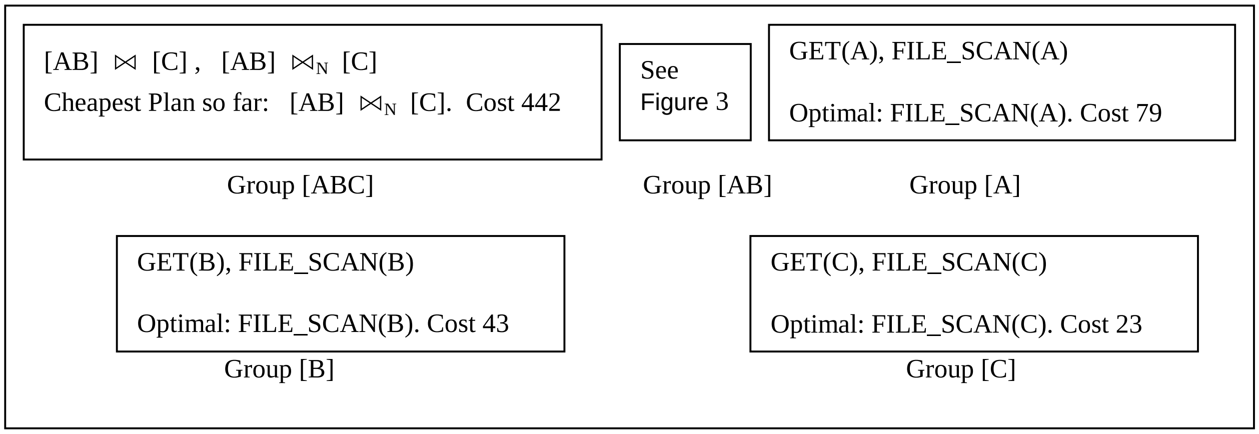
4.1 An example of the Cascades search strategy

We illustrate Cascades' search strategy with an example. Suppose the initial query is $(A \bowtie B) \bowtie C$, as in Figure 2 (i). We assume that the nontrivial join conditions are between A and B, and between B and C. (This condition is used only to infer, as described in Section 5.1, that $A \bowtie C$ is a Cartesian product.)

Cascades' search strategy will use CopyIn( ) to initialize the search space with the groups and multiexpressions illustrated in Figure 5.

After initialization, OptimizeGroup( ) will be called on the group [ABC], with no required property and an infinite upper bound. Suppose the first physical multiexpression enumerated at Figure 4, line (2), is [AB] $\bowtie_N$ [C]. The first recursive call from the [ABC] level, at Figure 4 line (6), will seek an optimal multiexpression (with no required properties) within the input group [AB]. This call will lead to one or more visits to the group [A], seeking optimal multiexpression(s) in A, and similarly for [B]. After these calls return to the [ABC] level, [AB] might look like Figure 3. The second recursive call from [ABC] for [AB] $\bowtie_N$ [C] ,at line (6), will seek an optimal multiexpression for the second input [C], again with no required properties.

After the second call returns, we can calculate a cost for the multiexpression [AB] $\bowtie_N$ [C]. At this point the resulting groups might look like Figure 6. Further along [AB] $\bowtie_M$ [C] will be considered, which will result in [AB] being revisited seeking different physical properties (namely a sort order). Logical transforms will produce [A] $\bowtie$ [BC] at some point, which entails the creation and intialization of the new group [BC]. (If we were working with more complex queries, such as ones with aggregation, there would be more groups than just one for each subset of relations.) Eventually group [ABC] will contain multiexpressions for all equivalent plans that can be generated by the optimizer's transformatons.



[AB] $\bowtie$ [C] , [AB] $\bowtie_N$ [C]
Cheapest Plan so far: [AB] $\bowtie_N$ [C]. Cost 442
Group [ABC]
See Figure 3
Group [AB]
GET(A), FILE_SCAN(A)
Optimal: FILE_SCAN(A). Cost 79
Group [A]
GET(B), FILE_SCAN(B)
Optimal: FILE_SCAN(B). Cost 43
Group [B]
GET(C), FILE_SCAN(C)
Optimal: FILE_SCAN(C). Cost 23
Group [C]

Figure 6: Cascades search space, after calculating the cost of [AB] $\bowtie_N$ [C]

4.2 Memoization vs. dynamic programming

A bottom-up optimizer visits each group exactly once, and during that visit it determines all the optimal plans in the group, for all physical properties anticipated to be useful. As our previous example illustrates, a top-down optimizer such as Cascades visits any group, call it G, zero or more times, once for each call to OptimizeGroup(G, …). During each call to OptimizeGroup( ), the optimizer considers several multiexpressions and chooses one (perhaps the NULL multiexpression, indicating that no acceptable plan is available) as optimal for the desired property. Any new optimal multiexpression is stored at Figure 4 line (10). This storing of optimal multiexpressions is the original definition of memoization [Mic68, RuN95]: a function that stores its returned values from different inputs, to use in future invocations of the function. Note that for memoization to work in this case, we need only retain the multiexpression representing the best plan for the given physical properties in a group. However, in Columbia we choose to retain other multiexpressions, as shown in Figure 3. There are two reasons for retaining non-optimal multiexpressions. One is that transforms might construct the same multiexpression in two different ways, and we want to know that a given multiexpression has already been considered. This is a minor issue since the unique rule sets of Pellenkoft et al [PGK97] minimize this duplication of expressions. The other reason is that a retained multiexpression might turn out to be the best multiexpression for a different set of physical properties in a later call to the group. We could eliminate all non-optimal multiexpressions in a group once we know the group will never be revisited, but that termination condition is hard to determine in practice.

5. Group pruning in Columbia

We say that a group G is *pruned* if, during optimization, the group is never optimized, i.e., if no multiexpressions in it are generated during the optimization process[2]. A pruned group will thus contain only one multiexpression, namely the multiexpression that was used to initialize it. Group pruning can be very effective: A group representing the join of i tables contains O (2^i) logical multiexpressions[3], each of which gives rise to one or more physical multiexpressions, all of which are avoided by group pruning.

In this section we will describe how Columbia increases the likelihood of achieving group pruning over Cascades, through the use of an improved search strategy

[2] Note that pruning is a passive activity – we don't actually remove the group at any point; rather, at the end of optimization, we find that the group has never been optimized.
[3] There are 2^i -2 such expressions because each nontrivial subset of the set of i tables corresponds to a different join, between the subset and its complement, excluding the entire set and the empty set.

for optimization. Note that some group pruning could happen in Cascades, as OptimizeGroup( ) is not called on the second input group of MExpr when the search of the first group fails to result in a multiexpression under the limit.

We emphasize that an optimizer that does group pruning still produces optimal plans, since it will only prune plans that cannot participate in an optimal plan. We call such a pruning technique *safe*, in contrast to *heuristic* techniques that can return a non-optimal plan.

5.1 Computing lower bounds aggressively to increase the frequency of group pruning

In Section 4 we have noted that the Cascades search strategy can lead to group pruning, when the loop of Figure 4 line (5) exits at line (7) and subsequent inputs are not optimized. In this subsection we will demonstrate a more aggressive approach: We will compute a lower bound for the multiexpression under consideration by looking ahead at inputs that are already optimized, and using logical properties for other inputs. This lower bound will force an earlier exit of the loop of Figure 4 line (5) and thus force more frequent group pruning. Figure 7 is a change to Figure 4 that implements this strategy.

```
(3a)  LB =  Cost of root operator of Mexpr +
(3b)        Cost of inputs that have winners for the required properties +
(3c)        Cost of copying-out other inputs

(5a)  For each input of MExpr without a winner for the required properties

(8a)  LB = LB + cost of InputWinner – copying-out cost for input
```

Figure 7: Improvement to the Cascades search strategy. Replacements for lines (3),(5) and (8) of Figure 4

We will first motivate and explain Figure 7, then continue with the example of Figure 6. The goal of the improvement described in Figure 7 is to avoid optimizing input groups, that is, avoid calling OptimizeGroup( ) at line (6), by adding together, in lines (3a-c), all input costs that can be deduced without optimizing any input groups. If the sum of these input costs exceeds UB, then OptimizeGroup( ) need not be called. The lower bound has three components. The first, line (3a) is identical to line (3). Next, line (3b) can be deduced by looking at the winner's circles of all input groups, without optimizing any of those groups. For input groups which have not been covered in line (3b), i.e. those which do not have winners, we can estimate a lower bound on the cost of any winner by first estimating the size of the output of the group (the output size is a logical property so it is the same for any multiexpression in the group). Once the output size estimate is known, the cost model yields an estimated cost for copying the output (whether it is pointers or records) to the next operator. This value is the *copying-out* cost in line (3c).

If the loop exits at line (4), we have avoided calling OptimizeGroup( ) on any input groups and they may never be optimized, i.e., they might be pruned. If the loop continues, control passes to line (5a), which then loops over all input groups whose winners were not found in line (3b). Line (8a) includes the new term "minus copying-out cost" because that cost was included at line (3c) previously and has now been replaced by the entire cost of the winner for this input, which includes the copying-out cost.

Next we will continue with the example of Section 4.1, which we left at Figure 6, where the cheapest plan so far has a cost of 442. Thus 442 is an upper bound on the cost of the optimal plan in group [ABC]. At this point, the multiexpression [AB] $\bowtie$ [C] will be transformed, at line (2) of Figure 4, to yield the merge-join [AB] $\bowtie_M$ [C]. Then OptimizeGroup( ) will be called on the input groups [AB] and [C], but this time with sort properties. We will skip these steps, assuming that the sort-merge join costs more than 442. Next, the logical transform rule of associativity will be applied to [AB] $\bowtie$ [C], resulting in the addition of both multiexpressions [A] $\bowtie$ [BC] and [B] $\bowtie$ [AC] to the group [ABC]. (Two multiexpressions are produced because the group [AB] contains both [A] $\bowtie$ [B] and [B] $\bowtie$ [A].) Eventually, [B] $\bowtie$ [AC] will be transformed to [B] $\bowtie_N$ [AC] at line (2). Assume the root operator, nested-loops join, costs 200 at line (3a); the winner for [B] costs 43 at line (3b). We have 442-243=199 remaining cost to work with. Since the group [AC] has not yet been optimized, there is no winner for the input [AC]. The join AC is a Cartesian product so its cardinality is huge. Therefore the cost of copying-out any plan in the group [AC] will be large, say 1000, greater than the remaining cost of 199. Thus the loop of line (2) will exit with failure at line (4) and the group [AC] will not have been optimized. If similar upper and lower bounds are available whenever [AC] appears in the optimization, then [AC] will never be optimized and none of the multiexpressions in [AC], except the one needed to populate it initially, will be constructed.

5.2 Comparison with AI search strategies

The search strategy of Figures 4 and 7 is similar to AI search strategies, especially A* [RuN95]. Both search strategies use estimated costs together with precise costs. However, there are several differences. A* works with partial solutions and partial costs, plus an estimate of the remaining cost; group pruning compares the cost of one complete solution (UB) to a lower bound of the cost of a set of solutions. The purpose of A* is to choose which subplans to expand next, whereas the purpose of group pruning is to avoid expanding a set of subplans.

5.3 Left-deep inputs simplify optimization

In this subsection we prove a lemma which is important in its own right and which we will use in the next subsection.

Pellenkoft et al. [PGK97] show that, for the join queries we are studying, four transforms, along with conditions for their application, can generate uniquely all logical multiexpressions in any group. The following lemma shows that when the input operator tree is a left-deep tree, just two of these four transforms will suffice. This lemma is useful because any operator tree containing only join and file retrieval operators is logically equivalent to a left-deep tree. Thus one can simplify such a query's optimization by beginning with a left-deep tree.

Lemma 1: Let Q be a left-deep operator tree, as in Figure 2(i). Apply the search strategy described in Section 4 with Q as input query. Use only the two logical transforms:

Left-to-Right Associativity and Commutativity.

and the conditions described by Pellenkoft et al. in [PGK97] for them, namely: During optimization of each group, order transforms as follows: apply associativity once to the first multiexpression in a group, then apply commutativity once to all the resulting logical multiexpressions in the group. Then

(1) Each group which has been optimized will contain all possible equivalent logical multiexpressions;
(2) If a group in MEMO contains more than one table, then the second input of its first multiexpression will be a single-table group;
(3) Only the associative transform will produce new groups.

Proof: Condition (3) is trivially satisfied since commutativity cannot produce new groups. Thus we prove only conditions (1) and (2).

Let the top group of the MEMO space, representing Q, be [A_1, …, A_k]. Since Q is a left-deep tree, the first multiexpression in [A_1, …, A_k] is (perhaps with renumbering) [A_1, …, A_{k-1}] $\bowtie$ [A_k].

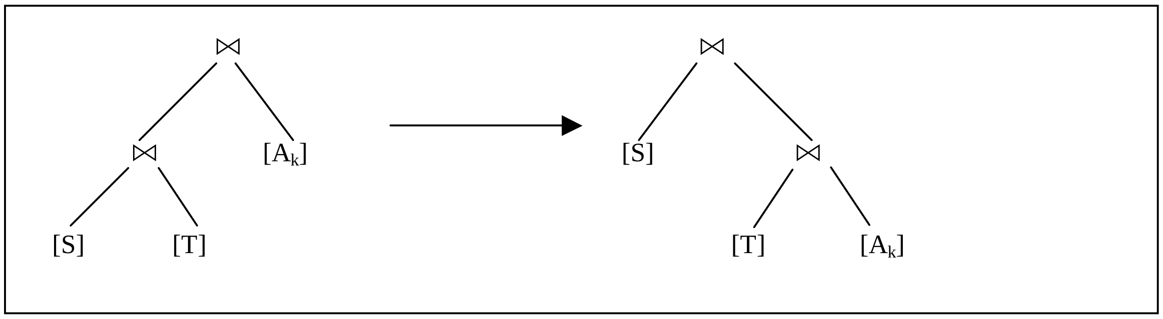


Figure 8: Associativity applied to a left-deep multiexpression

The proof proceeds by induction on k. The induction hypothesis is that conditions (1) and (2) hold for groups containing tables from the set {A_1, …, A_j}. The basis step, j=1, is trivially satisfied by single table groups. We assume the inductive hypothesis for j = k-1 and prove it for j = k.

We first prove condition (1) for the top group [A_1, …, A_k]. There are $2^k - 2$ logically equivalent multiexpressions in any group with k tables (see footnote 3) . Now count the multiexpressions generated by the two transforms when they are applied to the first multiexpression in the group, namely [A_1, …, A_{k-1}] $\bowtie$ [A_k]: $2^{k-1} - 2$ multiexpressions are generated by associativity, one for each nontrivial subset of {A_1, …, A_{k-1}}. Commutativity adds a mirror image to each of these and to the original multiexpression [A_1, …, A_{k-1}] $\bowtie$

$[A_k]$, for a total of $2(2^{k-1} - 2) + 2 = 2^k - 2$ distinct multiexpressions, proving condition (1). Since condition (2) is clear for the top group, we have proved the inductive step for the top group.

It remains to prove conditions (1) and (2) for any group generated from the further optimization of the top group, i.e., any group containing A_k. Since the commutative transform does not generate new groups, each new group is generated as a result of the associativity transform applied to the first multiexpression $[A_1, \ldots, A_{k-1}] \bowtie [A_k]$ in the top group. Any application of associativity will be of the form pictured in Figure 8, where S and T form a nontrivial partition of $\{A_1, \ldots, A_{k-1}\}$. The right multiexpression in Figure 8 has two input groups, namely [S] and $[TA_k]$. [S] is already in the search space by induction, but $[TA_k]$ is new. Its first multiexpression is given by the right input of the new multiexpression above, namely $[T] \bowtie [A_k]$, which satisfies condition (2). Since a counting argument similar to the one used above can verify condition (1) for this case, Lemma 1 is proved.

Lemma 1 says that we can arrange optimization so that the first multiexpression of each group has one table as the right input, but the left input may be a Cartesian product and therefore very expensive, giving us a high upper bound when we compute the cost of a physical multiexpression based on this logical multiexpression. The next subsection deals with this problem.

5.4 Obtaining cheap plans quickly

Group pruning will be most effective when cheap plans are obtained early in the optimization process, since the UB at line (4) of Figure 4 represents the cost of the cheapest plan seen so far. For example, if the original operator tree in the example of Section 4.1 had been (AC)B, i.e., included a Cartesian product, then the group [AC] would have been optimized and not pruned. We want to avoid such situations.

In many, but not all, situations, Cartesian product joins are the most expensive joins considered during optimization. There are exceptions to this heuristic[ONL90] – today we would call those exceptions star schemas [MMS98].

As usual, a connected query is one whose join graph is connected. We define a group to be *connected* if its corresponding query is connected. If a group is not connected, then any plan derived from the group will include at least one Cartesian product. Thus, for a non-connected group, there is typically little hope of obtaining a cheap plan quickly – none of the plans in the group will typically be cheap.

Therefore the best one can hope for during query optimization is that when optimizing a connected group, the first multiexpression in that group will contain a plan that includes no Cartesian product. The next Theorem shows that for a connected acyclic query this hope can always be achieved.

Lemma 2: Let Q be a connected acyclic query. Then Q is logically equivalent to a left-deep operator tree R such that the left input of any subtree of R is connected.

Proof: Construct R by removing from the join graph of Q one non-cut node at a time and adding the removed table to R, along with whatever join conditions are inherited from Q.

Theorem: Let Q be a connected acyclic query. Apply the search strategy of Section 4, as described in Lemma 1, to the left-deep tree given by Lemma 2. Then every connected group in the resulting MEMO will begin with a multiexpression containing a plan with no Cartesian product.

Proof: By induction, we can assume the theorem true for any connected acyclic query with k-1 tables or fewer. Assume Q has k tables. By Lemma 1, only associativity produces new groups, so we must show only that in Figure 8, if the new group $[TA_k]$ has a connected join graph then its first multiexpression $[T] \bowtie [A_k]$ will have a connected plan. First we will prove that [T] is connected.

We observe that there is only one edge between A_k and any table in S or T. If there were two such edges, the connectedness of [ST] would yield a cycle in $[STA_k]$. We let B denote the other table in that edge. Since $[TA_k]$ is connected, B must be in T. Since there is only one edge between A_k and T, and $[TA_k]$ is connected, T must be connected.

Since [T] is connected, it contains a connected plan. Since $[TA_k]$ is connected and A_k is a single table, the join of that plan with A_k is also connected. Thus we have produced the desired plan.

We note that this theorem is not true for connected cyclic queries. For example, let Q be the query with tables A, B, C, D and edges between the pairs AB, BC, CD, and DA. Suppose the first multiexpression in the top group is $[ABC] \bowtie [D]$. Using the subset {A, C} to do associativity yields the multiexpression $[AC] \bowtie [D]$, which generates the connected group [ACD]. However, $[AC] \bowtie [D]$ cannot have a connected plan since the input [AC] is a Cartesian product.

The theorem gives us a prescription for optimizing any connected acyclic query and guarantees that all connected groups will begin with a multiexpression that is in some sense cheap. Hopefully the other, nonconnected, groups will be pruned by the optimization. We will test this hypothesis in the next section.

This theorem gives us a second benefit, namely that fewer transforms are needed to optimize the query, so optimization should be more efficient.

6. Performance analyses

It is difficult to compare the performance of different optimizers. Measures such as elapsed time or memory usage are not comparable unless both optimizers are implemented in the same environment. OPT++ [KaD96] makes a significant contribution in this direction, implementing several approaches within a single framework, but OPT++ reports only the Volcano, and not the Cascades or Columbia, search strategies. As we have noted in Section 2, the Volcano search strategy does not lend itself to pruning. Furthermore, it is not clear that the structure of OPT++ is appropriate for the multiexpressions used in top-down optimizers.

We use the measure of number of multiexpressions generated in order to compare bottom-up optimizers, such as Starburst, with Columbia. For example, in Figure 3, six multiexpressions have been generated in the group [AB]. This measure is clearly independent of platform. While top-down optimizers need to check each newly generated multiexpression to ensure that it is not already in the MEMO, our experiments show that this is not a major expense. Memory usage is another expense which is more significant for top-down than bottom-up optimizers, but we leave that analysis to future work.

Our goal in this section is to determine the effectiveness of group pruning in minimizing the number of multiexpressions generated during optimization. First we will compare Columbia with group pruning to Starburst, which considers all logically equivalent multiexpressions for each group. As mentioned in Section 5, Columbia with group pruning produces optimal plans, so its output will be the same as Starburst's. Then we compare Starburst using heuristics to Columbia.

The most significant factors affecting the number of multiexpressions generated during a query's optimization are the query's shape and the number of tables involved [OnL90]. The extremes are given by chain (also called linear) and star queries. Chain query optimization gives rise to the greatest number of Cartesian products and star queries the least, for a given number n of quantifiers (tables represented by variables) and n-1 predicates. Cartesian products affect complexity because they give rise to pruning possibilities as we have seen previously, and they are the core of Starburst's most important heuristic.

Our experiments use the nested-loops and sort-merge operators described above. They also use the methodology described by the theorem, namely transforming each input query into a left-connected deep tree before optimization begins. All Columbia data in our graphs were derived from executions of the Columbia optimizer. Starburst values were derived from the formulas of Ono and Lohman [OnL90].

Our choice of queries and catalogs is influenced by the work of Vance and Maier[VaM96]. Each of the queries used in our experiments uses tables denoted T_1, …, T_n . The geometric mean of table cardinalities $|T_i|$ will be fixed at 4096. We have used other values with similar results. In a given catalog, the ratios $|T_i|/|T_{i+1}|$ are equal for all i and the $\log_2$ of the ratio $|T_1|/|T_n|$ is called the *LOGRATIO*. Thus if a catalog has 7 tables and a LOGRATIO of 6 then the tables' cardinalities are 2^{15}, 2^{14}, …, 2^{10}, 2^9. The following results are sensitive to join selectivities. Increasing join selectivity has the same effect as decreasing the LOGRATIO.

The graphs involving Starburst are independent of table cardinalities because whether a plan includes a Cartesian product or is a bushy tree does not depend on cardinalities. However, the graphs representing Columbia do depend on table cardinalities since those cardinalities contribute to the bounds used in pruning.

For chain queries, we assume each join of T_i with T_{i+1} is a foreign key join, for which the join selectivity is derived from $|T_i \bowtie T_{i+1}| = |T_i|$. For star queries, the foreign key joins are of T_i with T_1 for i = 2, …, n.

6.1 Effectiveness of group pruning

Figure 9 shows that Columbia's group pruning can be quite powerful. It also demonstrates the effect of query shape and of table cardinalities. In this Figure, "Default" is the number of multiexpressions generated by Columbia with no group pruning or by Starburst using no heuristics. The other quantities in Figure 9 represent group pruning in Columbia of Star and Chain queries using tables whose cardinalities vary by factors ranging from 1 to 2^{24}. The savings in multiexpressions ranges from approximately 60% for star queries to 98% for chain queries. We note that all these savings are accomplished while yielding optimal solutions. The effect of table cardinalities is minimal – the savings varies by only a few percent in each case. In future examples we will use LOGRATIO = 12.

6.2 Group pruning compared with heuristics

Figure 10 compares the effectiveness of Columbia's group pruning with Starburst's primary heuristics, using the same 13 table example queries. The Default column is as in Figure 9. The leftmost columns represent and chain queries, is very competitive. However, the heuristic of considering only left deep plans is, for chain queries, inferior to group pruning. Of course either of the heuristics mentioned can be applied in top-down optimizers such as Columbia.

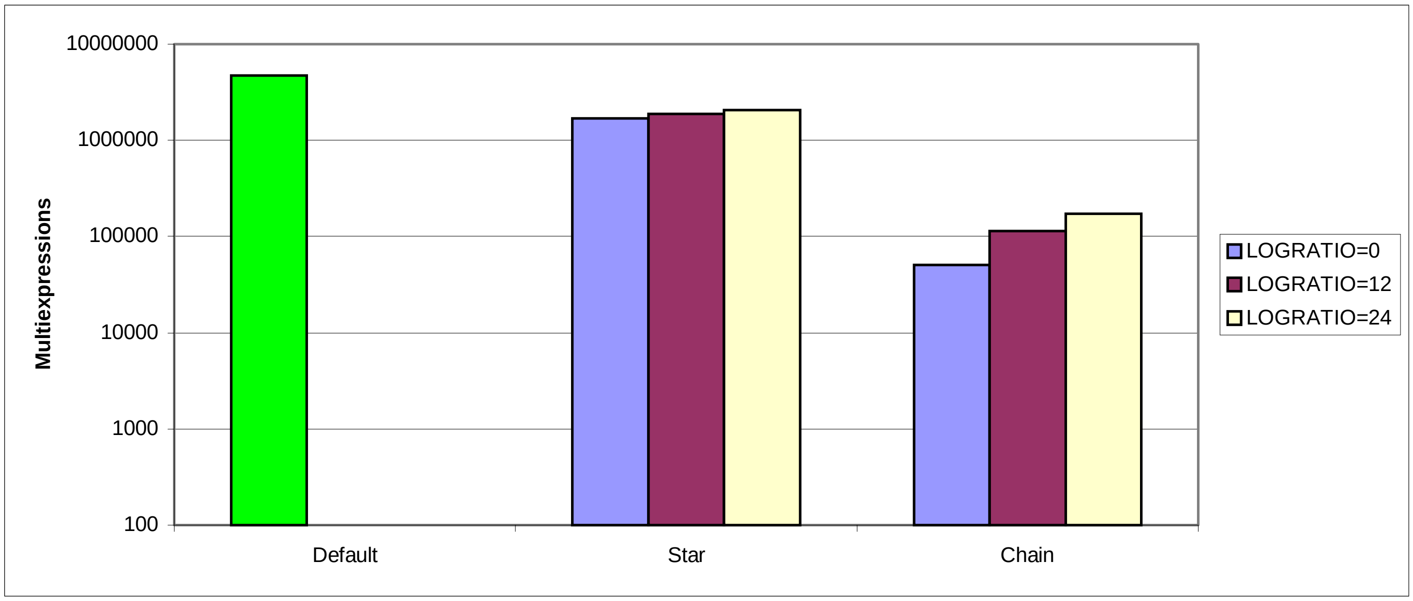


Figure 9: Effectiveness of group pruning. Example with 13 tables.

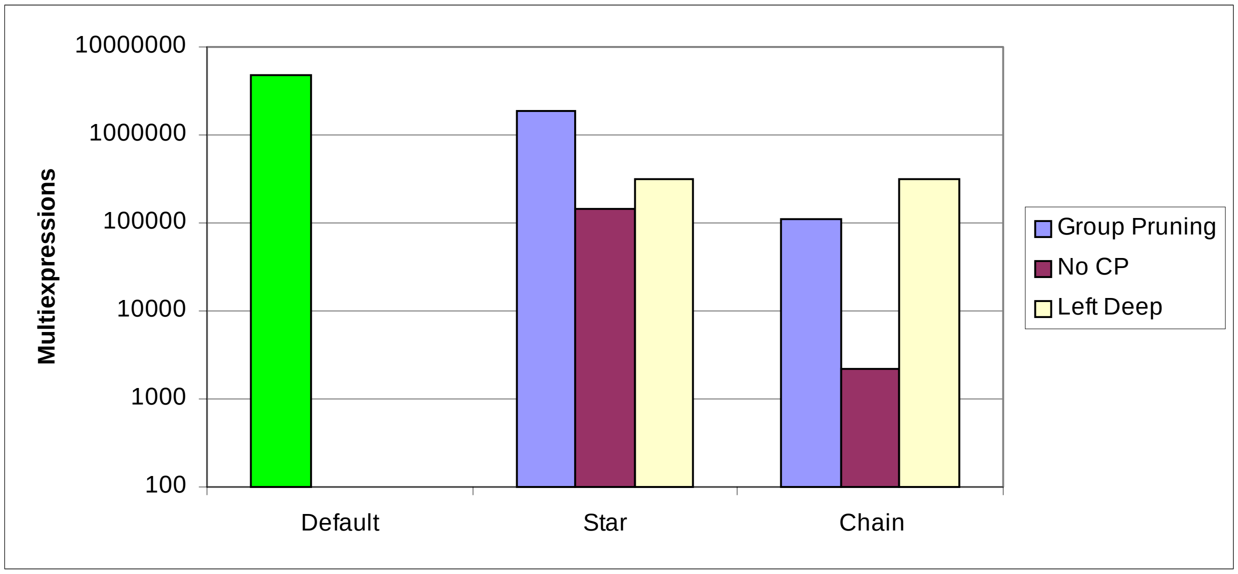


Figure 10: Comparison of group pruning and heuristics

Columbia's group pruning strategy, yielding optimal results. Starburst's "postpone Cartesian products" heuristic, which for connected queries amounts to no Cartesian products, shown by the middle column for star

6.3 Significance of number of tables

Figure 11 shows the variance in multiexpression complexity as the number of tables varies. The relative impact of the default case versus group pruning for star and chain queries, and the effect of excluding Cartesian products or of considering only left deep trees, is the same regardless of how many tables are involved. Note that the "left deep" column is identical for chain and star queries since a query's predicates do not affect whether or not a plan for it is left deep.

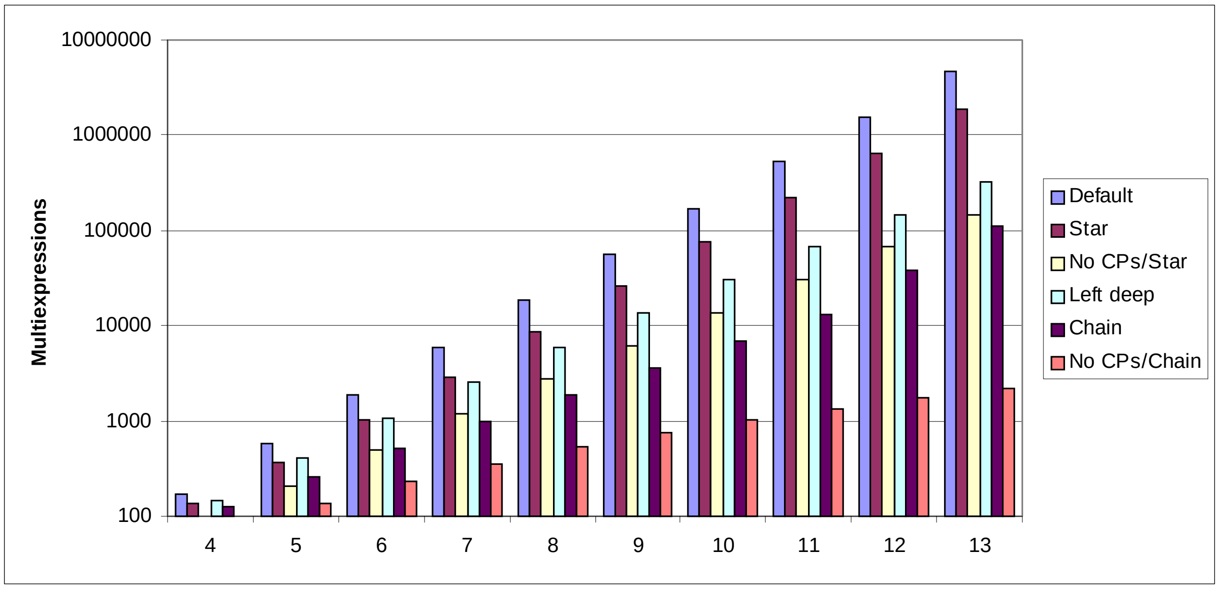


Figure 11: Effect of varying number of tables

7. Summary and future work

We have explained that top-down and bottom-up optimizers use the principle of optimality to populate two data structures (groups and multiexpressions) and pointed out that a key difference is in the order of enumerating the groups. We have described the search strategy of the Cascades query optimizer and the improvements on that search strategy that we have implemented in a query optimizer called Columbia. Those improvements use upper and lower bounds to implement group pruning, a method for avoiding the generation and testing of candidate multiexpressions that cannot participate in an optimal solution. Such bounds are not available to bottom-up optimizers such as Starburst. We have proven that any connected acyclic query can be optimized in such a way that cheap upper bounds are likely to be obtained early in the optimization process.

We have described the performance characteristics of Columbia compared to bottom-up optimizers such as Starburst. We considered chain and star queries with varying numbers of tables and varying cardinalities. For our benchmarks, when both Columbia and Starburst are required to produce optimal results, Columbia generates between 60 and 98% fewer multiexpressions than Starburst. Given that Starburst uses heuristics and risks generating nonoptimal plans, we show that there are cases in which Columbia can produce optimal results with fewer multiexpressions than Starburst.

Our major conclusion is that, judging by the number of multiexpressions generated, top-down optimizers have the potential to outperform bottom-up optimizers.

Our future work will focus on two areas. The first is memory usage. Bottom-up optimizers have the advantage of being able to discard non-winner multiexpressions at each level when that level is completed. Top-down optimizers normally retain all multiexpressions, both because OptimizeGroup( ) may be called more than once on the same group and because transforms such as associativity may use a group multiple times. This retention of multiexpressions leads to poor memory usage. We plan to compare alternative solutions to this problem, including the use of heuristics.

The second area we plan to pursue is more foundational. Because of the complexity of top-down optimization, several interesting questions remain to be answered about it. For example, can one prove rigorously that it yields the same plans as bottom-up optimization? Optimization does not always terminate (e.g., if the machine being modeled has an infinite number of processors and plans with infinitesimally small costs), but are there conditions on rule sets that guarantee termination? Is every rule set equivalent to a set of rules of some simple form?

Acknowledgements

Goetz Graefe provided us with a copy of Cascades and helped us to understand it. Bill McKenna, Cesar Galindo-Legaria and Pedro Celis kindly shared with us many facets of their work on commercial implementations of top-down optimizers. Leonidas Fegaras helped with several stimulating discussions during the early part of this work.

References

[Bel75] R. E. Bellman, *Dynamic Programming*, Princeton University Press, Princeton, New Jersey, 1975.

[Bil97] Keith Billings, *A TPC-D Model for Database Query Optimization in Cascades*, M.S. Thesis, Portland State University, Spring 1997.

[Cel96] P. Celis, The Query Optimizer in Tandem's ServerWare SQL Product, *Proceedings of VLDB 1996*, Pg. 592.

[CSL90] M. Carey, E. Shekita, G. Lapis, B. Lindsay and J. McPherson, An Incremental Join Attachment for Starburst, *Proceedings of VLDB 1990*, Pg. 662-673.

[GLS93] P. Gassner, G. M. Lohman and K. B. Schiefer, Query Optimization in IBM's DB2 Family of DBMSs, *IEEE Data Engineering Bulletin*, 16(4), December 1993, Pg. 4-18.

[Gra95] G. Graefe, The Cascades Framework for Query Optimization, *Bulletin of the IEEE Technical Committee on Data Engineering*, 18(3), September 1995, Pg. 19-29.

[Gra96] G. Graefe, The Microsoft Relational Engine, *Proc. Data Engineering Conf. 1996*, Pg. 160-161.

[GrD87] G. Graefe and D. J. DeWitt, The EXODUS Optimizer Generator, *Proc. SIGMOD 1987*, Pg. 160-172.

[GrM93] G. Graefe and W. J. McKenna, The Volcano Optimizer Generator: Extensibility and Efficient Search, *Proc. Data Engineering Conf. 1993*, Pg. 209-218.

[HKW97] L. Haas, D. Kossman, E. Wimmers, J. Yang, Optimizing Queries Across Diverse Data Sources, *Proc. VLDB 1997*, Pg. 276-285.

[HCL90] L. Haas, W. Chang, G. Lohman et al., Starburst Mid-Flight: as the Dust Clears, TKDE, 2(1), Pg. 143-160, March 1990.

[KaD96] N. Kabra, D. DeWitt, OPT++ : an object-oriented implementation for extensible database query optimization, *VLDB Journal: Very Large Data Bases*, 8(1), pp. 55-78, May 1999

[Loh88] G. Lohman, Grammar-like Functional Rules for Representing Query Optimization Alternatives, *Proc. SIGMOD 1988*, Pg. 18-27.

[MBH96] W. McKenna, L. Burger, C. Hoang and M. Truong, EROC: A Toolkit for Building NEATO Query Optimizers, *Proc. VLDB 1996*, Pg. 111-121.

[McK93] W. McKenna, *Efficient Search in Extensible Database Query Optimization: The Volcano Optimizer Generator.* PhD thesis, University of Colorado, Boulder, 1993.

[MDZ93] G. Mitchell, U. Dayal and S. B. Zdonik, Control of an Extensible Query Optimizer: A Planning-Based Approach, *Proc. VLDB 1993*, Pg. 517-528.

[Mic68] D. Michie, 'Memo' Functions and Machine Learning, *Nature*, No. 218, Pg. 19-22, April 1968.

[MMS98] D. Maier, M. Meredith and L. Shapiro, Selected Research Issues in Decision Support Databases, *Journal of Intelligent Information Systems*, 11, 169-191 (1998).

[ONK95] F. Ozcan, S. Nural, P. Koksal, M. Altinel, A. Dogac, A Region Based Query Optimizer through Cascades Optimizer Framework, *Bulletin of the Technical Committee on Data Engineering*, Vol 18 No. 3, September 1995, Pg 30-40.

[OnL90] K. Ono and G. M. Lohman, Measuring the Complexity of Join Enumeration in Query Optimization, *Proc. VLDB 1990*, Pg. 314-325.

[PGK97] A. Pellenkoft, C. Galindo-Legaria, M. Kersten, The Complexity of Transformation-Based Join Enumeration, *Proc. VLDB 1997*, Pg. 306-315.

[Ram00] R. Ramakrishnan, J. Gehrke, *Database Management Systems, Second Edition*, McGraw Hill, 2000

[RuN95] S. Russel, P. Norvig, *Artificial Intelligence: A Modern Approach*, Prentice Hall Series in Artificial Intelligence, 1995.

[SAC+79] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie and T. Price, Access Path Selection in a Relational Database Management System, *Proc. SIGMOD 1979*, Pg. 22-34.

[VaM96] B. Vance and D. Maier, Rapid Bushy Join-order Optimization with Cartesian Products, *Proc. SIGMOD 1996*, Pg. 35-46.