Cost Estimation of User-Defined Methods in Object-Relational Database Systems

Jihad Boulos*, Kinji Ono
NACSIS (National Center for Science Information Systems)
Otsuka 3-29-1,
Bunkyo-Ku
Tokyo 112
Japan
{boulos,ono}@rd.nacsis.ac.jp

Abstract: In this paper we present a novel technique for cost estimation of user-defined methods in advanced database systems. This technique is based on multi-dimensional histograms. We explain how the system collects statistics on the method that a database user defines and adds to the system. From these statistics a multi-dimensional histogram is built. Afterwards, this histogram can be used for estimating the cost of the target method whenever this method is referenced in a query. This cost estimation is needed by the optimizer of the database system since this cost estimation needs to know the cost of a method in order to place it at its optimal position in the Query Execution Plan (QEP). We explain here how our technique works and we provide an example to better verify its functionality.

Keywords: Advanced Database Systems, User-defined Methods, Cost Estimation, Optimization.

1. Introduction

The extensions introduced in advanced Database Management Systems (DBMSs) in the last few years has made it possible for a user to add new types and predicates –that may also be called user-defined methods or functions– to the system and to reference one or more of these predicates in a query. This is the case for both Object-Relational (OR) and Object-Oriented (OO) database systems. The dynamic process of System R in optimizing relational queries is being increasingly adopted in optimizing new types of queries in OR and OO systems. This optimization process is mainly concerned with ordering joins in a query while using the heuristic of pushing selections down in the query tree, *i.e.*, applying selections as soon as possible. As pointed out earlier in Hellerstein and Stonebraker (1993) this heuristic is not valid any more when referencing user-defined methods as predicates. In addition, ordering expensive predicates on a relation or collection becomes non-trivial.

Both Hellerstein and Stonebraker (1993) and Hellerstein (1994) on one hand and Chaudhuri and Shim (1997) on the other hand address the problem of optimizing queries with expensive predicates. However, both studies assume the cost of any user-defined method is known *a-priori*; they assume that the user who adds a method provides also its cost according to its input data. This assumption is a limiting factor for the applicability of the proposed optimization schemes since estimating the cost of a user-defined method requires a highly skilled user that can analyze his method and estimate its cost according to the metrics used by the database system. Such expertise is not always available to normal users. Moreover, the method that a user wants to add may be a third party executable (*i.e.*, a black box) that the user does not have access to its internal mechanisms and cannot easily analyze and estimate its execution cost. An automatic process for user-defined method cost estimation is hence crucially needed in order to alleviate this limitation of the applicability of optimization processes for queries with user-defined methods. This paper is a first step to address this issue.

We had already studied an approach for automatic cost estimation of user-defined methods in Boulos,

* Currently with The Boeing Company, M&CT - Phantom Works. Jihad.F.Boulos@boeing.com

Viémont, and Ono (1997). This approach is based on a curve-fitting like mechanism and uses neural networks. Under that approach a neural network is trained according to some measurements made on the target user-defined method. The neural network is feed with different values/sizes of the data entries of the method and the execution time for each measured value and size. In a later phase the network is used to estimate the execution cost of the method with other entries (*i.e.*, other values/sizes). The main limitation of this approach is its applicability: it is not trivial to integrate neural networks in an already very complex system like a DBMS. Since histograms have proven their effectiveness in selectivity estimation (Poosala et. al. (1996)) and are widely used in commercial systems, we study here their effectiveness in capturing the execution costs of user-defined methods and estimating in a second phase these costs.

The paper is organized as follows: Section 2 presents the motivation for our work. In Section 3 we give a formal definition to the problem we are addressing with some approaches we think may be applicable to resolve it. In Section 4 we present our solution that is based on multi-dimensional histograms with an algorithm to build them. Section 5 presents an example where we applied our proposed technique. In Section 6 we discuss some related works and in Section 7 we present a conclusion and some directions for future works.

2. Motivation

Our work is motivated by a new generation of applications that we think are going to extensively use the new functionality in ORDBMSs. Multimedia and web-based applications may be the most direct targets that needs to use the capability in OR systems of user-defined and expensive predicates. Algorithms such as compression, text search, time-series manipulation and analysis, similarity search (DNA sequences, fingerprints, images, etc.), audio and video manipulations are being aggressively investigated and added as new functionality in database systems. These algorithms (*i.e.*, methods) are sometimes added by the commercial database vendors (*e.g.*, DataBlades modules in Informix and Cartridges in Oracle) but they can also be added by application developers. An intelligent database system must be able to automatically collect statistics on these algorithms, estimate their costs and selectivity, and place them at their optimal positions in QEPs whenever they are referenced by SQL queries. We are addressing here cost estimation only and not selectivity estimation. Future complex queries may contain several of these expensive methods, interleaved with joins and simple predicates.

Examples of simple queries that contain expensive and user-defined methods are given here. Such type of queries may benefit from our proposed technique.

```
select * from Map
where Contained(shape,
            Circle(POINT, RADIUS))
 and shape.Area() > AREA;
// can also be expressed as
// Greater(Area(shape), AREA);

select Extract(Roads,
            SatelliteImage)
from Map
where contained(SatelliteImage,
      Circle(POINT, RADIUS))
 and SnowCoverage(SatelliteImage)
            < PERCENTAGE;

select name, location
from document
where contains(text, STRING)
 and SimilarityDistance(image,
       SHAPE) < DISTANCE;

select distinct point.name,
      point.location
from point, polygon
where polygon.landuse = LANDUSE
 and overlaps(polygon.shape,
            point.location);

select *
from suspects
where addeddate > DATE
 and zip like '75%'
 and (hasbarbe(face)
      or similar(face, IMAGE));

select Company.Name
from Company
where addeddate > DATE
 and zip like '98%'
 and Similarity(MovingAvg(Period,
            StockPrice), SHAPE)
      < THREESHOLD;
```

3. Problem Formulation

We give here a better definition for the problem we are addressing, so it would be easier for the reader to understand our goals. An *n*-ary method is of the form *MethName*(Arg_1, …, Arg_n). The execution cost of this method depends upon its internal processing complexity and the type and size of each of its arguments Arg_1, …, Arg_n. Formally

$$Cost_{MethName} = InitCost_{MethName} + \sum_{i=1}^{n} Cost_{Arg_i}$$
$$+ \sum_{i=1}^{n} InfluencingSize_{Arg_i} \times perbytecpu$$
$$+ \sum_{i=1}^{n} \left(InfluencingVal_{Arg_i} \times perincvalcpu \right)$$

$InitCost_{MethName}$ is the cost to initiate the execution of the method independent of any argument. $\sum_{i=1}^{n} Cost_{Arg_i}$ is necessary because Arg_i may be by itself another user-defined method. An example of this situation is when a query references something like Similarity(MovingAvg(Period, StockPrice), SHAPE). The third factor in the equation is meant to compute the differential cost whenever the size of an argument influences the cost of the method. Similarly, the fourth factor is meant to compute the differential cost whenever the value of an argument influences the cost of the method. Other factors may also play a role in increasing or decreasing the cost of some methods; such factors may be the access cost to some stored data in order for the method to proceed. We do not discuss these factors here.

A user-defined method may be a stored procedure within the DBMS written in a general-purpose language (such as C) or in a fourth generation language such as SQL or OQL; it may be also an external executable called dynamically from the DBMS whenever it is being referenced in a query. We explain in the next subsection different types of methods that have different behavior in their costs in relevance to their input arguments.

3.1 Method Costs Classification

It should be noted that several types of variability in cost exist depending upon the values and sizes of the input arguments to a method. This is mainly related to the sensitivity of the method's cost to its input arguments. Some arguments may have a great influence on the method's cost while others may not have any effect. The variability in method's cost is also related to the complexity of the method's internal processing. We divided the types of methods depending upon their cost variability in relevance to their input arguments into three classes.

Constant Costs: each method falling into this class has a constant execution cost that is independent of the values and sizes of its input arguments. The costs of such methods are hence dependent only upon their internal processing complexity. Some simple examples of such methods are Add(Arg_1, Arg_2), ComputeAge(DateOfBirth), MultipleMatrix(10x10, 10x10).

Monotone Variable Costs: methods falling into this class have variable costs dependent upon the values and sizes of at least one of their input arguments. Typically, the cost of a method here goes up with the values and sizes of its input –but this is not necessarily the case always. Examples from this class are: Search(Text, STRING), MovingAvg(Period, StockPrice).

Non-monotone Variable Costs: methods from this class have a higher level of complexity in their internal processing in relevance to their input arguments and hence have ups and downs in their costs depending upon complex relations between their input arguments. Examples of this class are: Overlaps(Land, Road) (for this method, an execution may immediately reveal that an input land and an input road do not overlap when comparing their bounding rectangles from an R-tree index. However, if the bounding rectangles overlap, the execution must proceed to compare pixels of the two spatial objects, which is a much expensive operation. In addition, the cost of this method is highly sensitive to the size of the input arguments). Another example from this class would be GreaterSnowCoverage(SatImage, PERCENTAGE).

Estimating the costs of methods from the first class is a simple task. It is sufficient to execute a method from this class once, record its execution cost, and use this execution cost whenever needed. The second class of methods has a higher complexity to estimate its cost. However, we think the estimation approach we are proposing in the next section is quite suitable for this class and will have an acceptable estimation error rate most of the time. The third class is even harder to estimate the costs of its methods; our proposed technique may apply to this class but for some methods, it may have a high estimation error rate. Further investigations must be carried on this class.

3.2 Possible Cost Estimation Approaches

We believe that there is a high similarity in estimating the costs of user-defined methods and the (eternal) selectivity estimation problem in database systems. Therefore we think that the same approaches that have been investigated and applied to selectivity estimation may apply to method cost estimation. These different approaches may be divided into four categories: 1) parametric functions, 2) histograms, 3) sampling and 4) curve fitting. As we mentioned earlier in this paper, we have already investigated the appropriateness of neural networks for method cost estimation. This was a curve fitting approach. A histogram approach for method cost estimation may be easier to be integrated in database systems, since histograms are the only approach that is effectively used in commercial systems.

There is however a major difference between selectivity estimation and method cost estimation. The former has only one argument for its input and hence is only two-dimensional (one dimension for the values of the target attribute –*i.e.*, the argument– and the second dimension for the selectivity). The later has multiple dimensions; these are the input arguments to the method in addition to the cost dimension. Hence, each *n*-ary method must have $n+1$ dimensions in its histogram to capture its input arguments and provide a cost estimation. We explain in the next section how to build a multi-dimensional histogram from statistics collected while executing a user-defined method.

4. Histogram Approach

To build a multi-dimensional histogram for estimating the cost of a user-defined method, the system must first execute the method several times with different values of its input arguments, collect these values and the different costs, and then build the histogram. To do so, a measurement campaign must be carried. The user in this regard may help the system in providing what he thinks might be the upper and lower values for each of the input arguments. The system then carries the measurement campaign, varying the values of each argument between its upper and lower values. A pseudo-algorithm to carry the measurement phase is given in Figure 1.

Procedure CollectStats(*MethName*,
n,
Pointer to $Arg_1,\ldots, Arg_n$) **{**
If (n == 1) **do {**
Perform a measurement campaign varying the value of Arg_1
return;
}
for Arg_1 **in** $Arg_1,\ldots, Arg_n$ **do {**
Select different values for Arg_1 to be measured
Foreach value of Arg_1 **do**
CollectStats(*MethName*,
n - 1,
Pointer to $Arg_2,\ldots, Arg_n$);
}
}

Figure 1: Algorithm for the measurement phase and statistics collection.

When the first phase of statistics collection is finished, the system will have a multi-dimensional array of all the collected measurements. This array may be very large if the target method has several arguments in its input and for each argument the measurement campaign tested several values. The number of cells in this array is $NbTestedValues_{Arg_1} \times NbTestedValues_{Arg_2} \times \cdots \times NbTestedValues_{Arg_n}$.
Hence, it becomes non-practical to store this large array in the database system. Two size reduction techniques may be applied to alleviate this burden.

The first technique to reduce the size of the large array is to replace it by a multi-dimensional histogram where several measured values for an argument are replaced by only one value. This value is equivalent to one bucket in histograms for selectivity estimation. Hence, a multi-dimensional histogram is built to reduce the size of the array to a much smaller size.

The second technique is meant to further reduce the number of buckets on some dimensions. This is because the number of buckets grows exponentially with the number of arguments in a defined method.

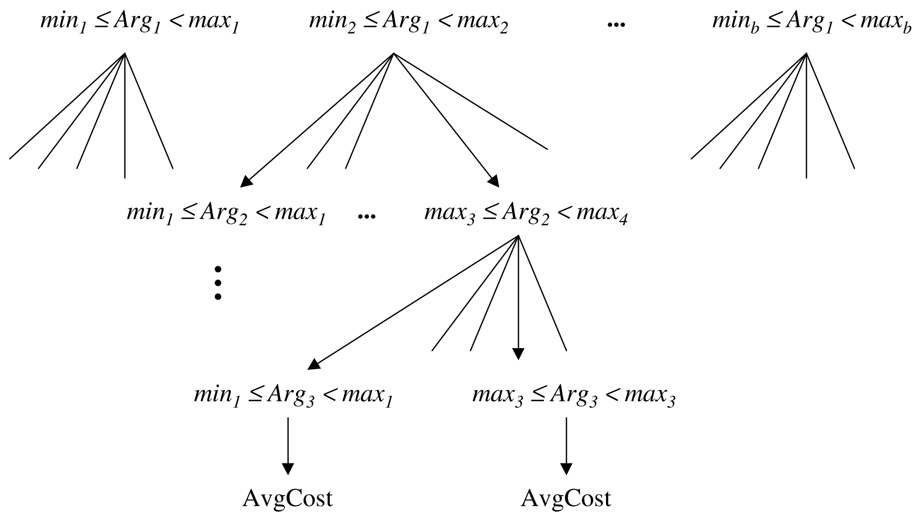

Figure 3: The multi-dimensional histogram is stored like a tree and is traversed top-down to get the estimated cost for the target method for a specific set of values of the method arguments.

That is, the number of buckets is B^n. In order to alleviate this burden the number of buckets for each argument may be adapted to the influence this argument has on the cost of the method. In this way, a high influencing argument will have a higher number of buckets than a low influencing argument. We only give in Figure 2 a pseudo-algorithm to construct the multi-dimensional histogram. Reduction in the number of buckets for some dimensions is left for further discussions in future works.

```
Procedure ConstHisto(n,
        Pointer to ArrayOfMeasurement) {
 Sum = Sum of all values in ArrayOfMeasurement;
 ForFirst dimension in ArrayOfMeasurement do {
  Divide the dimension into B buckets such that each
bucket has approximately Sum/B summing value;
  If (n == 1) return;
  Foreach bucket from the previous division do
   ConstHisto(n – 1, bucket);
 }
}
```

Figure 2: Algorithm for constructing the multi-dimensional histogram.

The constructed multi-dimensional histogram for a method will practically be stored as a tree in a database system. This tree is traversed like an index whenever needed to get the estimated cost for a method with a specific set of values for the method's input arguments. Figure 3 gives a better visualization for this concept.

5. Experimental Results

In order to experimentally validate our proposed technique we built a program in C to construct both equi-width and equi-height multi-dimensional histograms. We run a manual measurement campaign on a text search engine in which we varied the sizes of both the text in which the search is performed and the pattern we are searching for. The search had the form Search(Text, String) where Text pointed to a file and String pointed to a string in memory. We varied the size of the Text file from 5 to 50 MB by a step of 5 MB and the size of the string from 4 words to 28 words by a step of 4 words.

The measurement campaign generated a two-dimensional array with 70 cells that contained each the measured execution cost for two specific values of Text and String. To store this array, the system needs to store 140 numbers (the different values need 70 numbers to be stored and there are 70 other numbers for the cost).

We applied the program we coded to construct the multi-dimensional histograms on the measured data. The program generated an equi-height and another equi-width histograms. The reduction in needed number of points from the measurement array to the histogram went from 140 numbers needed to be stored to 34 numbers –*i.e.*, a reduction of factor 4. Figure 4 visualizes the three graphs that were constructed from all the measurement numbers and

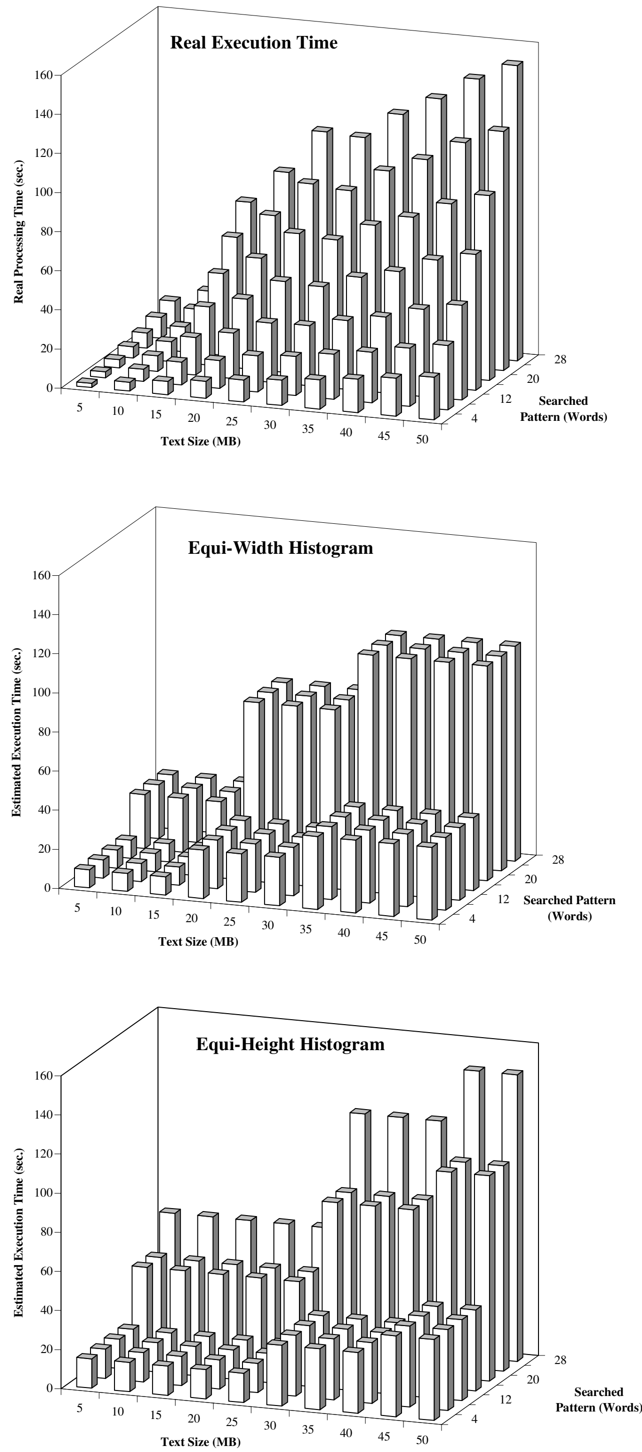

Figure 4: A comparison between graphs for the real, equi-width, and equi-height histograms for the text search method that has been experimented.

the two generated histograms. The graphs give a much better idea of the purpose of our approach. Any method with a higher number of arguments cannot be visualized because it would need more than 3 dimensions.

The error rates for both the equi-height and the equi-width are reported in Table 1. These error rates may seem high but the reader can see from Figure 4 that these error rates are still acceptable for the benefit they yield in reducing the number of points that must be stored. In this text search example the equi-width histogram seems to have a slight advantage over the equi-height histograms; however, this may be due to the approximately perfect increases in the method's execution costs in relevance to the input sizes. These increases have a logarithmic shape for the Text sizes and an exponential shape with the String size. Further experiment are necessary to elect the most suitable multi-dimensional histograms for most types of methods.

	Avg. Absolute Error	Avg. Relative Error
Equi-Height Histogram	14.7	78.4%
Equi-Width Histogram	13.9	54.9%

Table 1: Comparison of average error rates for both multi-dimensional histograms.

6. Related Work

To the best of our knowledge, there has been little work on estimating the cost of expensive predicates. Most research works in this area concentrated on sequencing several expensive predicates and/or interleaving them with joins. As we have said, Hellerstein and Stonebraker (1993) and Hellerstein (1994) address these two issues and provide some heuristics to sequencing and interleaving expensive predicates in QEPs. Chaudhuri and Shim (1997) also address the problem of ordering and interleaving expensive methods with joins. Both studies assume the cost of any expensive predicate known *a–priori*.

Recently, Shivakumar, Chekuri, and Garcia-Molina (1998) considered selecting and applying approximate predicates to be applied on data before the real expensive predicates. In this manner less expensive approximate predicates filter out non-qualified input before applying the expensive predicates. Here also, the cost of any approximate or full expensive predicate is assumed to be known in advance.

7. Conclusion

We presented in this paper a novel technique for estimating the costs of user-defined methods. This technique is based on multi-dimensional histograms. The cost estimation of user-defined methods is needed by the optimizer of an OR or OO database system in order to correctly place the methods at their optimal positions in a QEPs. Several issues remain to be resolved before a beneficial integration of this technique into a DBMS. These are mainly the selectivity estimation of expensive predicates and the cost of methods with values of arguments given outside the measurements and histogram boundaries. This paper presented a step toward a better understanding and optimization of user-defined methods in advanced database systems.

References

Boulos, J., Viemont, Y., and Ono, K. 1997. A Neural Networks Approach for Query Cost Evaluation. *Transaction of Information Processing Society of Japan*. Vol. **38**, No. 12, (1997) 2566–2575.

Chaudhuri, S., and Shim, K. 1997. Optimization of Queries with User-defined Predicates. *Proceedings of the 22nd VLDB Conference*, Mumbai, India.

Hellerstein, J., and Stonebraker, M. 1993. Predicate Migration: Optimizing Queries with Expensive Predicates. *Proceedings of the 1993 ACM-SIGMOD Int. Conference on Management of Data*, Washington, DC.

Hellerstein, J. 1994. Practical Predicate Placement. *Proceedings of the 1994 ACM-SIGMOD Int. Conference on Management of Data*, Minneapolis, Minnesota.

Poosala, V., Ioannidis, Y., Haas, P., and Shekita, E. 1996. Improved Histograms for Selectivity Estimation of Range Predicates. *Proceedings of the 1996 ACM-SIGMOD Int. Conference on Management of Data*. Montreal, Canada.

Shivakumar, N., Chekuri, C., and Garcia-Molina, H.1998. Filtering Expensive Predicates. *Stanford Report*.