ISSN:0975-9646

A Novel Optimization Technique for Determination of Parametric Queries

Arepu Yuvakishore & Kumar Vasantha
Department of CSE
Avanthi Institute of Engg & Tech
Tamaram, Visakhapatnam, India.

Abstract- In this paper we proposed to progressively explore the parameter space and build a parametric plan during several executions of a query. Novel algorithms which resemble parametric plans are populated, are able to frequently bypass the optimizer but still execute optimal or near-optimal plans. It is known that commercial applications usually rely on precompiled parameterized procedures to interact with a database. Unfortunately, executing a procedure with a set of parameters different from those used at compilation time may be arbitrarily suboptimal. Hence Parametric query optimization (PQO) attempts to solve this problem by exhaustively determining the optimal plans at each point of the parameter space at compile time.

Keywords - Parametric query optimization, frequency estimation, dynamic recompilation.

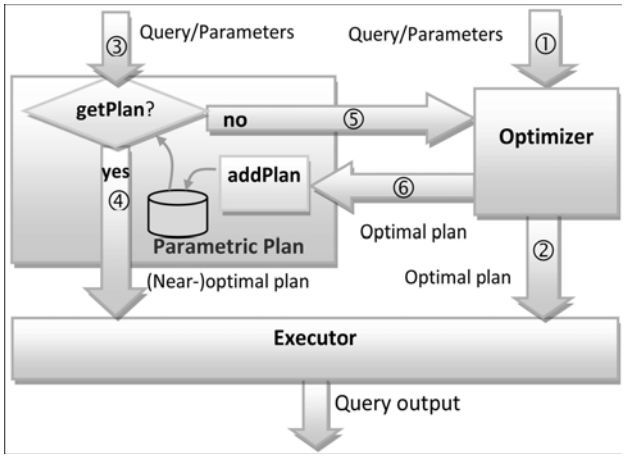
I. INTRODUCTION

The values of runtime parameters of the system, data, or queries themselves are unknown when queries are originally optimized but in certain situations two methods are used these scenarios, there are typically two trivial alternatives to deal with the optimization and execution of such parameterized queries. One approach, termed here as Optimize-Always, is to call the optimizer and generate a new execution plan every time a new instance of the query is invoked. Another trivial approach, termed Optimize-Once, is to optimize the query just once, with some set of parameter values, and reuse the resulting physical plan for any subsequent set of parameters. Both approaches have clear disadvantages. In addition, Optimize-Always may limit the number of concurrent queries in the system, as the optimization process itself may consume too much memory. On the other hand, Optimize-Once returns a single plan that is used for all points in the parameter space.

However, in reality, the cost functions of physical plans and regions of optimality are not so well behaved. A more important problem results from the fact that PQO has a much higher start-up cost than optimizing a query a single time (PQO usually requires several invocations of the optimizer with different parameters [1], [2]). When a previously unseen query arrives, it is therefore not clear to determine whether PQO should be used: it may not be cost-effective to solve the full PQO problem if the query is not executed frequently or if it is repeatedly executed with values covering a small subspace of the entire parameter space.

II. PROGRESSIVE PARAMETRIC QUERY OPTIMIZATION

The main goal / idea of Progressive parametric query optimization (PPQO) solves the solution to the PQO problem as successive 582 query execution calls are submitted to the DBMS as given in Fig. 1,



This shows a high-level architecture of our approach. Given a query and its parameter values, a traditional optimizer returns the optimal execution plan along with its estimated cost. In contrast, a PPQO-enabled optimizer introduces a data structure called PP, which incrementally maintains plans and optimality regions, allowing us to reuse work across optimizations. When a new instance of a parametric query arrives, PPQO tries to obtain an optimal (or near-optimal) plan by consulting the PP data structure. If it is successful, it returns such plan, and a full optimization call is avoided.

Otherwise, it makes an optimization call, and both the resulting optimal plan and cost are added to the PP for future use. Due to the size of the parameter space, PPs should not be implemented as exact lookup caches of plans because there would be too many "cache misses." Also, due to the nonlinear and discontinuous nature of cost functions, PPs should not be implemented as nearest neighbor lookup structures as there will be no guarantee that the optimal plan of the nearest neighbor is optimal or close to optimal for the point in the parameter space being considered [3], [4]. We now describe the PPQO problem in more detail, borrowing notation and definitions from the classic parametric optimization problem.

III. PARAMETER TRANSFORMATION FUNCTION

It is known that a value parameter refers to an input value of the parametric SQL query to execute, a cost parameter is an input parameter in the formulas used by the optimizer to estimate the cost of a query plan. Cost parameters are estimated during query optimization from value parameters and from information in the database catalog. (Physical characteristics that affect the cost of plans but do not depend on query parameters, such as the average tuple size or the cost of a random I/O, are considered physical constants instead of cost parameters).

Example 1. Table FRESHMEN (NAME, AGE) succinctly describes first-year graduate students. The age distribution of students is showed in Fig. 2. Consider queries of the following form:

SELECT * FROM FRESHMEN
WHERE AGE=X OR AGE=Y

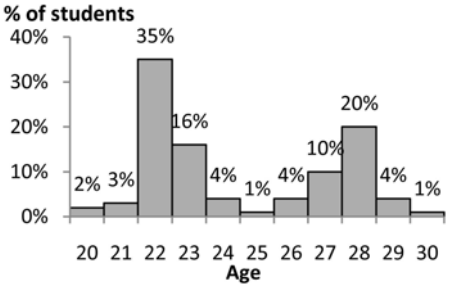


Fig. 2.Age distribution in table FRESHMEN

The parameters of this query can be represented as the absolute values used for parameters X and Y or as the selectivities of predicate age = X and predicate age = Y. Accordingly, the costs of physical P_{IDX} and P_{FS} can be represented in value-based parameter spaces, shown in Fig. 3, or in selectivity-based (also referred to as cost-based) parameter spaces, shown in Fig. 4.

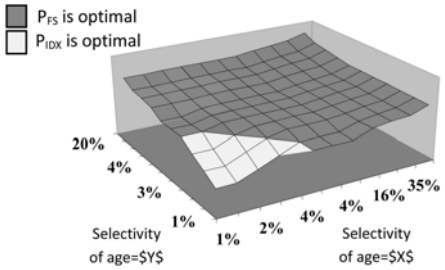


Fig. 3.Value-based parameter space

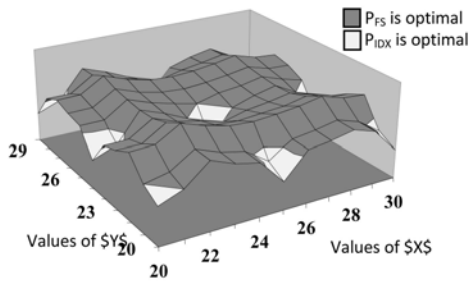


Fig. 4.Selectivity-based parameter space

It becomes much easier to characterize the regions of optimality using a cost-based parameter space than using a value-based parameter space. We assume that function Ψ' takes query Q and its SQL parameters, *vpt*, and returns *cpt* as a vector of selectivity's. Computing the selectivity's in cpt corresponds to the task of selectivity estimation, a subroutine inside of query optimization. We note that the arity of the value-based parameter space and that of the selectivity-based parameter space are not necessarily the same. On one hand, it is possible to have predicates of the form age > X and age < Y, where two value predicates are collapsed into a single selectivity value for the combined predicate. In our prototype and experimental evaluation, we use a simple one-to-one mapping between parametric predicates and selectivity values.

```
function processQuery (
   inputs: Query Q, ValuePoint vpt
   input/output: ParametricPlan pp )
01 CostPoint cpt ← φ(Q, vpt);      // ValuePoint to CostPoint
02 Plan p ← pp.getPlan(Q, cpt);    // what plan to use?
03 if (p == NULL)
04   Cost cost;                    // cost is output param below
05   p ← optimize(Q, vpt, cost);// finds optimal plan & cost
06   pp.addPlan(Q, cpt, p, cost);// stores plan & cost in pp
07 execute(p);
```

The reasons behind our choice are the following: 1) this is the mapping used in previous work on parametric optimization, 2) it can be implemented without deep knowledge about the underlying query optimizer, and 3) our experiments show that this simple model is very competitive.

IV. PARAMETRICS PLANS:

(a) Requirements and Goals: The main trade-off in PPQO is to avoid as many optimization calls as possible as long as we are willing to execute suboptimal—but close to optimal—plans (note that this goal has also been proposed in [5] and [6] in the context of classical PQO). Thus, PP implementations must obey the Inference Requirement below.

(b) Inference Requirement. After a number of add Plan calls, there must be cases where getPlan returns an (near-)optimal plan p for query Q and parameter point cpt, even if addPlan(*Q,cpt,p.cost*) was never called. Given a sequence of execution requests of the same query with potentially different input parameters, PPQO has therefore two conflicting goals:

Goal 1. Minimize the number of optimization calls.

Goal 2. Execute plans with costs as close to the cost of the optimal plan as possible.

Consider a trivial cache implementation of the PP interface, which stores (*Q, cpt*) pairs as the lookup key and (*p, cost*) as the inserted value. This implementation cannot fulfill the inference requirement because it would return hits only for

previously inserted (*Q,cpt*) pairs. In the next we propose two PPQO implementations, each giving priority to one of the above goals.

V. THE BOUNDED – PPQO IMPLEMENTATION

The proposed PPQO implementations known as Bounded-PPQO or simply bounded. This implementation provides guarantees on the quality of the plans returned by getPlan (*Q, cpt*), thus focusing on Goal 2 of PPQO (see previous). Either the returned plan p is null (and an optimization call cannot be avoided) or p has a cost guaranteed to be within a user specified bound of the cost of the optimal plan. Specifically, the cost of plan p returned by get next is guaranteed to be bounded by OptCost_M+A, where OptCost is the cost of the optimal plan, and M≥1 and A≥0 are user-defined constants. Both M and A can be used to specify different bounds on suboptimality and are generally application specific.

The ellipse-ppqo implementation

Bounded's getPlan provides strong guarantees on the cost of plans returned. However, we expect low hit rates of Bounded's getPlan for small values of M and A or before Bounded's T_Q has been populated. In this we propose the Ellipse-PPQO (or simply Ellipse) implementation of the PP interface, designed to address Goal 1. For that purpose, Ellipse's get Plan returns-acceptable plans rather than guaranteed near-optimal costs.

It follows from the definition of Δ acceptable that if p is optimal at cpt_1 and cpt_2, then p is 1-acceptable only on points between cpt_1 and cpt_2 and p is 0-acceptable at all points. Note that in a two-dimensional space, the area where p is Δ acceptable is equivalent to the definition of an ellipse; if p is optimal for cpt1 and cpt2, then p is Δ acceptable at cpt if cpt is on or inside an ellipse of foci cpt1 and cpt2 such that the distance between the foci, ||cpt1-cpt2||. Over the sum of the distances between cpt and the foci, ||cpt-cpt1||+ || cpt- Cpt2||, is at least Δ shows the areas where p is 0.5-acceptable, 0.8-acceptable, and 1-acceptable if p is optimal at cpt1 and cpt2. Ellipse-PPQO encodes the heuristic that if a plan p is optimal in two points cpt1 and cpt2, then p is likely to be optimal or near-optimal in a convex region that encloses cpt1 and cpt2. Note that a nearest neighbour algorithm could be used as an alternative to Ellipse-PPQO. However, since regions of optimality are frequently long and narrow [4], for any given cpt point, the closest known plan could very well be from another region of optimality. In addition, Δ acceptable areas can easily encode both small and large regions of optimality

Implementation of addPlan for Ellipse: The implementation of addPlan for Ellipse proceeds as follows: For each query Q and for each plan p that is optimal in some point of the parameter space, Ellipse's addPlan(Q; cpt; p; cost) essentially maintains a list of (cpt; cost) pairs, where p is optimal for Q

VI. EVALUATION OF EXPERIMENT

An experimental evaluation of PPQO using Microsoft SQL Server 2005. The client application implements the pseudocode and Microsoft SQL Server is used to obtain estimated optimal plans and estimated costs of plans.

Data Set, Metrics, and Setup

Table 1 shows which tables are joined by each query. The tables are line item (L), orders (O), customer (C), supplier (S), part (P), partsupp (T), nation (N), and region (R). As in the work of Reddy and Haritsa [4] and unless otherwise noted, we added two extra selections to the TPC-H queries to more easily explore the parameter space. The two selections are of the form $col_i \leq val_i$, i=1,2 where for each query, col_i is one of the two columns shown in Table 1, and val_i is a random value from the domain of the column. For each query tested, we generated 10,000 random val_1 and val_2 values. ($A(val_1,val_2)$pair is a ValuePoint.) To guarantee that random parameter values uniformly explore the parameter space, we altered the values in the columns subject to the extra selections to such that those values are uniformly distributed in their domains instead of using the nonuniform TPC-H generated distributions. For each query and each ValuePoint vpt, we make a getPlan lookup call where PP is either Optimize- Once, Optimize-Always, Bounded, or Ellipse. If getPlan returns a plan, we call it a hit and check if the plan is optimal; if it is not optimal, we check how its estimated cost compares with the estimated optimal cost.

VII. VARIATION ON HIT RATE AND OPT RATE

The first experiment consisted of processing queries using 10,000 different random ValuePoints (value vectors) for each query and observing how HitRate and OptRate varied for Bounded and Ellipse. This experiment was performed for the five TPC-H queries listed in Table 1, and the results for three are shown in Figs. Several trends can be observed:

- Ellipse always has a higher HitRate than Bounded.
- Except for Query 8 (more on this below), Bounded always has a higher OptRate than Ellipse.
- HitRate converges quickly, but OptRate converges slightly faster.
- HitRate monotonically increases as a function of QP (more processed queries imply more misses, and each miss adds information to the ParametricPlan, therefore increasing the likelihood of future hits).
- OptRate naturally varies up and down, as the initial random (cpt,plan,cost) triples are added to the ParametricPlan object, until it converges.

Number of Plans and of Points, Space, and Time

Storing the number of plans and the number of points took only between ~600 Kbytes to ~1,300 Kbytes using the original uncompressed XML plan representations provided by SQL Server. Storing zip-compressed XML plans instead would decrease the size of the plan representation by a factor of 10. (Plans do not need to be understood, zipped, or unzipped by addPlan or getPlan functions.)It reports the time and space taken by the Bounded and Ellipse approaches during optimization. Time (in seconds) includes the time elapsed during optimization (if there is a miss), during addPlan, and during getPlan but not the execution time nor the time consumed by function Ψ. For comparison purposes, the time taken for Optimize-Once and Optimize-Always is also included. After 10,000 queries have been processed,

Optimize- Always took between 5.2 and 13.6 times longer than Bounded and between 10.7 and 18.5 times longer than Ellipse. Thus, although Boundedonly used between 7 percent and 20 percent of the optimization time, it still returned plans that were on the average just 1 percent more costly than the optimal plan. Ellipse used between 5 percent and 9 percent of the optimization time and returned plans that were 6 percent more costly than the optimal plan. Ellipse was always faster than Bounded because it had less optimize and addPlan calls (due to higher HitRates) and faster getPlan calls (because it has less information stored in its PPs). Note that although Optimize-Once spends the least optimization time, it is not the best overall approach. In fact, the entire PQO research area aims to overcome the performance problems of using Optimize-Once.

VIII. CONCLUSION

Before PPQO, processing parameterized queries was an allor-nothing approach: either the optimizer explores all the parameter space and computes the full PQO solution (traditional PQO) or it relies on luck and uses the very first plan it gets for a query. At execution time, PPQO selects which plan to execute by using only the input cost parameters without recosting plans. PPQO is an adaptive technique that works prior to execution (and assumes the optimizer to be correct—just like any other PQO approach). Query reoptimization [6] and other adaptive query processing (AQP) approaches [1], [4] work during optimization and execution and assume that the optimizer can make mistakes or that the system characteristics change significantly during the execution of a single query. Also, PPQO is an interquery adaptive approach, while AQP are frequently intraquery optimization approaches. PPQO is also amenable to be implemented in a complex commercial database system as it requires no changes in the optimization or execution processes.

In fact, our PPQO prototype ran outside the DBMS server. For technical reasons, we did not implement function Ψ ourselves but instead used SQL Server's cost model to transform value into cost parameters. For that reason, we did not evaluate the impact of such function in our experimental evaluation. PPQO was evaluated in a variety of settings, with queries joining up to table, with multiple sub queries, up to four parameters, and in plan spaces with close to 400 different optimal plans. PPQO yielded good results in all scenarios except for the Bounded algorithm in complex queries using a four-dimensional parameter space. However, even in this challenging scenario, Ellipse on the average executed plans just 3 percent more costly than the optimal, while avoiding 87 percent of all optimization calls.

REFERENCES

[1] A. Hulgeri and S. Sudarshan, "Parametric Query Optimization for Linear and Piecewise Linear Cost Functions," Proc. 28th Int'l Conf. Very Large Data Bases (VLDB), 2002.

[2] A. Hulgeri and S. Sudarshan, "AniPQO: Almost Non-Intrusive Parametric Query Optimization for Nonlinear Cost Functions," Proc. 28th Int'l Conf. Very Large Data Bases (VLDB), 2003.

[3] D. Harish, P. Darera, and J. Haritsa, "On the Production of Anorexic Plan Diagrams," Proc. 33rd Int'l Conf. Very Large Data Bases (VLDB), 2007.

[4] N. Reddy and J.R. Haritsa, "Analyzing Plan Diagrams of Database Query Optimizers," Proc. 31st Int'l Conf. Very Large Data Bases (VLDB), 2005.

[5] S. Ganguly, "Design and Analysis of Parametric Query Optimization Algorithms," Proc. 24th Int'l Conf. Very Large Data Bases (VLDB), 1998.

[6] N. Kabra and D.J. DeWitt, "Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans," Proc. ACM SIGMOD, 1998.

[7]V.G.V. Prasad, "Parametric Query Optimization: A Geometric Approach," MSc thesis, IIT, Kampur, 1999.

[8] S.V.U. Maheswara Rao, "Parametric Query Optimization: A Non-Geometric Approach," master's thesis, IIT, Kampur, 1999.

AUTHORS:

Mr. AREPU YUVAKISHORE received the B.Tech degree in S.V.H College Of Engineering in 2005 under Acharya Nagarjuna University and He is currently pursuing M.Tech in Software Engineering at Avanthi Institute of Engineering and Technology, Vishakhapatnam His research interests include Data Mining and Query Optimization.

Mr. KUMAR VASANTHA received the M. Tech degree from the Department of Computer Science and Engineering, Avanthi Institute of Engineering and Technology, Vishakhapatnam, JNT University, Kakinada in 2009 and working as an Assistant Professor in Avanthi Institute of Engineering and Technology, Vishakhapatnam. His research interests include Information Security and Data Mining.