SWATHI SAMBANGI, NAGARAM PHANI KUMAR, KIRAN KATTA, RAJEEV GANDHI BAKKA, DEEPIKA RANI / International Journal of Engineering Research and Applications (IJERA) ISSN: 2248-9622 www.ijera.com
Vol. 2, Issue 2,Mar-Apr 2012, pp.891-896

ENHANCED PROGRESSIVE PARAMETRIC APPROACH FOR QUERY OPTIMIZATION

SWATHI SAMBANGI, NAGARAM PHANI KUMAR, KIRAN KATTA, RAJEEV GANDHI BAKKA, DEEPIKA RANI

Abstract—
To interact with Database industrial application depends on precompiled parameterized procedures. Unfortunately, executing a procedure with a set of parameters different from those used at compilation time may be arbitrarily suboptimal. By identifying the optimal plans at each point of the parameter space at the time of compilation by using parametric query optimization the above mentioned issue can be solved .Parametric Query Optimization is likely not cost-effective if the executed with values only within a subset of the parameter space or if it is query is executed infrequently . As an alternative to progressively exploring the parameter space and building a parametric plan during several executions of the same query , we are going to propose an algorithm as parametric parametric plans are populated, are able to frequently bypass the optimizer but still execute optimal or near-optimal plans.

Introduction

Query optimization is a function of many relational database management systems in which multiple query plans for satisfying a query are examined and query plan is identified. This may or not be the absolute best strategy because there are many ways of doing plans. There is a tradeoff between the amount of time spent figuring out the best plan and the amount for running the plan. Different qualities of database management systems have different ways of balancing these two. Cost based query optimizers evaluate the resource footprint of various query plans and use this as the basis for plan selection. In many applications, the values of runtime parameters of the system, data, or queries themselves are unknown when queries are originally optimized. In 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. Optimize-Always requires an optimization call for each execution of a query instance. These optimization calls may be a significant part of the total query execution time, especially for simple queries. 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. The chosen plan may be arbitrarily suboptimal for parameter values different from those for which the query was originally optimized.

Typically the resources which are costed are CPU path length, amount of disk buffer space, disk storage service time, and interconnect usage between units of parallelism. The set of query plans examined is formed by examining possible access paths (e.g., primary index access, secondary index access, full file scan) and various relational table join techniques (e.g., merge join, hash join, product join). The search space can become quite large depending on the complexity of the SQL query. There are two types of optimization. These consist of logical optimization which generates a sequence of relational algebra to solve the query. In addition there is physical optimization which is used to determine the means of carrying out each operation. The goal is to eliminate as many unneeded tuples, or rows as possible. The following is a look at relational algebra as it eliminates unneeded tuples.

The project operator is straightforward to implement if <attribute list> contains a key to relation R. If it does not include a key of R, it must be eliminated. This must be done by sorting (see sort methods below) and eliminating duplicates. This method can also use hashing to eliminate duplicates Hash table.

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. As the PP data structure becomes populated, it is possible to completely bypass the optimization process without hurting the quality of the resulting execution plans. 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.

Previous Work

Before PPQO, processing parameterized queries was an all or 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. PPQO is able to progressively construct information about the parametric space and approximate optimality regions, being able to bypass the optimizer up to 99 percent of the times, while still returning plans within 5 percent of the cost-optimal plan for 99 percent of the cases. Previous 3 techniques used are:

1. Adaptive Query Processing
2. Plan Reduction or Plan Diagram
3. Compilation Queries

1. Adaptive Query Processing

It has been applied log running continuous query over data streams.There is no unifying comparison from one time bound another time bound.It can contains Large Body inter-related work communication process.It can provides the outdated statics representation.Some of the deisadvantages of using this kind of query processing are: It can provides some of the optimization errors.It can contains large data sets for executing the query processing environment.There is no prediction technique implementation.There is no environment change processing.

2. Plan Reduction Or Plan Diagram

It can contains more number of execution plans.Which execution plan can cover total space no one can identify.It can contains different types of patterns.Which pattern is the best scalability pattern no one can identified exactly. Some of the deisadvantages of using this kind of query processing are:It can contains redundancy patterns.It can contains more search space specification process.It can be identified as a NP hard problem.There is no fast estimators.

3. Compilation Queries

It can working based on query solution specification. Each and every query plan can takes how much execution time now can identify exactly. Which Query complier is the optimization query complier no one can identified specifically. Which path is best path no one can defined efficiently. Some of the disadvantages of using this kind of query processing are: Every Region maintains many number of query plans specification process. Some compilation of queries can takes more computation time specification process. It can contains optimal set solutions.

ARCHITECTURE OF QUERY PROCESSING

The main idea of PPQO is to incrementally solve (or approximate) the solution to the PQO problem as successive query execution calls are submitted to the DBMS. Fig. 1 shows a high-level architecture of our approach.

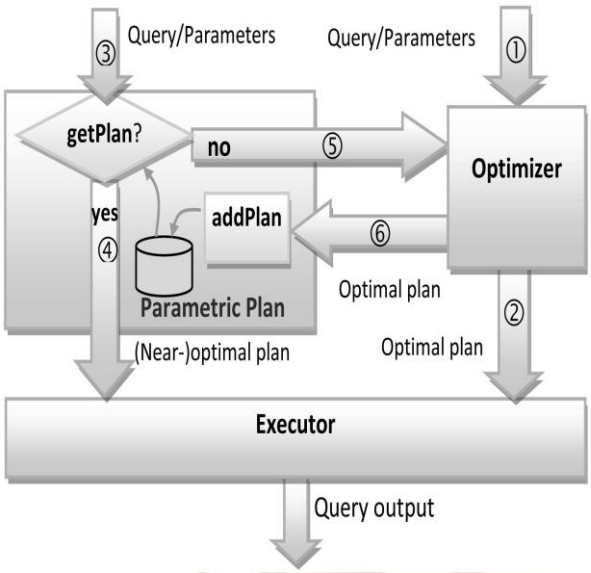

Fig.1: Processing a Query

Given a query and its parameter values, a traditional optimizer returns the optimal execution plan along with its estimated cost ((1) and (2) in the figure). 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. As the PP data structure becomes populated, it is possible to completely bypass the optimization process without hurting the quality of the resulting execution plans. When a new instance of a parametric query arrives ((3) in Fig. 1), 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 ((4) in Fig. 1). Otherwise, it makes an optimization call ((5) in Fig. 1), and both the resulting optimal plan and cost are added to the PP for future use ((6) in Fig. 1).

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 neighbour lookup structures as there will be no guarantee that the optimal plan of the

nearest neighbour is optimal or close to optimal for the point in the parameter space being considered.

A parametric query Q is a text representation of a relational query with placeholders for m parameters vpt = (v1….vm). Vector vpt is called a Value Point. Examples of parameter values are system parameters (e.g., available memory) and query-dependant parameters (e.g., constants in parametric predicates). Using vpt directly to model the parameter space and characterize regions of optimality for plans is in general difficult To address this problem, a transformation function Ω is used, which is optimizer specific and transforms Value Points into Cost Points. A Cost Point is a vector cpt = (c1…..cn), where each ci is a cost parameter with an ordered domain. A well known implementation of Ω is transforming parametric predicate values into the corresponding predicate selectivities.

For instance, consider predicate age < X, with parameter X. Function Ω would then map a specific constant c for X into the selectivity of the nonparametric predicate age < c. Let p be some execution plan that evaluates query Q for a given vpt. The cost function of p, denoted p(cpt), takes a Cost Point cpt as an input and returns the cost of evaluating plan p under cpt. For every legal value of the parameters, there is some plan that is optimal. Given a parametric query Q, the maximum parametric set of plans (MPSP) is the set of plans, each of which is optimal for some point in the n-dimensional cost-based parameter space. The region of optimality for plan p, denoted r(p), is defined as

r(p)={(t1……..tn) | p is optimal at (c1 = t1; . . . ; cn = tn)}

Finally, a parametric optimal set of plans (POSP) is a minimal subset of MPSP that includes at least one optimal plan for each point in the parameter space. 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 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. A crucial cost parameter that is used during optimization is the estimated number of tuples in (intermediate) relations processed by the query plan: most query plans have cost formulas that are monotonic in the number of tuples processed by the query.

Implementation

The current application is implemented on an Insurance management System. In that system for each and every query execution, these optimization techniques are called and the queries are implemented. The functional requirements which we are going to propose in this paper are as follows:

1. The Query cost should be minimal.
2. The Query has to consume minimum recourses like CPU cycles, memory etc.
3. The Optimizer has to be implemented for query optimization.
4. Query Optimization should be done with PPQO.

The proposed Modules in this paper are:

- Parametric Query Representation
- Parameter Transformation Function
- Parametric Plan Interface
- Bounded PPQO implementation
- Efficient implementation of get plan

Parametric Query Representation

A parametric query Q is a text representation of a relational query with placeholders for m parameters vpt =(v1; . . . ; vm).

Vector vpt is called a Value Point. Examples of parameter values are system parameters (e.g., available memory) and query-dependant parameters (e.g., constants in parametric predicates). We focus on query-dependant parameters since they cover the most common scenarios.

Parameter Transformation Function

Recall that a value parameter refers to an input value of the parametric SQL query to execute. On the other hand, 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.

Parametric Plan Interface

We component of PPQO by describing its two main operations:

Add Plan Q; cpt; p; cÞ. This operation registers that plan p, with estimated cost c, is optimal for query Q at Cost Point cpt.

Get Plan Q; cptÞ. This operation returns the plan that should be used for query Q and cost values cpt or returns null if no plan is considered good enough for Q. Now give an operational description of the PP

```
Optimize-Always implements PP
    addPlan(inputs: Query Q, CostPoint cpt,
                    Plan p,  Cost cost)
    return; // does nothing

    getPlan(inputs:  Query Q, CostPoint cpt;
            outputs: Plan p)
    return null;
```

Fig : Optimize always implementation

```
Optimize-Once implements PP
   private Plan p = null;

   addPlan(inputs: Query Q, CostPoint cpt,
                   Plan plan, Cost cost)
   if (p == null) p = plan;   // saves first plan

   getPlan(inputs:  Query Q, Cost-Point cpt;
           outputs: Plan plan)
   return p;                  // returns first plan
```

Fig : Optimize once implementation

Bounded PPQO implementation

The intuition for the Bounded-PPQO implementation is given as follows: Consider a parametric query with two parameters. If plans pi and pj are optimal in some Cost Points cpti and cptj, which delimit a box.

```
addPlan (inputs: Query Q, CostPoint cpt,
                 Plan p, Cost cost) {
01 List TQ ← getList(Q);     // Gets the list of triples for Q
02 if (TQ ==null)
03    TQ = new List();       // If no list, create one
04 TQ.insert(cpt, p, cost); // Inserts triple in cost order
05 setList(Q, TQ);}          // adds/replaces TQ into catalog
```

Fig : Bounded's addplan Implementation

```
getPlan(inputs: Query Q, Cost-Point cpt;
        outputs: Plan plan)

01 List TQ ← getList(Q);      // gets list of triples for Q
02 for each (t1, t2) in TQ    // look any pair of triples
03   if (t1.cost≤t2.cost≤t1.cost*M+A and
        t1.cpt ≺ cpt ≺ t2.cpt)
04       return t1.p;
05 return null;
```

Fig : Bounded's Getplan implementation

Efficient implementation of get plan

The naive implementation of get Plan in enumerates all pairs of tuples, that were introduced by add Plan and tests if any pair bounds cpt. If some pair (ti; tj) bounds cpt, then plan pj can be returned as the answer to get Plan.

The complexity of this procedure is clearly quadratic in the size of TQ. To avoid the enumeration of all of pairs of triples that have to be checked, we apply an optimization that allows us to choose a single pair of triples (t1; t2) to be checked.

```
getPlan(inputs: Query Q, Cost-Point cpt;
        outputs: Plan plan)
01 List TQ ←getList(Q);          // gets list of triples for Q
02 if (TQ ==null) return null;
03 Triple last=null;             // last triple of TQ≺cpt
04 for Triple t in TQ            // in cost order
05    if (t.cpt ≡ cpt) return t.p; // exact match
06    else if (t.cpt ≺ cpt) // keep track of last triple of TQ≺cp
07       last = t;
08    if (t.cpt ≻ cpt)          // first triple of TQ≻cpt
09       if (last == null) return null;
10       if (last.c ≤ t.c ≤ last.c*M+A)
11          return t.p;
```

Fig : Effecient Getplan Implementation

RESULTS

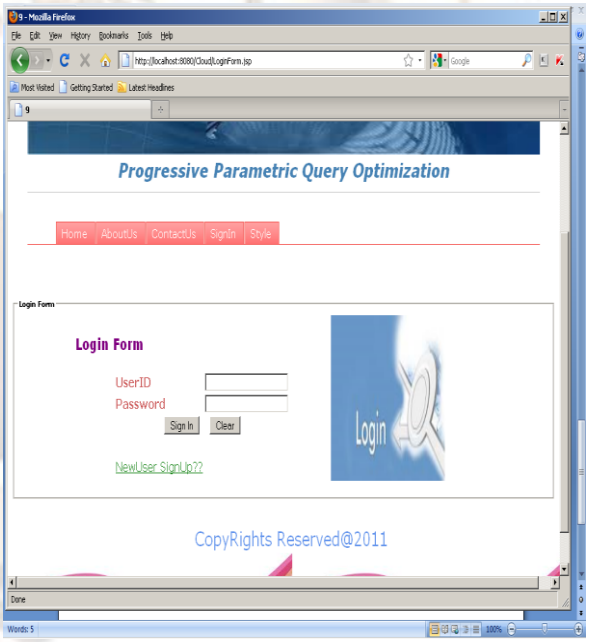

Fig : Login Page of PPQO

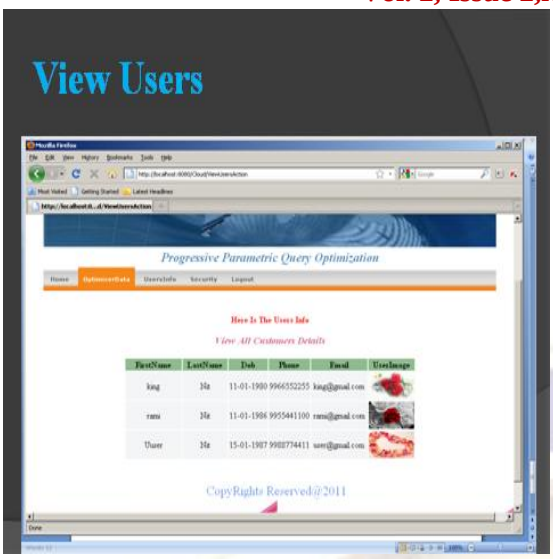

Fig : TO View all customer information

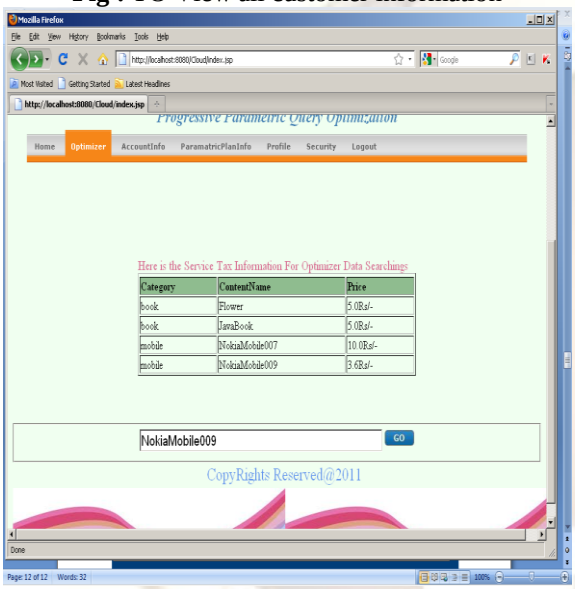

Fig : To retrieve optimizer data

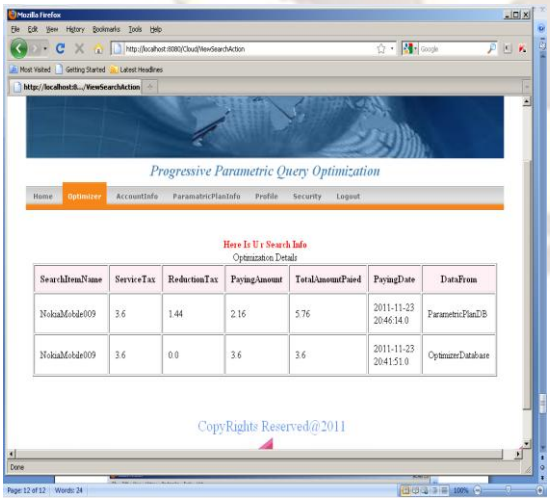

Fig : user's optimization data

Conclusion

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, PPQO prototype ran outside the DBMS server. However, it is important to note that the function can be implemented by simply manipulating in memory histograms, which is a negligible fraction of optimization time and would not have resulted in any noticeable difference in our experimental evaluation. PPQO was evaluated in a variety of settings, with queries joining up to eight tables, with multiple sub queries, up to four parameters, and in plan spaces with close to 400 different optimal plans.

References

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

[2] A. Ghosh, J. Parikh, V.S. Sengar, and J.R. Haritsa, "Plan Selection Based on Query Clustering," Proc. 28th Int'l Conf. Very Large Data Bases (VLDB), 2002.

[3] G. Graefe and K. Ward, "Dynamic Query Evaluation Plans," Proc. ACM SIGMOD, 1989.

[4] 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.

[5] 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.

[6] Y.E. Ioannidis, R.T. Ng, K. Shim, and T.K. Sellis, "Parametric Query Optimization," Proc. 18th Int'l Conf. Very Large Data Bases (VLDB), 1992.

[7] Microsoft Corp., "Plan Forcing Scenario: Create a Plan Guide That Uses a USE PLAN Query Hint," SQL Server 2005 Books Online, 2005.

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

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

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

AUTHORS LIST:

SWATHI SAMBANGI received her Bachelor's Degree in Information Technology in GMR Institute of Technology Affiliated to JNTU Kakinada and pursuing Masters of Technology in Software Engineering in Gokul Institute of Technology and Sciences affiliated to JNTU Kakinada. She is presently working as Assistant

Professor in Vishaka Institute of Engineering and Technology, Visakhapatnam. She is a reviewer of IJCSIS Journal and her research areas of interest are Software Engineering, Computer Networks and Data Mining.

NAGARAM PHANI KUMAR received B.E (C.S.E) from JNTU-Kakinada, M.Tech (IT) from Vignan University. Presently he is working as Asst Professor at RAJA MAHENDRA COLLEGE OF ENGINEERING, IBRAHIMPATNAM, RANGA REDDY DIST. Andhra Pradesh, India. He is having 6 months of teaching experience in the field of Computer Science & Engineering.

KIRAN KATTA received B.E (C.S.E) from Acharya Nagarjuna University, M.Tech (CSE) from JNTU-Kakinada. Presently he is working as Asst Professor at RAJA MAHENDRA COLLEGE OF ENGINEERING, IBRAHIMPATNAM, RANGA REDDY DIST. Andhra Pradesh, India. He is having 5+ years of teaching experience in the field of Computer Science & Engineering.

RAJEEV GANDHI BAKKA received B.E (C.S.E) from Acharya Nagarjuna University, M.Tech (CSE) from Acharya Nagarjuna University. Presently he is working as Asst Professor at MALLA REDDY GROUP OF INSTITUTIONS (CM ENGINEERING COLLEGE), DHULAPULLY, RANGA REDDY DIST. Andhra Pradesh, India. He is having 5+ years of teaching experience in the field of Computer Science & Engineering.

DEEPIKA RANI KAMPALLY received Bachelor's degree in Computer science and Engineering from JNTUH, Pursuing M.Tech in Computer Science and Engineering from JNTUH. She is a research scholar in field of Information Security. She is having an experience of 3.8 Years in the field of Computer Science and Engineering, presently working as Assistant Professor in the department of CSE.