Counting, Enumerating, and Sampling of Execution Plans in a Cost-Based Query Optimizer

Florian Waas[1,2]
flw@cwi.nl

César Galindo-Legaria[2]
cesarg@microsoft.com

[1]CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

[2]Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
U.S.A.

Abstract

Testing an SQL database system by running large sets of deterministic or stochastic SQL statements is common practice in commercial database development. However, code defects often remain undetected as the query optimizer's choice of an execution plan is not only depending on the query but strongly influenced by a large number of parameters describing the database and the hardware environment. Modifying these parameters in order to steer the optimizer to select other plans is difficult since this means anticipating often complex search strategies implemented in the optimizer.

In this paper we devise algorithms for counting, exhaustive generation, and uniform sampling of plans from the complete search space. Our techniques allow extensive validation of both generation of alternatives, and execution algorithms with plans other than the optimized one—if two candidate plans fail to produce the same results, then either the optimizer considered an invalid plan, or the execution code is faulty. When the space of alternatives becomes too large for exhaustive testing, which can occur even with a handful of joins, uniform random sampling provides a mechanism for unbiased testing.

The technique is implemented in Microsoft's SQL Server, where it is an integral part of the validation and testing process.

1 Introduction

Cost-based query optimizers typically consider a large number of candidate execution plans, and select one for execution. The choice of an execution plan is the result of various, interacting factors, such as database and system state, current table statistics, calibration of costing formulas, algorithms to generate alternatives of interest, and heuristics to cope with the combinatorial explosion of the search space. Normally, experimental validation and testing of the query processor is limited to considering the one plan that was chosen by the optimizer for execution. This is a severe limitation, as this plan is only a minuscule fraction of the space of alternatives. In fact, during regular development and maintenance of a query processor, it has been our experience that some code defects can remain undetected for a long time, until the right combination of factors steer the optimizer to chose a plan that exposes the problem.

Rather than waiting for these problem scenarios to occur, or trying to manually influence optimizer choices towards "potentially problematic" cases, we generate alternative execution plans by enumeration and sampling, from the space of alternatives *considered* by the optimizer. The plans are generated independently from optimizer decisions and provide a large set of test cases for both the optimizer—are the alternatives considered really valid execution plans?—and the execution engine—do different but semantically equivalent plans produce the same output?

This approach to testing is similar to that taken by Slutz [11], in which a large number of random SQL statements are submitted to the database. Random statements can be generated quickly, and extensive coverage of the code can be achieved in a short time. Multiple execution plans for a given query test smaller system components; it shows the result of arbitrary combinations of optimization rules, and exercises execution algorithms in configurations that are less common. Starting from a query with that has specific properties, e.g. joins and outerjoins, or joins and aggregations, an "area" of the optimizer and execution code is targeted and exercised in a variety of combinations.

We develop a general approach based on *ranking* elements of a space, which allows enumeration and sampling of plans. The basic idea is to establish a one-to-one mapping between integers $0, \ldots, N-1$ and the N elements of a space. *Ranking* an element, e.g. an execution plan, means finding its number; *unranking*

a number means constructing the corresponding plan. Once an unranking mechanism is available, uniform sampling of elements in the space reduces to random generation of numbers in the range $0, \dots, N-1$.

None of the known ranking and unranking techniques for tree structures apply to the current problem [10, 2], as the space of alternatives considered by industrial query optimizers is not restricted to an abstract combinatorial problem, such as join reordering. Multiple execution algorithms, index utilization, reordering of grouping operators, special-purpose physical operators, and heuristics to control the time spent on searching, all make up for an *actual* space that is hard to describe succinctly using abstract, regular structures.

The technique we devised achieves an unranking mechanism based on the compact representation of multiple alternatives, in the style of the MEMO structure of Volcano [7, 5], used in Microsoft's SQL Server and Tandem's NonStop SQL. Initially introduced in a transformation-based system, this data structure simply captures the multiple choices available to a cost-based optimizer, not necessarily constructed using transformation rules—a bottom-up enumeration approach implicitly uses a similar data structure.

After performing the regular optimization of a query, we modify this data structure to facilitate the counting of all possible plans and the subsequent generation of a particular plan. The overhead incurred by this kind of post processing is negligible for both, counting and extracting a certain plan. Furthermore, we extended the SQL syntax to allow the specification of a plan, i.e., the specification of a the plan's number, within the standard interfaces.

Its marginal overhead together with a simple and easy to use interface have made this technique a valuable tool and integral part of the testing process in the SQL Server development.

In addition to its immediate use for testing, we also used this mechanism to perform some experiments in a largely unexplored field of query processing: The cost distribution of query plans. Cost distributions are of interest, because they can be taken as obvious indicators of the stochastic difficulty of a problem, by simply considering the ratio of high quality to low quality plans [6].

The remainder of this paper is organized as follows. In Section 2, we briefly outline the optimizer framework and the MEMO structure. The counting and unranking schemes are introduced in Section 3. In Section 4, we report on the experience with using the tools in the ongoing development of Microsoft's SQL Server. We present initial results on cost distributions computed for TPC-H queries in Section 5. Section 6 concludes the paper.

2 Preliminaries

In this section we review the concept of a compact representation of the plan space in form of the MEMO structure. This concept was developed by Graefe and DeWitt in the context of transformation-based query optimization [4, 5, 1]. Independent of this development, a similar structure has been developed for bottom-up enumeration of join trees in Starburst [8]. Our technique is based on the MEMO but could be transferred easily to the Starburst enumerator.

We will briefly introduce the essential aspects of the MEMO and refer the interested reader to [3] for further reading.

A *query plan* determines the execution order of a set of relational algebra operators which implements a given, declarative query. Query plans are n-ary trees whose nodes correspond to algebra operators and are therefore referred to as operators too. Due to the tree structure, every operator represents a sub-goal of the plan, that is, the partial query evaluation done by the sub-tree rooted in it.

A cost function computes a *cost value* for a query plan which is for instance the time needed to execute the plan. The goal of the optimization is to generate the query plan with the least cost value, i.e. to solve the associated combinatorial optimization problem. Cost based query optimizers like the ones used in Microsoft's SQL Server, Tandem's NonStopSQL or IBM's DB2 generate partial query plans, cost them and—if a partial plan is a candidate to be part of the optimal plan—store them in a lookup table. The generation of sub-plans and their alternatives is guided by strategies and can be implemented for instance in a transformation-based framework or with dynamic programming.

In the following we outline the optimization process as implemented in SQL Server, which is similar to that of Volcano. We distinguish two kinds of operators: (1) *logical operators* that map to relational algebraic operators, e.g. join operator, and (2) *physical operators* that represent a particular implementation of a logical operator, e.g. hash join. Only physical operators may be used in the final query plan. Following Volcano, we call the aforementioned lookup table *MEMO structure*. It is a data structure that manages a system of *groups*, which represent different sub-goals of a query plan, i.e. every group corresponds to the root of a sub-plan.

We start out with an initial query plan that consists of logical operators only. This plan is a direct translation of a declarative query given in SQL. We map the initial query plan to the groups of the MEMO so that every operator is assigned to one group. The group that contains the initial plan's root operator is referred to as *root group*. We substitute the original references to an operator's children by references to the respective

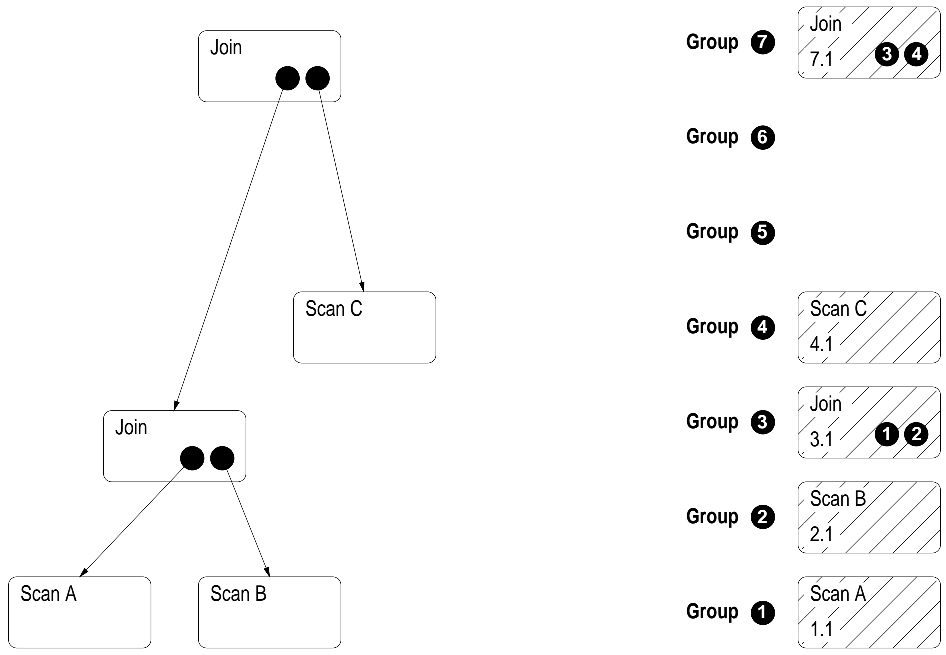

Figure 1: Copying the initial plan into the MEMO structure.

groups. Figure 1 shows an initial tree and its equivalent after copying it into the MEMO structure. Operators in the MEMO are depicted as rounded boxes with the references to the children's groups in the lower right corner and a unique identifier in the lower left corner. The references to the children's groups are ordered, that is, the left number represents the first child's groups, and if available, the right is the second child's. For simplicity, we use only unary and binary operators in the examples, however, the methods we present are not limited to any given degree. To avoid renumbering of groups at a later point in time we put the root operator immediately into group 7 in this example. Thus, group 7 becomes the root group. Notice, in the actual implementation, groups are not ordered but only referred to by their numbers. However, putting it directly into group 7 and maintaining an order makes this example more intuitive and easier to understand.

Once the initial plan is copied into the MEMO, we derive alternatives by applying transformations to the logical operators. A transformation rule can generate:

1. a logical operator in the same group, e.g. join(A,B) → join(B,A);
2. a physical operator in the same group, e.g. join → hash join;
3. a set of logical operators that form a connected sub-plan; the root goes to the original group, other operators may go to any group, including the creation of new groups as necessary, e.g. join(A,join(B,C) → join(join(A,B),C).

In Figure 2, a partially expanded MEMO structure is depicted. The physical operators are shaded and an example plan is shown with darkened operators.

We do not apply rules to transform physical operators since everything that could be derived from a physical operator can also be derived from the logical one. A technicality that needs special attention is the fact that operators of the same group—i.e. with the same logical properties—may differ in physical properties. For instance, one operator may deliver a sort order whereas another operator of the same group does not, or it may deliver a sort order on a different attribute. In case the parent operator requires a sort order on a certain attribute, not all operators may be chosen as potential children.

The MEMO framework includes routines that analyze the results of a rule application and assign it to the groups, detect and eliminate duplicates, and create new groups. Furthermore, it also provides costing techniques that estimate and assign costs to each operator in the MEMO, that is, the costs of the sub-plan rooted in each operator. For every group we keep track of the best physical operator for a each set of physical properties. When costing a new operator we compute the costs using the children's best implementations. Moreover, the MEMO contains scheduler primitives that implement different strategies as to when to apply what rule. A

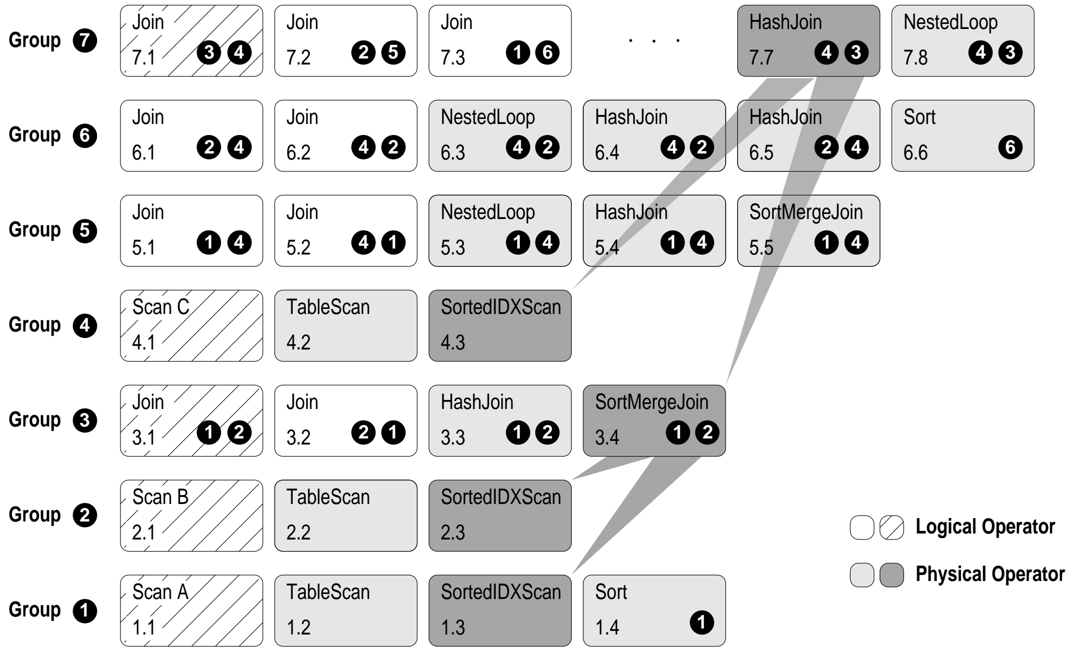

Figure 2: MEMO structure representing alternative solutions.

cost based pruning heuristic helps avoid expansion of very costly alternatives that, given the current state of optimization, cannot be a sub-plan of the optimal plan, and therefore need not to be explored.

The optimal query plan is the one rooted in the most cost effective operator in the root group. To extract this plan, we follow the references to the children's groups and select the most cost effective operator of each group, observing compatibility of physical properties. This step is repeated until we reach the terminal operators. Note, this plan was already implicitly used for costing the best operator in the root group.

Though we described the use of transformations to generate alternative sub-plans form an initial plan, also other techniques like bottom-up enumeration [8] could be used to populate a structure functionally equivalent to the MEMO. The methods developed in the following are independent of the algorithms to construct the MEMO structure, and simply rely on this structure as a compact representation of the candidate plans considered by the optimizer. Some optimizers by default discard suboptimal expressions. For our technique to be most effective, it is useful to have the optimizer keep each alternative generated, so they can be freely used, regardless of their cost.

3 Counting and Unranking Query Plans

Once all alternatives are generated, the MEMO structure contains all operators but does not keep track of how many combinations of operators there are, and only the optimal plan is completely assembled. That is, at the end of the optimization, the MEMO contains a concise and compact encoding of the complete search space that was considered during the optimization.

To illustrate the counting framework, let us assume a final state of the MEMO —after generation of alternatives is complete— as given in Figure 3.

3.1 Preparatory Steps

In order to facilitate later operations we extract all physical operators and materialize the links between operators and their possible children. In Figure 3, the materialized links for all children of the previous example's root (operator 7.7) are shown. The resulting structure describes all possible execution plans that can be rooted in this operator.

Due to the differences in physical properties some operators of a group may qualify as potential children while others do not. For instance operator 3.3 in Figure 3, can have any operator from group 1 and 2 as left and right child, respectively. Operator 3.4 however can use only the darkened operators 2.3 and 1.3 or 1.4.

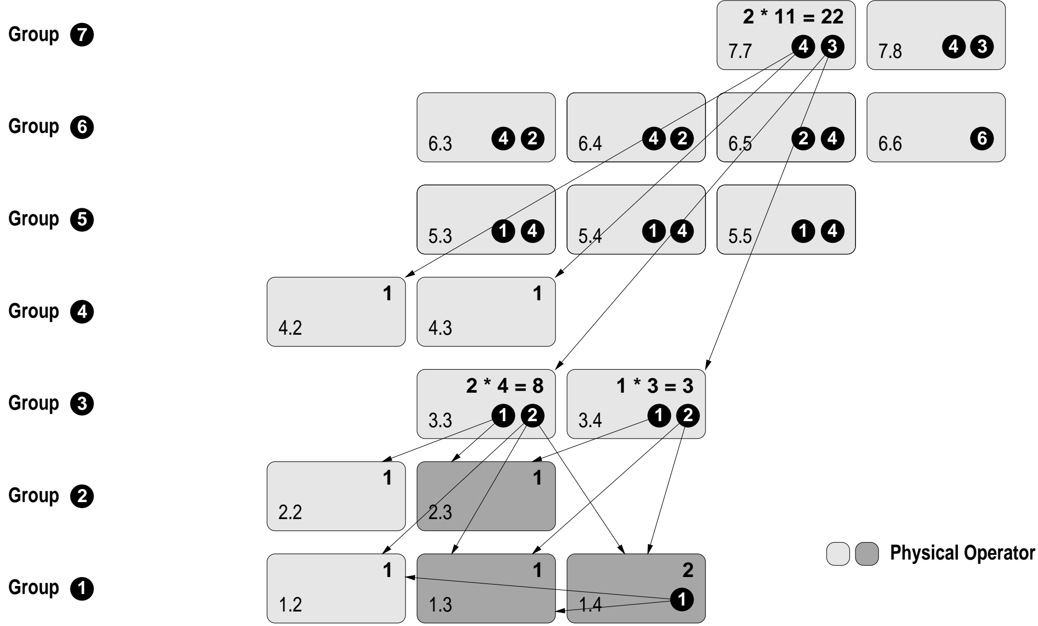

Figure 3: MEMO Structure with materialized links between operators and children, for possible plans rooted in operator 7.7.

3.2 Counting Query Plans

We compute the total number of possible plans bottom-up by computing the individual numbers of possible plans that can be extracted from each operator. We denote the number of children of operator v by $|v|$, and the j-th alternative for the i-th child of v by $w_{i,j}^{(v)}$. For example, in Figure 3, take $v = 7.7$, then $w_{1,1}^{(v)} = 4.2$, and $w_{2,2}^{(v)} = 3.4$.

To compute the number of plans $N(v)$ rooted in an operator v, we first determine the number of possible alternatives for each child i as

$$b_v(i) = \sum_j N(w_{i,j}^{(v)}).$$

Operator v will take any of the available alternatives for each child, independently, so the number of combined choices is given by a product. The numbers of plans we can generate using only the first k children is

$$B_v(k) = \prod_{i=1}^{k} b_v(i).$$

Hence, the number of plans rooted in v is

$$N(v) = \begin{cases} 1, & \text{if } |v| = 0 \\ B_v(|v|), & \text{otherwise} \end{cases}$$

In Figure 3, this process is illustrated for operator 7.7. The upper right corner of operators has the computation of the number of alternatives that can be extracted using it as a root.

The total number of plans is the sum of possible plans rooted in any of the root group's operators:

$$N = \sum_i N(v_i), \qquad v_i \in G_{root}$$

where G_{root} denotes the root group.

Computing the counts for operators takes linear time on the size of the MEMO, as each operator has to be visited exactly once.[1] In practice, the time needed for counting never exceeded 1 second even for large queries.

3.3 Unranking Plans

Before we describe the unranking mechanism in detail, it might be helpful to give a short outline of the idea:

Starting with the root group and the rank r, we choose an operator of the group to be the root of the tree. We then compute a *local rank* for this operator. This local rank for an operator v is in the interval $0, \ldots, N(v)$.

[1]For the number of logical operators for the problem of join reordering, see [8, 9]. There are a few physical operators for each logical joins, implementing different alternatives of hash join, merge join, and index lookups, so the number of physical joins is usually a small multiple of the count of logical joins.

Now, assume operator v has children alternatives

$$\{w^{(v)}_{1,1}, \dots, w^{(v)}_{1,j_1}\}, \dots, \{w^{(v)}_{n,1}, \dots, w^{(v)}_{n,j_n}\},$$

with $n = |v|$. n *sub-ranks* are computed, and used in each child choice to recursively unrank a sub-plans. The resulting tree is assembled from unranked sub-plans, using v as the root.

Detailed steps are described next.

1. Given a pair (r, G) consisting of a rank and a group we determine which operator of this group becomes the root of the sub-plan.

   The first physical operator in the group covers the plans $0, 1, \dots, N(v_1)-1$, the second $N(v_1), N(v_1)+1, \dots, N(v_1)+N(v_2)-1$ and so on. Thus, the sought operator has index

   $$k = \max\{i | \sum_i N(v_i) \leq r\}.$$

   v_k becomes the root of the (sub-)plan. The local rank r_l of v_k is

   $$r_l = r - \sum_{i=1}^{k-1} N(v_i)$$

   The local rank is necessary to determine the sub-ranks for the children in the next step.

2. Using the concepts introduced in the previous section, we can write the sub-rank for the i-th child as

   $$s_v(i) = \begin{cases} R_v(i), & \text{if } i = 1 \\ \left\lfloor \frac{R_v(i)}{B_v(i-1)} \right\rfloor, & \text{else} \end{cases}$$

   with

   $$R_v(i) = \begin{cases} r_l, & \text{if } i = |v| \\ R_v(i+1) \bmod B_v(i), & \text{otherwise} \end{cases}$$

3. We add operator v_k to plan and repeat this step for each child, i.e. for the i-th child we unrank $(s_v(i), G_i)$ where G_i is the group for this child.

4. We repeat steps 1 through 3 recursively until we reach the terminal operators.

Unranking is in $O(m)$, m being the number of operators in the tree which is limited by the number of groups in the MEMO. In terms of running time, unranking takes only a small fraction of the time needed for counting and is thus negligible.

4 Verifying Query Processors

In [11], Slutz presents a tool to generate SQL statements probabilistically, to increase the test coverage of the database engine. One simple advantage of this approach is the sheer speed at which new, different tests are generated, making it a very effective testing tool. The same claim can be made for our schema of selection and execution of multiple plans given a single query, which increases even further the coverage of query optimizer and execution logic.

In our current implementation in Microsoft's SQL Server, we extend the SQL syntax with an option to specify what plan to use for the execution. The SQL statement shown in Figure 4 causes the optimizer to build the MEMO structure, count the possible plans, and select plan number 8 for execution.

```
SELECT  *
FROM    Professors P, Students S, Enrolled E,
        Courses C
WHERE   S.Name = "Sam White" AND
        S.SID = E.SID AND
        E.Title = C.Title AND
        C.By = P.PID
OPTION  (USEPLAN 8)
```

Figure 4: Query with USEPLAN directive

Using scripting primitives, any given query can be extended easily with the OPTION clause and a loop construct that iterates over a deterministically or randomly selected set of possible plans. This way developers are able to generate test cases for specific queries, instantly extending existing test libraries substantially.

The main advantages of using these techniques in testing are:

1. It is easy to generate large test sets for the engine to scrutinize both correctness of the query execution and its performance.

2. The results are simple to verify since all plans should deliver the same outcome. The probability that an incorrect result is overlooked is rather small as opposed to conventional testing where each result requires manual verification.

3. It is possible to test operator implementations that the optimizer would not chose for the current state of the test database.

4. Optimizer decisions and correct assembling of plans by the optimizer can be easily verified. This point is of particular importance when extending the set of both operators and their implementations.

		In a sample of 10000				
Query	#Plans	Min°	Mean°	Max°	costs°≤ 2	costs°≤ 10
Q5	68572049	1.14	17098	4034135	0.47%	12.15%
Q7	228107572	1.15	3318	178720	0.11%	44.55%
Q8	20112521035	1.01	111	609	1.11%	14.7%
Q9	67503460	1.10	4107	109825	0.11%	4.08%
Q5*	455348910	1.23	105418	1287700	0.29%	5.70%
Q7*	3907373772	1.48	1793052	1523086611	0.03%	2.79%
Q8*	4432829940185	1.31	28159718	32595091399	0.06%	1.85%
Q9*	250657568	1.30	38363213	35866936219	0.02%	7.00%

°as factor of the optimum (=scaled costs); *including Cartesian products

Table 1: Parameters of search spaces of TPC-H join queries.

5. The verification and calibration of cost formulas is no longer restricted to one single plan per query but can also check cost values of sub-optimal plans.

6. The enumeration of complete search spaces for small queries helps check and analyze optimizer principles like cost-bound pruning and search strategies.

The features described are part of the routine testing in the development of Microsoft's SQL Server.

5 Cost Distributions

Besides their practical application to testing which was the driving force behind our efforts, the techniques presented are of importance for the experimental analysis of cost distributions, which we believe to be a promising area of research. Cost distributions capture the frequency of plans of certain costs, and they can be indicators of the difficulty of a query in that they show how many plans of what quality there are in the whole space.

Ioannidis and Kang were the first to report on cost distributions explicitly, i.e. they performed a sampling of the search space for the restricted problem of join ordering [6]. They pointed out that knowledge of the cost distribution helps understanding certain effects occurring in optimization, specifically needed for the tuning of probabilistic optimization techniques. They developed a search space model based on this analysis which provides useful insights into the working of randomized join ordering. The distributions they found were asymmetric and resembled Gamma-distributions implying certain topological structures in the search space. They attributed their findings to the particular cost model used.

However the question as to what degree do those results apply to the unrestricted, general case of query optimization is still open so far.

Using our framework we are able to perform a fair random sampling of costs in the search spaces that are not limited to join ordering only but may include arbitrary relational operators, various kinds of indexes and aggregates, and even cover parallel processing. We carried out numerous experiments with both standard benchmark queries like TPC-H and customer queries taken from various applications. Under the precondition that the queries were of sufficiently large size, i.e., involving more than 4 or 5 joins, the distributions obtained were characterized by a relatively strong concentration of costs relatively close to the optimum, asymmetric, and often resembling exponential distributions. These shapes correspond to Gamma-distributions with shape parameter close to 1, which were also observed by Ioannidis and Kang.

Figures presented here are the result of experiments with TPC-H queries 5, 7, 8, 9, which are the join-intensive queries of the benchmark, and have a larger search space. Table 1 summarizes some of the relevant values obtained. The first four rows consider a space of alternatives that does not allow cross products; while the last four rows allow cross products. Each experiment consists of a random sample of 10,000 plans from the space. All costs are normalized to the optimum plan found by the optimizer, which has cost 1.0.[2] The "min" column shows that with a relatively small sample from the space, it is possible to find plans that are pretty close to the optimum. In fact, the percentage of plans that are within twice the optimum cost is non-trivial. Also, it should be noted that the results are slightly different for the different queries, which vary in their selectivity and other properties. But the same trends can be seen in all the experiments.

Figure 5 shows histograms of the cost distributions discussed. All plots show that almost all plans are

[2]The measure of a very large number of plans in the space considered by the optimizer does not imply that a structure requires as many bytes—recall that the plans are obtained through composition and reuse of operators, from the compact encoding of the MEMO structure.

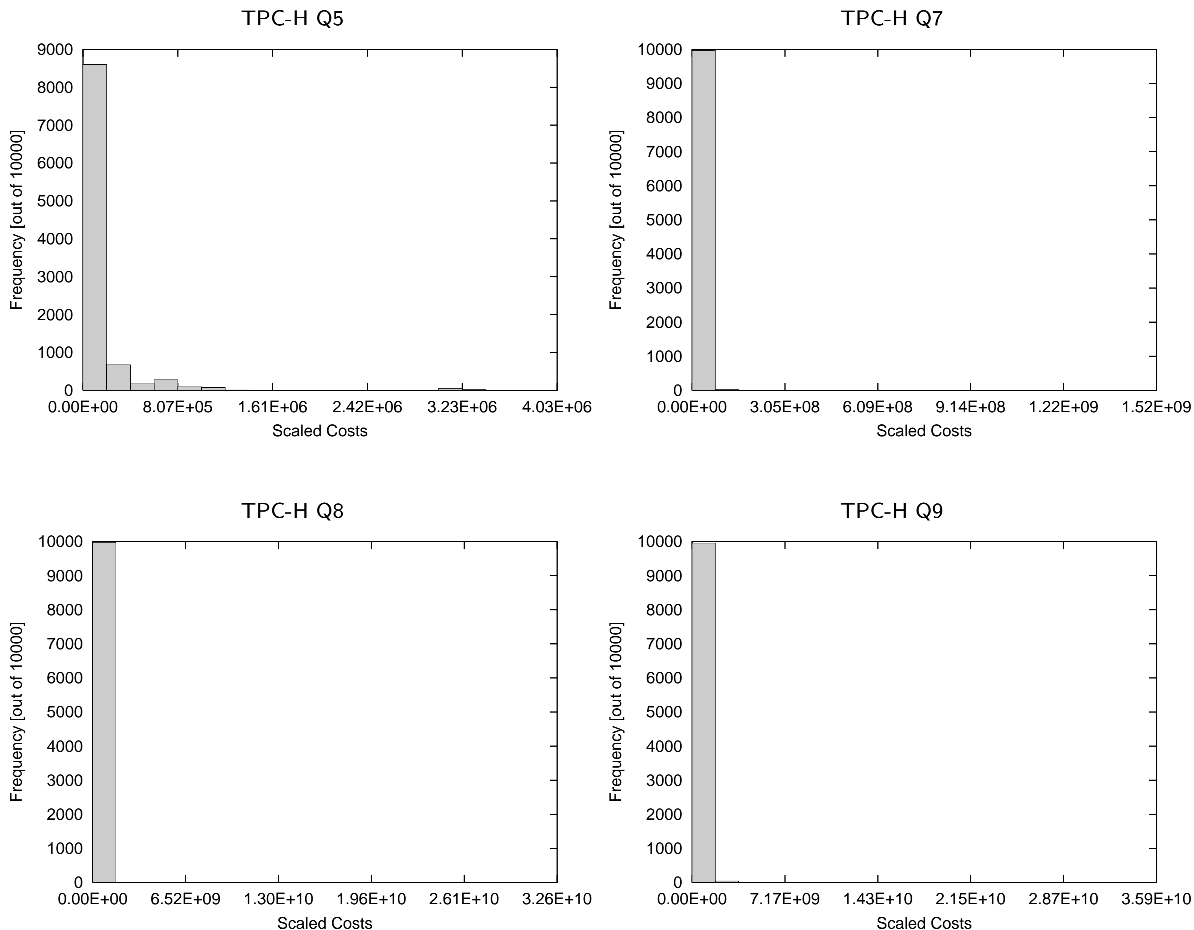

Figure 5: Cost distributions for TPC-H Query 5,7,8, and 9.

within the lower 10% of the entire cost range, suggesting a Gamma-distribution of costs. Figure 6 shows zoom-ins to the lower 50% sampled costs; that is, the part of the distribution that makes up for 50% of the space with the optimum as left edge. Still all four plots show a very strong resemblance with Gamma distributions. However, little disturbances are visible, particularly in case of Query 5.

Finally, Figure 7 shows a further zoom, to the points that are up to 50 times the cost of the optimum. In the "macro" view, we find that plans tend to be clustered to the left, close to the optimum solution. As we zoom in to the dense area, the histograms get less smooth, but they still seem to suggest Gamma distributions.

Our findings lend strong support to the assumption that cost distributions of the form detailed above are characteristic to query optimization and are of a much more general nature than first suspected by Ioannidis and Kang.

The distributions of queries that contained few tables were of no particular shape but consisted only of random noise (e.g. TPC-H 6). Although it is hypothetically possible to devise queries of arbitrary size where the cost distribution degenerates to a single point—e.g. the cross product of several instances of the same table, with a space restricted to be linear joins—we never observed any such tendency in practical instances or customer queries.

These results are only preliminary and further research is needed to investigate this subject. Besides the observation that cost distributions are generally of a certain shape, it would be particularly interesting to know what parameters are responsible in order to predict the distribution analytically.

6 Summary

Query optimizers select one execution plan out of a large number of alternatives considered, and traditional testing can verify only this one plan. In this paper we developed primitives to generate either the whole

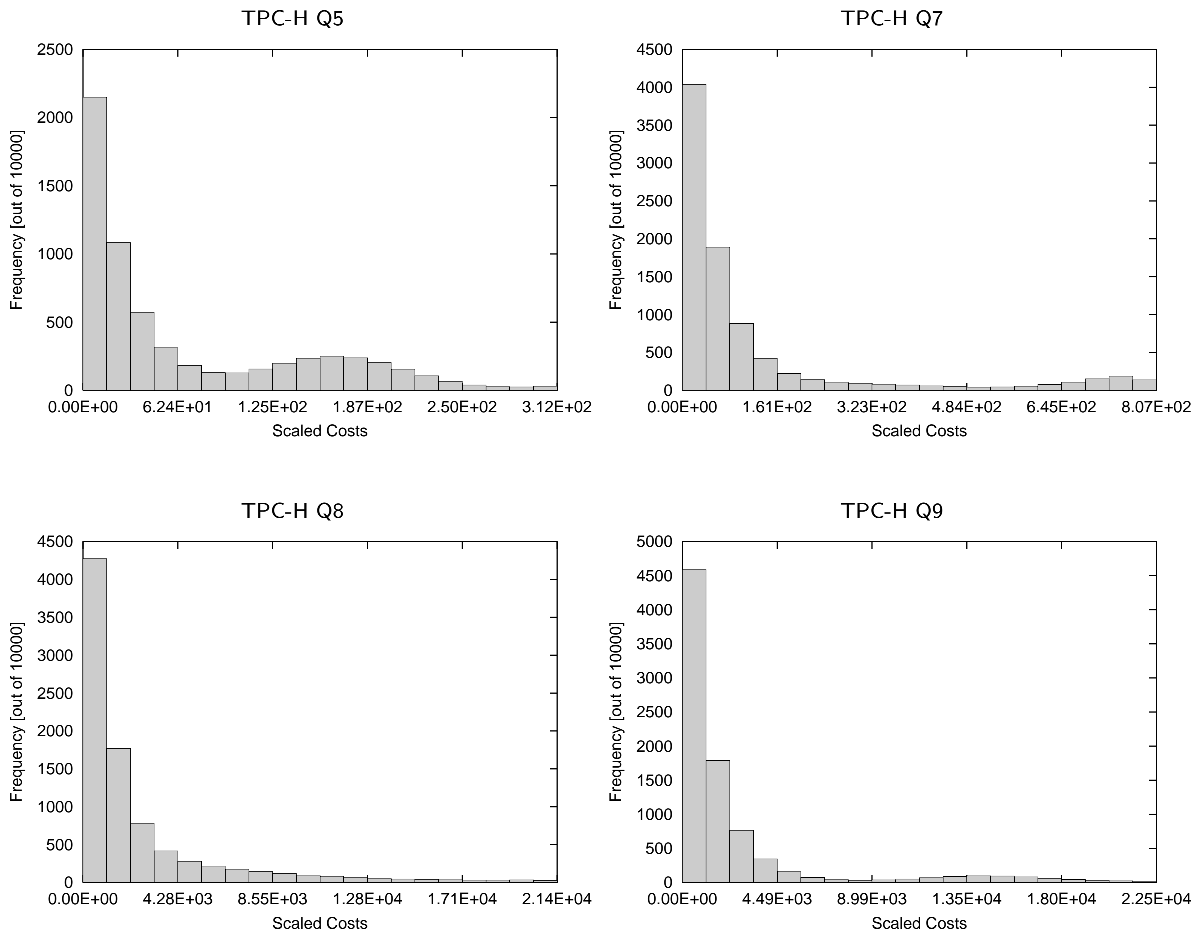

Figure 6: Cost distributions for TPC-H Query 5,7,8, and 9; lower 50% sampled costs.

space of alternatives, or a uniform random sample within that space. The problem is challenging because cost-based optimizers do not represent entire execution plans explicitly, but rather rely on data structures that maximize sharing of common expressions between candidate plans.

By opening up the space of alternatives to stocastic testing, we are able to validate the optimizer logic, and exercise the execution engine effectively. Unexpected interactions between different transformation rules can be seen, and execution iterators are tested in uncommon, but possible configurations. This provides a valuable tool to certify and increase the quality of a query processor, which would be difficult to match using only hand-crafted examples, either written by testers or obtained from customers.

Our validation tool is unintrusive to the workings of the optimizer, and it can be implemented separately, as long as it can access the table of alternatives constructed during optimization. A small extension to the language provides access to the functionality, so it is easy to write scripts to do the extensive testing.

A further use of our enumeration and sampling primitives is the study of the search space itself. What was it all that the optimizer considered, and how does it compare with the actual optimal plan? We were able to obtain for the first time some initial results on cost distributions of *real* search spaces. Results on cost distributions are important for work on randomized query optimization, and we are also interested in their use to characterize the difficulty of particular problems —and the optimization effort required to solve them. This is a subject for future research.

References

[1] J. A. Blakeley, W. J. McKenna, and G. Graefe. Experiences Building the Open OODB Query Optimizer. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 287–296, Washington, DC, USA, May 1993.

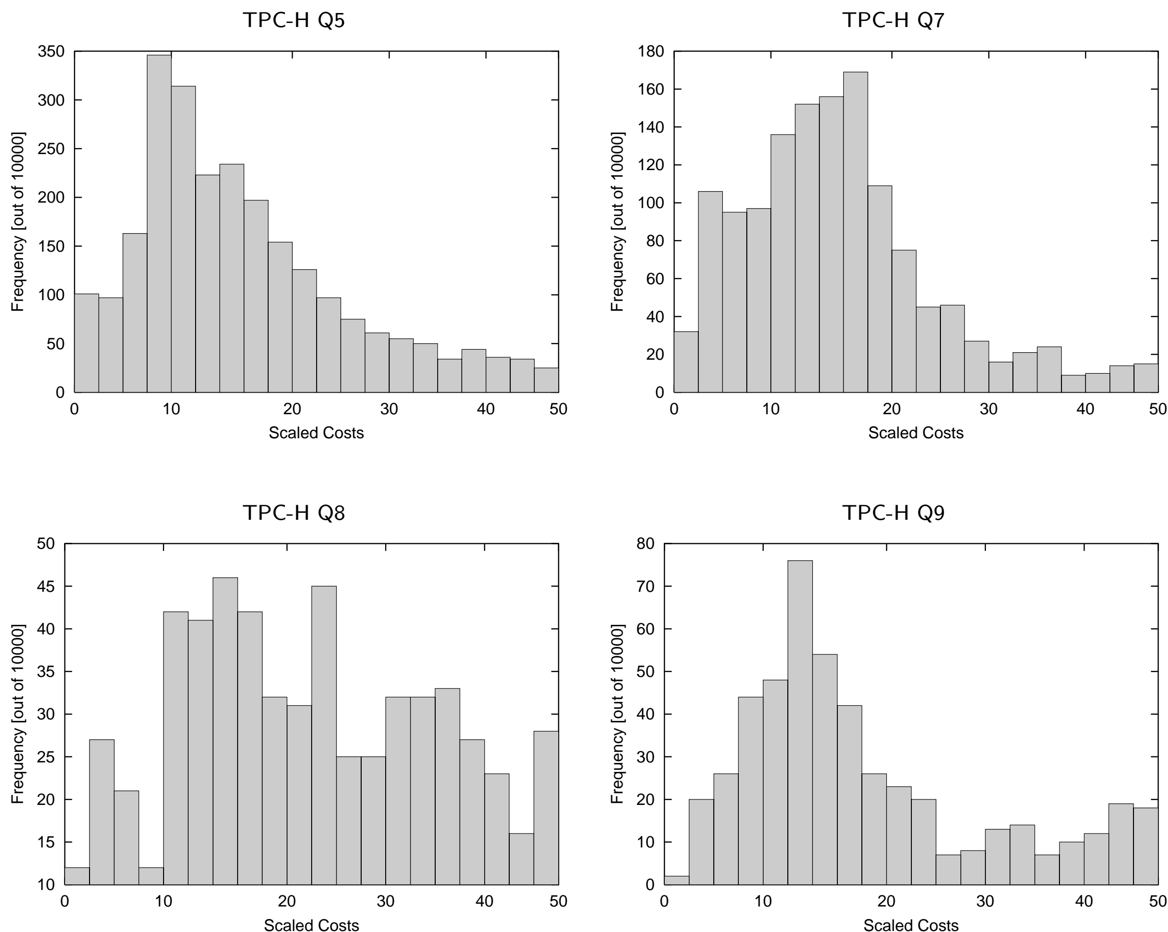

Figure 7: Cost distributions for TPC-H Query 5,7,8, and 9; blow-up of the interval $[0, 50]$.

[2] C. A. Galindo-Legaria, A. Pellenkoft, and M. L. Kersten. Uniformly-distributed Random Generation of Join Orders. In *Proc. of the Int'l. Conf. on Database Theory*, pages 280–293, Prague, Czech Republic, January 1995.

[3] G. Graefe. The Cascades Framework for Query Optimization. *IEEE Data Engineering Bulletin*, 18(3):19–29, September 1995.

[4] G. Graefe and D. J. DeWitt. The EXODUS Optimizer Generator. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 160–172, San Francisco, CA, USA, May 1987.

[5] G. Graefe and W. J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proc. of the IEEE Int'l. Conf. on Data Engineering*, pages 209–218, Vienna, Austria, April 1993.

[6] Y. E. Ioannidis and Y. C. Kang. Left-Deep vs. Bushy Trees: An Analysis of Strategy Spaces and its Implications for Query Optimization. In *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, pages 168–177, Denver, CO, USA, May 1991.

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

[8] K. Ono and G. M. Lohman. Measuring the Complexity of Join Enumaration in Query Optimization. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 314–325, Brisbane, Australia, August 1990.

[9] A. Pellenkoft, C. A. Galindo-Legaria, and M. L. Kersten. The Complexity of Transformation-Based Join Enumeration. In *Proc. of the Int'l. Conf. on*

Very Large Data Bases, pages 306–315, Athens, Greece, September 1997.

[10] R. Ruskey and T. C. Hu. Generating Binary Tree Lexicographically. *SIAM Journal of Computation*, 6(4):745–758, December 1977.

[11] D. Slutz. Massive Stochastic Testing of SQL. In *Proc. of the Int'l. Conf. on Very Large Data Bases*, pages 618–622, New York, NY, USA, September 1998.