

Self-tuning UDF Cost Modeling Using the Memory-Limited Quadtree

Zhen He Byung S. Lee Robert R. Snapp

Department of Computer Science
University of Vermont
Burlington, VT 05405
{zhenhe,bslee,snapp}@emba.uvm.edu

Abstract. Query optimizers in object-relational database management systems require users to provide the execution cost models of user-defined functions(UDFs). Despite this need, however, there has been little work done to provide such a model. Furthermore, none of the existing work is self-tuning and, therefore, cannot adapt to changing UDF execution patterns. This paper addresses this problem by introducing a self-tuning cost modeling approach based on the quadtree. The quadtree has the inherent desirable properties to (1) perform fast retrievals, (2) allow for fast incremental updates (without storing individual data points), and (3) store information at different resolutions. We take advantage of these properties of the quadtree and add the following in order to make the quadtree useful for UDF cost modeling: the abilities to (1) adapt to changing UDF execution patterns and (2) use limited memory. To this end, we have developed a novel technique we call the *memory-limited quadtree(MLQ)*. In MLQ, each instance of UDF execution is mapped to a query point in a multi-dimensional space. Then, a prediction is made at the query point, and the actual value at the point is inserted as a new data point. The quadtree is then used to store summary information of the data points at different resolutions based on the distribution of the data points. This information is used to make predictions, guide the insertion of new data points, and guide the compression of the quadtree when the memory limit is reached. We have conducted extensive performance evaluations comparing MLQ with the existing (static) approach.

1 Introduction

A new generation of object-relational database applications, including multimedia and web-based applications, often make extensive use of user-defined functions(UDFs) within the database. Incorporating those UDFs into ORDBMSs entails query optimizers should consider the UDF execution costs when generating query execution plans. In particular, when UDFs are used in the ‘where’ clause of SQL select statements, the traditional heuristic of evaluating predicates as early as possible is no longer valid [1]. Moreover, when faced with multiple UDFs in the ‘where’ clause, the order in which the UDF predicates are evaluated can make a significant difference to the execution time of the query.

Consider the following examples taken from [2, 3].

```

select Extract(roads, m.SatelliteImg) from Map m
where Contained(m.satelliteImg, Circle(point, radius))
   and SnowCoverage(m.satelliteImg) < 20%;

select d.name, d.location from Document d
where Contains(d.text, string)
   and SimilarityDistance(d.image, shape) < 10;

select p.name, p.street_address, p.zip from Person p, Sales s
where HighCreditRating(p.ss_no) and p.age in [30,40]
   and Zone(p.zip) = 'bay area' and p.name = s.buyer_name
group by p.name, p.street_address, p.zip
having sum(s.amount) > 1000;

```

In the above examples, the decision as to which UDF (e.g., `Contained()`, `SimilarityDistance()`) to execute first or whether a join should be performed before UDF execution depends on the cost of the UDFs and the selectivity of the UDF predicates. This paper is concerned with the former.

Although cost modeling of UDFs is important to the performance of query optimization, only two existing papers address this issue [3, 4]. The other existing works are centered on the generation of optimal query execution plans for query optimizers catering for UDFs [1, 2, 5]. They assume UDF execution cost models are provided by the UDF developer. This assumption is naive since functions can often have complex relationships between input arguments and execution costs. In this regard, this paper aims to develop *automated* means of predicting the execution costs of UDFs in an ORDBMS.

No existing approach[4, 3] for automatically modeling the costs of UDFs is *self-tuning*. One existing approach is the static histogram(SH)-based cost modeling approach[3]. The other approach uses curve-fitting based on neural networks[4]. Both approaches require users to train the model a-priori with previously collected data. Approaches that do not self-tune degrade in prediction accuracy as the pattern of UDF execution varies greatly from the pattern used to train the model. In contrast, we use a self-tuning query feedback-driven approach similar to that used in [6, 7] for selectivity estimation of range queries and in [8] for relational database query optimization.

Figure 1 shows how self-tuning cost modeling works. When a query arrives, the query optimizer generates a query execution plan using the UDF cost estimator as one of its components. The cost estimator makes its prediction using the cost model. The query is then executed by the execution engine according to the query plan. When the query is executed, the actual cost of executing the UDF is used to update the cost model. This allows our approach to adapt to changing UDF execution patterns.

In this paper, we describe a *quadtree*-based approach to the cost modeling of UDFs. The quadtree is widely used in digital image processing and computer graphics for modeling spatial segmentation of images and surfaces[9–11], in the spatial database environment for the indexing and retrieval of spatial objects [12, 13], in the on-line analytic processing context to answer aggregate queries (e.g., SUM, COUNT, MIN, MAX, AVG)[14], and so on.

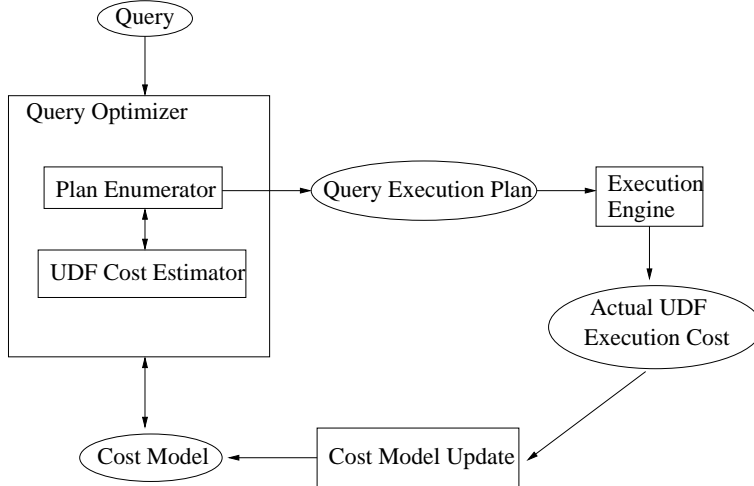


Fig. 1. Query feedback-driven UDF cost modeling.

The quadtree has the following inherent desirable properties. It (1) performs retrievals fast, (2) allows for fast incremental updates (without the need to store individual data points), and (3) stores information at different resolutions. The first and second properties are of particular importance to UDF cost modeling since the final goal of the cost modeling is to improve the query execution speed. Naturally, the overhead introduced by the cost estimator needs to be very low.

We take advantage of these properties of the quadtree and add the following in order to make the quadtree more useful for UDF cost modeling: the abilities to (1) adapt to changing UDF execution patterns and (2) use a limited amount of memory. The first property is important since UDF execution patterns may change over time. The second property is important since the cost estimator is likely to be allocated only a small portion of the memory allocated to the query optimizer for metadata. Moreover, the query optimizer needs to keep two cost estimators for each UDF in order to model both CPU and disk IO costs. In this regard, we call our approach the *memory-limited quadtree (MLQ)*.

In MLQ, each instance of UDF execution is mapped to a data point in a multi-dimensional space. The summary information of data points is stored in the nodes of a quadtree. The summary information consists of sum, count, and sum of squares for data points in the indexed data region. We store the summary information at every level of the quadtree, with coarser-grained information (over a larger region) at a higher level. This information is then used to make predictions and to guide the compression of the quadtree when the memory limit is reached.

Since MLQ is based on the quadtree which partitions the *entire* multi-dimensional space, it can start making predictions immediately after the first data point is inserted (with no a-priori training data). The prediction accuracy improves as more data points are inserted. Alternatively, MLQ can be trained with some a-priori training data before making the first prediction. This improves its initial prediction accuracy.

The key contributions of this paper are in (1) proposing a self-tuning UDF cost modeling approach that adapts to changing UDF execution patterns, (2) proposing a dynamic quadtree-based summary structure that works with limited memory, (3) conducting an extensive performance evaluation of MLQ against the existing static SH algorithm. To our knowledge, SH is the only existing UDF cost modeling algorithm feasibly usable in an ORDBMS.

The remainder of this paper is organized as follows. In Section 2 we outline related work. In Section 3 we formally define the problem. We then describe our MLQ approach to solving the problem in Section 4. In Section 5 we detail the experiments conducted to evaluate the performance of MLQ. Last, in Section 6 we conclude the paper.

2 Related Work

In this section we discuss existing work in two related areas: UDF cost modeling and self-tuning approaches to query optimization.

2.1 UDF cost modeling

As already mentioned, the SH approach in [3] is designed for UDF cost modeling in ORDBMSs. It is not self-tuning in the sense that it is trained a-priori with existing data and do not adapt to new query distributions. In SH, users define a set of variables used to train the cost model. The UDF is then executed using these variable values to build a multi-dimensional histogram. The histogram is then used to predict the cost of future UDF executions.

Specifically, two different histogram construction methods are used in [3], equi-width and equi-height. In the equi-width histogram method, each dimension is divided into N intervals of equal length. Then, N^d buckets are created, where d is the number of dimensions. The equi-height histogram method divides each dimension into intervals so that the same number of data points are kept in each interval. In order to improve storage efficiency, they propose reducing the number of intervals assigned to variables that have low influence on the cost. However, they do not specify how to find the amount of influence a variable has. It is left as future work.

In [4] Boulos proposes a curve-fitting approach based on neural networks. Their approach is not self-tuning either and, therefore, does not adapt to changing query distributions. Moreover, neural networks techniques are complex to implement and very slow to train[15], therefore inappropriate for query optimization in ORDBMSs[3]. This is the reason we do not compare MLQ with this neural networks approach.

2.2 Self-tuning approaches to query optimization

Histogram-based techniques have been used extensively in selectivity estimation of range queries in the query optimizers of relational databases [6, 7, 16]. STGrid[6] and STHoles[7] are two recent techniques that use a query feed-back-driven, self-tuning, multi-dimensional histogram-based approach. The idea behind both STGrid and STHoles is to spend more modeling resources in areas where there is more workload activity. This is similar to our aim of adapting to

changing query distributions. However, there is a fundamental difference. Their feedback information is the actual *number of tuples* selected for a range query whereas our feedback information is the actual *cost values* of individual UDF executions. This difference presents a number of problems when trying to apply their approach to solve our problem. For example, STHoles creates a “hole” in a histogram bucket for the region defined by a range query. This notion of a region does not make sense for a point query used in UDF cost modeling.

DB2’s LLearning Optimizer(LEO) offers a comprehensive way of repairing incorrect statistics and cardinality estimates of a query execution plan by using feedback information from recent query executions. It is general and can be applied to any operation – including the UDFs – in a query execution plan. It works by logging the following information of past query executions: execution plan, estimated statistics, and actual observed statistics. Then, in the background, it compares the difference between the estimated statistics and the actual statistics and stores the difference in an adjustment table. Then, it looks up the adjustment table during query execution and apply necessary adjustments. MLQ is more storage efficient than LEO since it uses a quadtree to store summary information of UDF executions and applies the feedback information directly on the statistics stored in the quadtree.

3 Problem Formulation

In this section we formally define UDF cost modeling and define our problem.

UDF cost modeling

Let $f(a_1, a_2, \dots, a_n)$ be a UDF that can be executed within an ORDBMS with a set of input arguments a_1, a_2, \dots, a_n . We assume the input arguments are ordinal and their ranges are given, while leaving it to future work to incorporate nominal arguments and ordinal arguments with unknown ranges. Let $T(a_1, a_2, \dots, a_n)$ be a transformation function that maps some or all of a_1, a_2, \dots, a_n to a set of ‘cost variables’ c_1, c_2, \dots, c_k , where $k \leq n$. The transformation T is *optional*. T allows the users to use their knowledge of the relationship between input arguments and the execution costs ec_{IO} (e.g., the number of disk pages fetched) and ec_{CPU} (e.g., CPU time) to produce cost variables that can be used in the model more efficiently than the input arguments themselves. An example of such a transformation is for a UDF that has the input arguments `start_time` and `end_time` which are mapped to the cost variable `elapsed_time` as `elapsed_time = end_time - start_time`.

Let us define *model variables* m_1, m_2, \dots, m_k as either input arguments a_1, a_2, \dots, a_n or cost variables c_1, c_2, \dots, c_k depending on whether the transformation T exists or not. Then, we define *cost modeling* as the process for finding the relationship between the model variables m_1, m_2, \dots, m_k and ec_{IO}, ec_{CPU} for a given UDF $f(a_1, a_2, \dots, a_n)$. In this regard, a cost model provides a mapping from a k -dimensional data space defined by the k model variables to a 2-dimensional space defined by ec_{IO} and ec_{CPU} . Each point in the data space has the model variables as its coordinates.

Problem definition

Our goal is to provide a self-tuning technique for UDF cost modeling with a strict memory limit and the following performance considerations: prediction accuracy, average prediction cost (APC), and average model update costs (AUC). The AUC includes insertion costs and compression costs.

APC is defined as:

$$APC = \frac{\sum_{i=0}^{N_P-1} P(i)}{N_P} \quad (1)$$

where $P(i)$ is the time it takes to make the i^{th} prediction using the model and N_P is the total number of predictions made.

AUC is defined as:

$$AUC = \frac{\sum_{i=0}^{N_I-1} I(i) + \sum_{i=0}^{N_C-1} C(i)}{N_P} \quad (2)$$

where $I(i)$ is the time it takes to insert the i^{th} data point into the model and N_I is the total number of insertions, and $C(i)$ is the time it takes for the i^{th} compression and N_C is the total number of compressions.

4 The Memory-Limited Quadtree

Section 4.1 describes the data structure of the memory-limited quadtree, Section 4.2 describes the properties that an optimal quadtree has in our problem setting, and Sections 4.3 and 4.4 elaborate on MLQ cost prediction and model update, respectively.

4.1 Data structure

MLQ uses the conventional quadtree as its data structure to store summary information of past UDF executions. The quadtree fully partitions the multi-dimensional space by recursively partitioning it into 2^d equal sized blocks (or, partitions), where d is the number of dimensions. In the quadtree structure, a child node is allocated for each non-empty block and its parent has a pointer to it. Empty blocks are represented by null pointers. Figure 2 illustrates different node types of the quadtree using a two dimensional example. We call a node that has exactly 2^d children a *full node*, and a node with fewer than 2^d children a *non-full node*. Note that a leaf node is a non-full node.

Each node —internal or leaf— of the quadtree stores the summary information of the data points stored in a block represented by the node. The summary information for a block b consists of the sum $S(b)$, the count $C(b)$, the sum of squares $SS(b)$ of the values of the data points that map into the block. There is little overhead in updating these summary values incrementally as new data points are added. At prediction time, MLQ uses these summary values to compute the average value as follows.

$$AVG(b) = \frac{S(b)}{C(b)} \quad (3)$$

During data point insertion and model compression, the summary information stored in quadtree nodes are used to compute the sum of squared errors ($SSE(b)$) as follows.

$$\begin{aligned}
 SSE(b) &= \sum_{i=0}^{C(b)} (V_i - AVG(b))^2 \\
 &= SS(b) - C(b)(AVG(b))^2
 \end{aligned} \tag{4}$$

where V_i is the value of the i^{th} data point that maps into the block b .

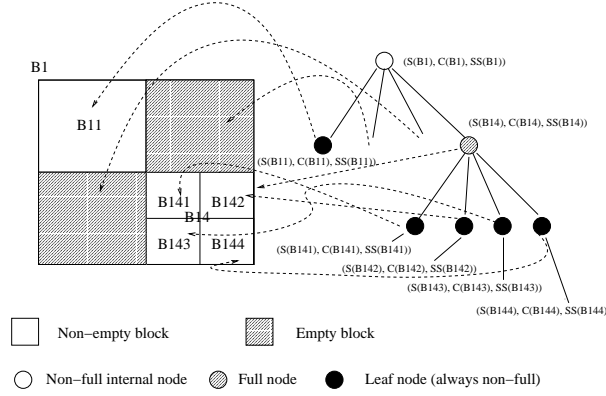


Fig. 2. The quadtree data structure.

4.2 Optimal quadtree

Given the problem definition in Section 3, we now define the optimality criterion of the quadtree used in MLQ. Let M_{max} denote the maximum memory available for use by the quadtree and DS denote a set of data points for training. Then, using M_{max} and DS , we now define $QT(M_{max}, DS)$ as the set of all possible quadtrees that can model DS using no more than M_{max} .

Let us define $SSENC$ as the sum of squared errors of the data points in block b excluding those in its children. That is,

$$SSENC(b) = \sum_{i=1}^{C(b_{nc})} (V_i - AVG(b))^2 \tag{5}$$

where b_{nc} is the set of data points in b that do not map into any of its children and V_i is the value of the i^{th} data point in b_{nc} .

$SSENC(b)$ is a measure of the expected error for making a prediction using a non-full block b . This is a well-accepted error metric used for the compression of a data array[17]. It is used in [17] to define the optimal quadtree for the purpose of building the optimal *static* two-dimensional quadtree. We can use it for our purpose of building the optimal *dynamic* multi-dimensional quadtree, where the number of dimensions can be more than two.

Then, we define the optimal quadtree as one that minimizes the *total SSEN*C (*TSSENC*) defined as follows.

$$TSSENC(qt) = \sum_{b \in NFB(qt)} (SSENC(b)) \quad (6)$$

where qt is the quadtree such that $qt \in QT(M_{max}, DS)$ and $NFB(qt)$ is defined as the set of the blocks of non-full nodes of qt . We use $TSSENC(qt)$ to guide the compression of the quadtree qt so that the resultant quadtree has the smallest increase in the expected prediction error. Further details of this will appear in Section 4.4.

4.3 Cost prediction

The quadtree data structure allows cost prediction to be fast, simple, and straightforward. Figure 3 shows MLQ’s prediction algorithm. The parameter β allows MLQ to be tuned based on the expected level of noise in the cost data. (We define noise as the magnitude by which the cost fluctuates at the same data point coordinate.) This is particularly useful for UDF cost modeling since disk IO costs (which is affected by many factors related to the database buffer cache) fluctuate more sharply at the same coordinates than CPU costs do. A larger value of β allows for averaging over more data points when a higher level of noise is expected.

Predict_Cost (QT: quadtree, QP: query point, β : minimum number of points)

1. Find the lowest level node of QT such that QP maps into the block b of the node and the count in the node $\geq \beta$.
2. Return sum/count ($= S(b)/C(b)$) from the node found.

Fig. 3. Cost prediction algorithm of MLQ.

4.4 Model update

Model update in MLQ consists of data point insertion and compression. In this subsection we first describe how the quadtree is updated when a new data point is inserted and, then, describe the compression algorithm.

Data point insertion: When a new data point is inserted into the quadtree, MLQ updates the summary information in each of the existing blocks that the new data point maps into. It then decides whether the quadtree should be partitioned further in order to store the summary information for the new data point at a higher resolution. An approach that partitions more eagerly will lead to higher prediction accuracy but more frequent compressions since the memory limit will be reached earlier. Thus, there exists a trade-off between the prediction accuracy and the compression overhead.

In MLQ, we let the user choose what is more important by proposing two alternative insertion strategies: eager and lazy. In the eager strategy, the quadtree is partitioned to a maximum depth (λ) during the insertion of every new data point. In contrast, the lazy strategy delays partitioning by partitioning a block

```

Insert_point ( DP: data point, QT: quadtree,  $th_{SSE}$ : SSE threshold,  $\lambda$ : maximum
depth )
1. cn = the current node being processed, its initialized to be the root node of QT.
2. update sum, count, and sum of squares stored in cn.
3. while ( ( $SSE(cn) \geq th_{SSE}$ ) and ( $\lambda$  has not been reached) ) or
4.   (cn is not a leaf node) {
5..   if DP does not map into any existing child of cn {
6.     create the child in cn that DP maps into.
7.     initialize sum, count, and sum of squares of the created child to zero.
8.   }
9.   cn = child of cn that DP maps into.
10.  update sum, count, and sum of squares of cn.
11. }

```

Fig. 4. Insertion algorithm of MLQ.

only when its SSE reaches a threshold (th_{SSE}). This has the effect of delaying the time of reaching the memory limit and, consequently, reducing the frequency of compression.

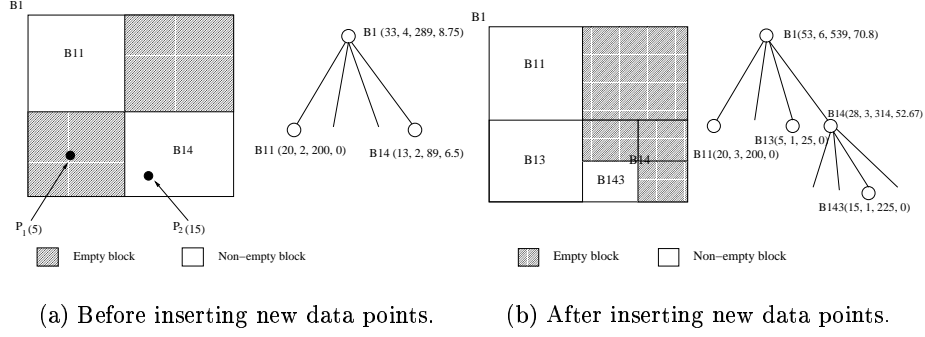
The th_{SSE} , used in the lazy insertion strategy, is defined as follows.

$$th_{SSE} = \alpha SSE(r) \quad (7)$$

where r is the root block and the parameter α is a *scaling factor* that helps users to set the th_{SSE} . The SSE in the root node indicates the degree of cost variations in the *entire* data space. In this regard, th_{SSE} can be determined relative to $SSE(r)$. If α is smaller, new data points are stored in a block at a higher depth and, as a result, prediction accuracy is higher. At the same time, however, the quadtree size is larger and, consequently, the memory limit is reached earlier, thus causing more frequent compressions. Thus, the α parameter is another mechanism for adjusting the trade-off between the prediction accuracy and the compression overhead.

Figure 4 shows the insertion algorithm. The same algorithm is used for both eager and lazy strategies. The only difference is that in the eager approach the th_{SSE} is set to zero whereas, in the lazy approach, it is set using Equation 7 (after the first compression). The algorithm traverses the quadtree top down while updating the summary information stored in every node it passes. If the child node that the data point maps into does not exist, a new child node is created (line 6-7). The traversal ends when the maximum depth λ is reached or the currently processed node is a leaf node with the SSE greater than the th_{SSE} .

Figure 5 illustrates how the quadtree is changed as two new data points P_1 and P_2 are inserted. In this example, we are using lazy insertion with the th_{SSE} of 8 and λ of 5. When P_1 is inserted, a new node is created for the block B13. Then, B13's summary information in the node is initialized to 5 for sum, 1 for the count, 25 for the sum of squares, and 0 for SSE. B13 is not further partitioned since its SSE is less than the th_{SSE} . Next, when P_2 is inserted, B14 is partitioned since its updated SSE of 67 becomes greater than the th_{SSE} .



(a) Before inserting new data points. (b) After inserting new data points.

$\mathbf{P}(v)$, $\mathbf{B}(s,c,ss,sse)$: v = value, s = sum, c = count, ss = sum of squares, sse = sum of squared errors

Fig. 5. An example of data point lazy insertion in MLQ.

Model compression: As mentioned in the Introduction, compression is triggered when the memory limit is reached. Let us first give an intuitive description of MLQ’s compression algorithm. It aims to minimize the expected loss in prediction accuracy after compression. This is done by incrementally removing quadtree nodes in a bottom up fashion. The nodes that are more likely to be removed have the following properties: a low probability of future access and an average cost similar to its parent. Removing these nodes is least likely to degrade the future prediction accuracy.

Formally, the goal of compression is to free up memory by deleting a set of nodes such that the increase in $TSSENC$ (see definition in Equation 6) is minimized and a certain factor (γ) of the memory allocated for cost modeling is freed. γ allows the user to control the trade-off between compression frequency and prediction accuracy.

In order to achieve the goal, all leaf nodes are placed into a priority queue based on the *sum of squared error gain (SSEG)* of each node. The *SSEG* of block b is defined as follows.

$$SSEG(b) = SSENC(p_{ac}) - (SSENC(b) + SSENC(p_{bc})) \quad (8)$$

where p_{bc} refers to the state of the parent block of b before the removal of b and p_{ac} refers to that after the removal of b . $SSEG(b)$ is a measure of the increase in the $TSSENC$ of the quadtree after block b is removed. Here, leaf nodes are removed before internal nodes to make the algorithm incremental since removing an internal node automatically removes all its children nodes as well.

Equation 8 can be simplified to the following equation. (Due to space constraints, we omit the details of the derivation and ask the readers to refer to [18].)

$$SSEG(b) = C(b)(AVG(p) - AVG(b))^2 \quad (9)$$

where p is the parent block of b . Equation 9 has three desirable properties. First, it favors the removal of leaf nodes that have fewer data points (i.e. smaller $C(b)$). This is desirable since a leaf node with fewer data points has a lower probability

of being accessed in the future under the assumption that frequently queried regions are more likely to be queried again. Second, it favors the removal of leaf nodes that show a smaller difference between the average cost for the node and that for its parent. This is desirable since there is little value in keeping a leaf node that returns a predicted value similar to that from its parent. Third, computation of $SSEG(b)$ is efficient as it can be done using the sum and count values already stored in the quadtree nodes.

Figure 6 shows the compression algorithm. First, all leaf nodes are placed into the priority query PQ based on the SSEG value (line 1). Then, the algorithm iterates through PQ while removing the nodes from the top, that is, from the node with the smallest SSEG first (line 2 - 10). If the removal of a leaf node results in its parent’s becoming a leaf node, then the parent node is inserted into PQ (line 5 - 7). The algorithm stops removing nodes when either PQ becomes empty or at least γ fraction of memory has been freed.

```

Compress_tree (QT: quadtree,  $\gamma$ : minimum amount memory to be freed,
              total_mem: the total amount of memory allocated)
1. Traverse QT and place every leaf node into a priority queue PQ with the
   node with the smallest SSEG at its top.
2. while (PQ is not empty) and (memory_freed / total_mem <  $\gamma$ ) {
3.   remove the top element from PQ and put it in current_leaf
4.   parent_node = the parent of current_leaf.
5.   if (parent_node is not the root node) and (parent_node is now a leaf node) {
6.     insert parent_node into PQ based on its SSEG.
7.   }
8.   deallocate memory used by current_leaf
9.   memory_freed = memory_freed + size of current_leaf
10. }

```

Fig. 6. Compression algorithm of MLQ.

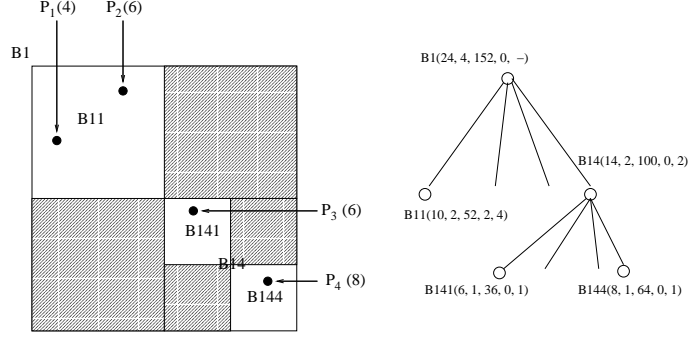
Figure 7 illustrates how MLQ performs compression. Figure 7(a) shows the state of the quadtree before the compression. Either B141 or B144 can be removed first since they both have the lowest SSEG value of 1. The tie is arbitrarily broken, resulting in, for example, the removal of B141 first and B144 next. We can see that removing both B141 and B144 results in an increase of only 2 in the $TSSENC$. If we removed B11 instead of B141 and B144, we would increase the $TSSENC$ by 2 after removing only one node.

5 Experimental Evaluation

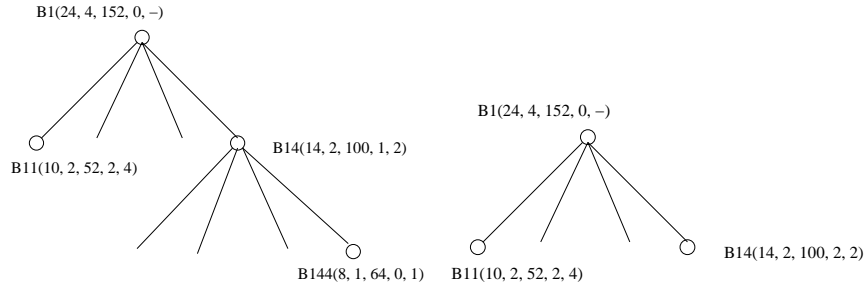
In this section we describe the experimental setup used to evaluate MLQ against existing algorithms and present the results of experiments.

5.1 Experimental setup

Modeling methods and model training methods: We compare the performance of two MLQ variants against two variants of SH: (1) **MLQ-E**, our method using eager insertions, (2) **MLQ-L**, our method using lazy insertions,



(a) Before compression.



(b) After removing block B141.

(c) After removing block B144.

$\mathbf{P}(e)$, $\mathbf{B}(s,c,ss,ssenc,sseg)$: e = execution cost, s = sum, c = count, ss = sum of square, $ssenc$ = sum of squared error of data points not associated with any of its children, $sseg$ = sum of squared error gain.

Fig. 7. An example of MLQ compression.

(3) **SH-H**[3] using equi-height histograms, and (4) **SH-W**[3] using equi-width histograms.

In these methods, models are trained differently depending on whether the method is self-tuning or not. The two MLQ methods, which are self-tuning, start with no data point and train the model incrementally (i.e., one data point at a time) while the model is being used to make predictions. In contrast, the two SH methods, which are not self-tuning, train the model a-priori with a set of queries that has the same distribution as the set of queries used for testing.

We limit the amount of memory allocated in each method to 1.8 Kbytes. This is similar to the amount of memory allocated in existing work[7, 16, 19] for selectivity estimation of range queries. All experiments allocate the same amount of memory in all methods. We have extensively tuned MLQ to achieve its best performance and used the resulting parameters values. In the case of the SH methods, there are no tuning parameters except the number of buckets used, which is determined by the memory size. The following is a specification of the MLQ parameters used in this paper: $\beta = 1$ for CPU cost experiments and 10 for

disk IO cost experiments, $\alpha = 0.05$, $\gamma = 0.1\%$, and $\lambda = 6$. We show the effect of varying the MLQ parameters in [18] due to space constraints.

Synthetic UDFs/datasets: We generate synthetic UDFs/datasets in two steps. In the first step, we randomly generate a number (N) of *peaks* (i.e. extreme points within confined regions) in the multi-dimensional space. The coordinates of the peaks have the uniform distribution, and the heights (i.e. execution costs) of the peak have the Zipf distribution[20]. In the second step, we assign a randomly selected *decay function* to each peak. Here, a decay function specifies how the execution cost decreases as a function of the Euclidean distance from the peak. The decay functions we use are uniform, linear, Gaussian, log of base 2, and quadratic. They are defined so that the maximum point is at the peak and the height decreases to zero at a certain distance (D) from the peak. This suite of decay functions reflect the various computational complexities common to UDFs.

This setup allows us to vary the complexity of the data distribution by varying N and D . As N and D increase, we see more overlaps among the resulting decay regions (i.e., regions covered by the decay functions).

The following is a specification of the parameters we have used: the number of dimensions d set to 4, the range of values in each dimension set to 0 - 1000, the maximum cost of 10000 at the highest peak, the Zipf parameter (z) value of 1 for the Zipf distribution, a standard deviation of 0.2 for the Gaussian decay function, and the distance D equal to 10% of the Euclidean distance between two extreme corners of the multi-dimensional space.

Real UDFs/datasets: Two different kinds of real UDFs are used: three keyword-based text search functions (simple, threshold, proximity) and three spatial search functions (K-nearest neighbors, window, range). All six UDFs are implemented in Oracle PL/SQL using built-in Oracle Data Cartridge functions. The dataset used for the keyword-based text search functions is 36422 XML documents of news articles acquired from the Reuters. The dataset used for the spatial search functions is the maps of urban areas in all counties of Pennsylvania State [21]. We ask the readers to see [18] for a more detailed description.

Query distributions: Query points are generated using three different random distributions of their coordinates: (1) uniform, (2) Gaussian-random, and (3) Gaussian-sequential. In the uniform distribution, we generate query points uniformly in the entire multi-dimensional space. In the case of Gaussian-random, we first generate c Gaussian centroids using the uniform distribution. Then, we randomly choose one of the c centroids and generate one query point using the Gaussian distribution whose peak is at the chosen centroid. This is repeated n times to generate n query points. In the Gaussian-sequential case, we generate a centroid using the uniform distribution and generate n/c query points using the Gaussian distribution whose peak is at the centroid. This is repeated c times to generate n query points.

We use the Gaussian distribution to simulate skewed query distribution in contrast to the uniform query distribution. For this purpose, we set c to 3 and the standard deviation to 0.05. In addition, we set n to 5000 for the synthetic datasets and 2500 for the real datasets.

Error Metric: We use the *normalized absolute error* (NAE) to compare the prediction accuracy of different methods. Here, the NAE of a set of query points Q is defined as:

$$NAE(Q) = \frac{\sum_{\mathbf{q} \in Q} |PC(\mathbf{q}) - AC(\mathbf{q})|}{\sum_{\mathbf{q} \in Q} AC(\mathbf{q})} \quad (10)$$

where $PC(\mathbf{q})$ denotes the predicted cost and $AC(\mathbf{q})$ denotes the actual cost at a query point \mathbf{q} . This is similar to the normalized absolute error used in [7].

Note that we do not use the relative error because it is not robust to situations where the execution costs are low. We do not use the (unnormalized) absolute error either because it varies greatly across different UDFs/datasets while, in our experiments, we do compare errors across different UDFs/datasets.

Computing platform: In the experiments involving real datasets, we use Oracle 9i on SunOS 5.8, installed on Sun Ultra Enterprise 450 with four 300 MHz CPUs, 16 KB level 1 I-cache, 16 KB level 1 D-cache, and 2 MB of level 2 cache per processor, 1024 MB RAM, and 85 GB of hard disk. Oracle is configured to use a 16 MB data buffer cache with direct IO. In the experiments involving synthetic datasets, we use Red Hat Linux 8 installed on a single 2.00 GHz Intel Celeron laptop with 256 KB level 2 cache, 512 MB RAM, and 40 GB hard disk.

5.2 Experimental results

We have conducted four different sets of experiments (1) to compare the prediction accuracy of the algorithms for various query distributions and UDFs/datasets, (2) to compare the prediction, insertion, and compression costs of the algorithms, (3) to compare the effect of noise on the prediction accuracy of the algorithms, and (4) to compare the prediction accuracy of the MLQ algorithms as the number of query points processed increases.

Experiment 1 (prediction accuracy): Figure 9 shows the results of predicting the CPU costs of the real UDFs. The results for the disk IO costs will appear in Experiment 3. The results in Figure 9 show MLQ algorithms give lower error (or within 0.02 absolute error) when compared with SH-H in 10 out of 12 test cases. This demonstrates MLQ’s ability to retain high prediction accuracy while dynamically ‘learning’ and predicting UDF execution costs.

Figure 8 shows the results obtained using the synthetic UDFs/datasets. The results show MLQ-E performs the same as or better than SH in all cases. However, the margin between MLQ-E and SH algorithms is smaller than that for the real UDFs/datasets. This is because the costs in the synthetic UDFs/datasets fluctuate less steeply than those in the real UDFs/datasets. Naturally, this causes the difference in the prediction errors of all the different methods to be smaller.

Experiment 2 (modeling costs): In this experiment we compare the modeling costs (prediction, insertion, and compression cost) of the cost modeling algorithms. This experiment is not applicable to SH due to its static nature and, therefore, we compare only among the MLQ algorithms. Figure 10(a) shows the results from the real UDFs/datasets. It shows the breakdown of the modeling

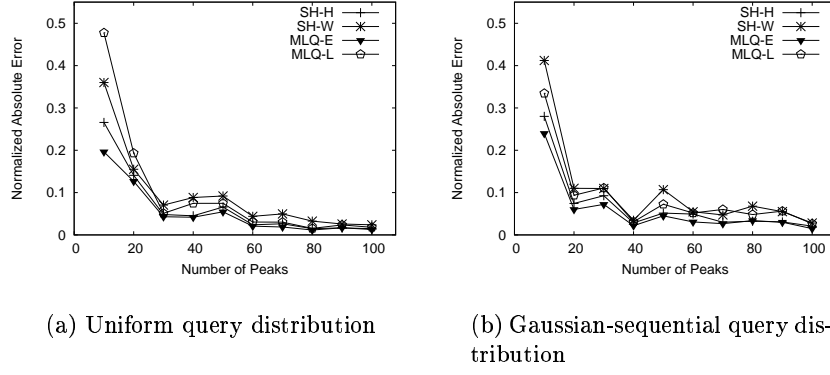


Fig. 8. Prediction accuracy for a varying number of peaks (for synthetic data).

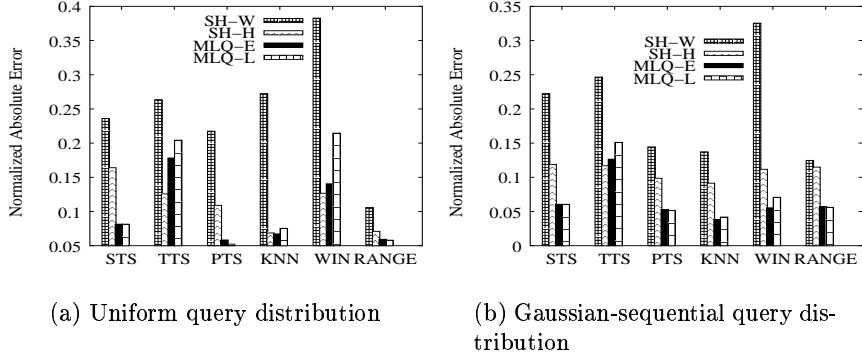


Fig. 9. Prediction accuracy for various real UDFs/datasets.

costs into the prediction cost(PC), insertion cost(IC), compression cost(CC), and model update cost(MUC). MUC is the sum of IC and CC. All costs are normalized against the total UDF execution cost. Due to space constraints, we show only the results for WIN. The other UDFs show similar trends. The prediction costs of both MLQ-E and MLQ-L are only around 0.02% of the total UDF execution cost. In terms of the model update costs, even MLQ-E, which is slower than MLQ-L, imposes only between 0.04% and 1.2% overhead. MLQ-L outperforms MLQ-E for model update since MLQ-L delays the time the memory limit is reached and, as a result, performs compression less frequently.

Figure 10(b) shows the results from the synthetic UDFs/datasets. The results show similar trends as the real UDFs/datasets, namely MLQ-L outperforms MLQ-E for model update.

Experiment 3 (noise effect on prediction accuracy): As mentioned in Section 4.3, the database buffer caching has a noise-like effect on the disk IO cost. In this experiment, we compare the accuracy of the algorithms at predicting the disk IO cost while introducing noise.

Figure 11(a) shows the results for the real UDFs/datasets. The results show MLQ-E outperforms MLQ-L. This is because MLQ-E does not delay partitioning and, thus, stores data at a higher resolution earlier than MLQ-L, thereby allowing

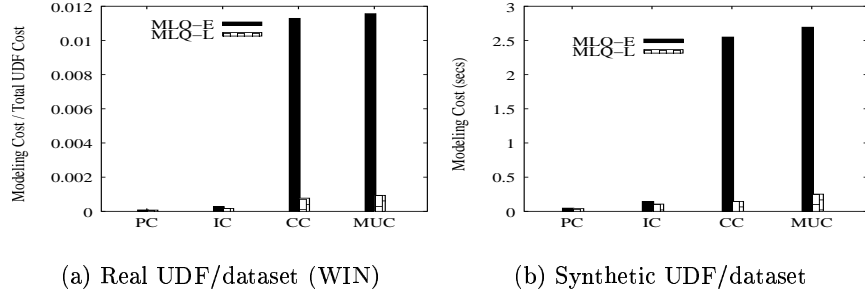


Fig. 10. Modeling costs (using uniform query distribution).

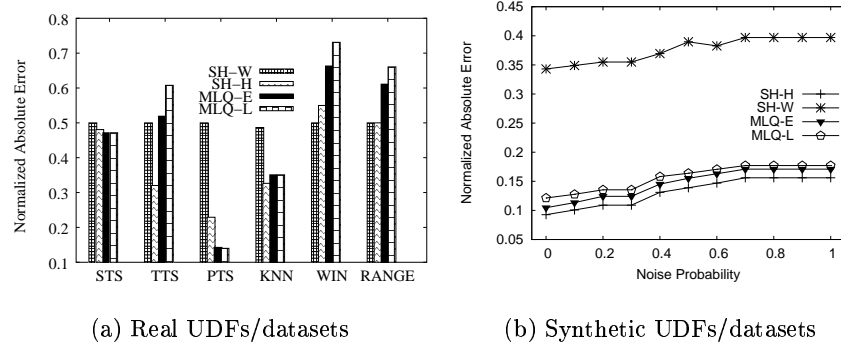


Fig. 11. Prediction accuracy for varying noise effect (using uniform query distribution).

prediction to be made using the summary information of *closer* data points. MLQ-E performs within around 0.1 normalized absolute error from SH-H in five out of the six cases. This is a good result, considering that SH-H is expected to perform better because it can absorb more noise by averaging over more data points and is trained a-prior with a complete set of UDF execution costs.

For the synthetic UDFs/datasets, we simulate the noise by varying *noise probability*, that is, the probability that a query point returns a random value instead of the true value. Due to space constraints, we omit the details of how noise is simulated and refer the readers to [18]. Figure 11(b) shows the results for the synthetic UDFs/datasets. The results show SH-H outperforms the MLQ algorithms by about 0.7 normalized absolute error irrespective of the amount of noise simulated.

Experiment 4 (prediction accuracy for an increasing number of query points processed): In this experiment we observe how fast the prediction error decreases as the number of query points processed increases in the MLQ algorithms. This experiment is not applicable to SH because it is not dynamic.

Figure 12 shows the results obtained using the same set of UDFs/datasets and query distribution as in Experiment 2. In all the results, MLQ-L reaches its minimum prediction error much earlier than MLQ-E. This is because of MLQ-L’s strategy of delaying the node partitioning limits the resolution of the summary

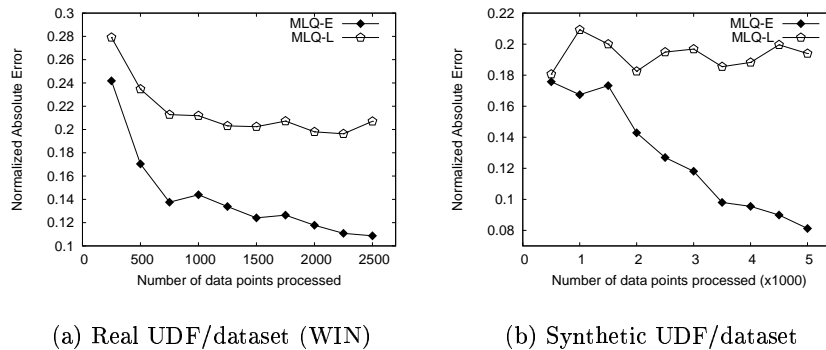


Fig. 12. Prediction error with an increasing number of data points processed (uniform query distribution).

information in the quadtree and, as a result, causes the highest possible accuracy to be reached faster.

6 Conclusions

In this paper we have presented a memory-limited quadtree-based approach (called MLQ) to self-tuning cost modeling with a focus on the prediction accuracy and the costs for prediction and model updates. MLQ stores and manages summary information in the blocks (or partitions) of a dynamic multi-resolution quadtree while limiting its memory usage to a predefined amount. Predictions are made using the summary information stored in the quadtree, and the actual costs are inserted as the values of new data points. MLQ offers two alternative insertion strategies: eager and lazy. Each strategy has its own merits. The eager strategy is more accurate in most cases but incurs higher compression cost (up to 50 times). When the memory limit is reached, the tree is compressed in such a way as to minimize the increase in the total expected error in subsequent predictions.

We have performed extensive experimental evaluations using both real and synthetic UDFs/datasets. The results show that the MLQ method gives higher or similar prediction accuracy compared with the SH method despite that the SH method is not self-tuning and, thus, trains the model using a complete set of training data collected a-priori. The results also show that the overhead for being self-tuning is negligible compared with the execution cost of the real UDFs.

Acknowledgments

We thank Li Chen, Songtao Jiang, and David Van Horn for setting up the real UDFs/datasets used in the experiments, and the Reuters Limited for providing Reuters Corpus, Volume 1, English Language, for use in the experiments. This research has been supported by the US Department of Energy through Grant No. DE-FG02-ER45962.

References

1. Hellerstein, J., Stonebraker, M.: Predicate migration: Optimizing queries with expensive predicates. In: Proc. of ACM-SIGMOD. (1993) 267–276
2. Chaudhuri, S., Shim, K.: Optimization of queries with user-defined predicates. In: Proc. of ACM SIGMOD. (1996) 87–98
3. Jihad, B., Kinji, O.: Cost estimation of user-defined methods in object-relational database systems. SIGMOD Record (1999) 22–28
4. Boulos, J., Viemont, Y., Ono, K.: A neural network approach for query cost evaluation. Trans. on Information Processing Society of Japan (1997) 2566–2575
5. Hellerstein, J.: Practical predicate placement. In: Proc. of ACM SIGMOD. (1994) 325–335
6. Abounaga, A., Chaudhuri, S.: Self-tuning histograms: building histograms without looking at data. In: Proc. of ACM SIGMOD. (1999) 181–192
7. Bruno, N., Chaudhuri, S., Gravano, L.: STHoles: A multidimensional workload-aware histogram. In: Proc. of ACM SIGMOD. (2001) 211–222
8. Stillger, M., Lohman, G., Markl, V., Kandil, M.: LEO - DB2's LEarning optimizer. In: Proc. of VLDB. (2001) 19–28
9. Hunter, G.M., Steiglitz, K.: Operations on images using quadtrees. IEEE Trans. on Pattern Analysis and Machine Intelligence **1** (1979) 145–153
10. Strobach, P.: Quadtree-structured linear prediction models for image sequence processing. IEEE Trans. on Pattern Analysis and Machine Intelligence **11** (742-748)
11. Lee, J.W.: Joint optimization of block size and quantization for quadtree-based motion estimation. IEEE Trans. on Pattern Analysis **7** (1998) 909–911
12. Aref, W.G., Samet, H.: Efficient window block retrieval in quadtree-based spatial databases. GeoInformatica **1** (1997) 59–91
13. Wang, F.: Relational-linear quadtree approach for two-dimensional spatial representation and manipulation. IEEE Trans. on Knowledge and Data Eng. **3** (1991) 118–122
14. Lazaridis, I., Mehrotra, S.: Progressive approximate aggregate queries with a multi-resolution tree structure. In: Proc. of ACM SIGMOD. (2001) 401–413
15. Han, J., Kamber, M.: 7. In: Data Mining: Concepts and Techniques. Morgan Kaufmann (2001) 303, 314–315
16. Poosala, V., Ioannidis, Y.: Selectivity estimation without the attribute value independence assumption. In: Proc. of VLDB. (1997) 486–495
17. Buccafurri, F., Furfaro, F., Sacca, D., Sirangelo, C.: A quad-tree based multiresolution approach for two-dimensional summary data. In: Proc. of SSDBM, Cambridge, Massachusetts, USA (2003)
18. He, Z., Lee, B.S., Snapp, R.R.: Self-tuning UDF cost modeling using the memory limited quadtree. Technical Report CS-03-18, Department of Computer Science, University of Vermont (2003)
19. Deshpande, A., Garofalakis, M., Rastogi, R.: Independence is good: Dependency-based histogram synopses for high-dimensional data. In: Proc. of ACM SIGMOD. (2001) 199–210
20. Zipf, G.K.: Human behavior and the principle of least effort. Addison-Wesley (1949)
21. PSADA: Urban areas of pennsylvania state. URL:<http://www.pasda.psu.edu/access/urban.shtml> (Last viewed:6-18-2003)