Self-Tuning Cost Modeling of User-Defined Functions in an Object-Relational DBMS

Zhen He[†], Byung Suk Lee[‡], Robert Snapp[‡]
[†]Department of Computer Science, La Trobe University, Bundoora VIC 3086, Australia[1]
[‡]Department of Computer Science, University of Vermont, Burlington VT 05405, USA
E-mails: z.he@latrobe.edu.au, Byung.Lee@uvm.edu, snapp@cs.uvm.edu

Query optimizers in object-relational database management systems typically 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. The existing approaches are static in that they require users to train the model a-priori with pre-generated UDF execution cost data. Static approaches can not adapt to changing UDF execution patterns and thus degrade in accuracy when the UDF executions used for generating training data do not reflect the patterns of those performed during operation. This paper proposes a new approach based on the recent trend of self-tuning DBMS, by which the cost model is maintained dynamically and incrementally as UDFs are being executed online. In the context of UDF cost modeling, our approach faces a number of challenges, that is, it should work with limited memory, work with limited computation time, and adjust to the fluctuations in the execution costs (e.g., caching effect). In this paper we first provide a set of guidelines for developing techniques that meet these challenges while achieving accurate and fast cost prediction with small overheads. Then, we present two concrete techniques developed under the guidelines. One is an instance-based technique based on the conventional k-nearest neighbor (KNN) technique which uses a multi-dimensional index like the R*-tree. The other is a summary-based technique which uses the quadtree to store summary values at multiple resolutions. We have performed extensive performance evaluations comparing these two techniques against existing histogram-based techniques and the KNN technique, using both real and synthetic UDFs/data sets. The results show our techniques provide better performance in most situations considered.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*Query Processing*

General Terms: cost modeling, object relational DBMS, query optimization, self-tuning

Additional Key Words and Phrases: K-nearest neighbors, quadtree, self-tuning

1. INTRODUCTION

1.1 Motivation

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. Algorithms for compression, text search, time-series manipulation and analysis, similarity search (e.g., DNA sequences, fingerprints, images), and audio and video manipulations are being aggressively investigated and added as new UDFs in database systems. These UDFs are typically created by application developers as stored procedures in an object-relational database management system (ORDBMS).

Incorporating UDFs into ORDBMSs entails query optimizers should consider the UDF execution costs (or "costs" in short) 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

[1]This work was partially done while the author was at the Department of Computer Science, University of Vermont.

longer valid [Hellerstein and Stonebraker 1993]. Moreover, when faced with multiple UDFs in the 'where' clause, the order in which the UDF predicates are evaluated can make a significant difference in the execution time of the query.

Consider the queries shown in Figure 1. The decision as to which UDF (e.g., Contains(), 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.

```
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;
```

(Sources: [Chaudhuri and Shim 1999; Boulos and Ono 1999])

Fig. 1. Example queries with UDFs.

There has been some work done on the generation of optimal query execution plans by query optimizers catering for UDFs [Chaudhuri and Shim 1999; Hellerstein 1998; 1994; Hellerstein and Stonebraker 1993]. Since a UDF is called per tuple, they introduce the notion of a "differential" (i.e., per-tuple) cost for join and selection (involving UDFs) and, then, estimate the per-tuple cost to be the same for every tuple in the same table [Hellerstein 1998]. They, however, do not discuss a method for generating the cost model of a UDF and, instead, assume the UDF cost models are *manually* provided by the UDF developer. This assumption is naive since functions often have complex relationships between input arguments and execution costs, which makes it difficult for UDF developers to develop their own models manually. We thus need an *automatic* means to develop the cost model of a UDF.

We have found only two existing papers addressing the automatic cost modeling problem [Boulos et al. 1997; Boulos and Ono 1999]. One uses a histogram-based approach [Boulos and Ono 1999], and the other uses a neural network-based curve-fitting approach [Boulos et al. 1997]. Both approaches are *static* in that they require users to train the model (i.e., histogram or neural network) a-priori with pre-generated UDF execution cost data. These static approaches rapidly degrade in their prediction accuracies when the patterns of UDF executions used for generating training data do not reflect the patterns of those performed during operation. We thus need a *dynamic* technique. For this purpose, we adopt the notion of *self-tuning* [Chaudhuri 1999] into our cost modeling.

1.2 Self-tuning cost modeling concepts

Figure 2 illustrates the self-tuning cost modeling process. 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 execution plan. When the query is executed, the actual cost of executing the UDF is used to update the cost model. This query feedback-based approach allows the cost model to adapt to changing UDF execution patterns. In this case, the query feedback information is the actual cost of executing the UDF. This self-tuning approach is similar to that used in [Aboulnaga and Chaudhuri 1999; Bruno et al. 2001] for selectivity estimation of range queries and in [Stillger et al. 2001] for relational database query optimization.

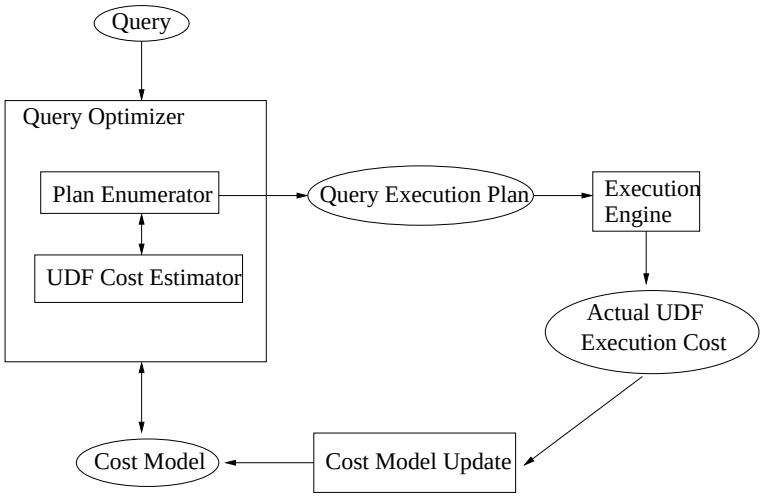


Fig. 2. Self-tuning UDF cost modeling.

The self-tuning modeling of UDF costs can be conceptualized as feedback-based incremental modeling of data values in a multidimensional model space. Specifically, each instance of UDF execution is mapped to a *query point* in a multi-dimensional space defined by *model variables*, i.e., variables identified or determined to influence the cost significantly. The data values predicted at a query point are the CPU cost and the disk IO cost. *Query feedback* is then provided based on the difference between the predicted cost and the actual cost. Then, some or all of the actual costs are inserted as *data points* into the model. This query-feedback-insertion cycle repeats to update the model *incrementally* as query points "arrive".

1.3 Challenges

Self-tuning UDF cost modeling described above bears the technical challenges of making accurate predictions despite limited memory, limited computation time, and cost fluctuations present in the system, as described below.

1.3.1 *Limited memory.* Only a limited portion of the memory used by the ORDBMS is available for query optimization and the related activities like UDF cost modeling, and even this limited memory is used to store the cost models of *multiple* UDFs. Therefore, the memory available for storing the cost models (i.e.,

CPU cost model, disk IO cost model) of a single UDF is very limited. However, the cost model in general grows in size as more data points from query feedback are inserted into it, thus eventually exhausting the allocated memory.

Once the memory runs out, the only way to insert more data points is to compress the cost model. This, however, causes the prediction accuracy to degrade. Thus, in order to achieve adequate prediction accuracy despite the memory limit, the frequency of compression should be kept small. This can be done by slowing down the frequency of inserting data points and achieving a high compression ratio (in order to free a large amount of memory) with the minimum loss of prediction accuracy.

1.3.2 *Limited computation time.* Query optimization should be done fast, i.e., within very limited computation time. (This would be more important in interactive query processing in which queries are compiled at execution time.) Since predicting the UDF cost using the model is only one small step in query optimization, it should be done even faster. In order to achieve this, we need to use an efficient data structure and access method for the model.

In addition, data point insertions and subsequent compressions should incur small overheads because, although they are not part of the query optimization itself, they consume computation time that would otherwise be available for query optimization. In order to keep the insertion overhead small, data points should be inserted conservatively (only inserting data points that are likely to significantly increase future prediction accuracy) into the model and the model data structure should support efficient *insertion* methods. In order to keep the compression overhead small, the frequency of data point insertion should be low and the compression ratio should be high.

1.3.3 *Cost fluctuations.* We have observed that UDF cost at the same query point in the model space fluctuates over time. The main causes are the caching effects in CPU caches (e.g., level 1 cache, level 2 cache) and disk I/O caches (e.g., database buffer cache, operating system buffer cache). The total caching effect amounts to "noise" in the data values, which refers to the *magnitude* by which the data value changes at a particular point in the model space.

Noisy data render the prediction unreliable. One way of handling this is the common statistical technique of averaging over a sufficient number of data points in a region. The optimal number of data points and the size of the region vary depending on the level of noise. In the case of UDF cost modeling, however, the noise level changes over time. Therefore, the prediction needs to automatically tune the number of data points and the region size according to the current level of noise.

1.4 Outline

In this paper, we first propose a set of guidelines for meeting the challenges described above. The guidelines, when applied, can produce different possible techniques by using different data structures for the cost model and different corresponding algorithms for querying, updating, and compressing the model.

We then present two concrete cost modeling techniques developed to embody the guidelines. One is called the *memory-limited K-nearest neighbors* (MLKNN), and the other is called the *memory-limited quadtree (MLQ)*[He et al. 2004]. MLKNN is an instance-based technique based on KNN, whereas MLQ is a summary-based technique using the quadtree. The traditional KNN is known to incur high computational and storage overheads for making predictions but achieve high prediction accuracy and efficient incremental model updates[Han and Kamber 2001]. MLKNN preserves these merits and overcome the limitations while meeting the challenges described in Section 1.3. For MLQ, the quadtree has the inherent desirable properties of fast retrievals (in response to queries), fast incremental updates (without storing individual data points), and multi-resolution model (stored at different resolutions). MLQ preserves these properties while meeting the challenges described in Section 1.3.

We have performed extensive performance evaluations, using both real and synthetic UDFs/datasets, to compare MLKNN and MLQ against the existing static histogram(SH) techniques[Boulos and Ono 1999] and KNN (which keeps all data points). We have made three key observations from the experiments. First, both MLKNN and MLQ give higher prediction accuracy than SH in most situations considered. The reason is both MLKNN and MLQ adapt to the query and data distributions and thus make more efficient use of the limited memory. Second, the modeling costs (i.e., the sum of the prediction, insertion, and compression costs) of both MLKNN and MLQ are within acceptable limits – specifically, less than 8% of the execution costs of the real UDFs. Third, MLKNN and MLQ are comparable with different strengths and weaknesses; For instance, overall, MLKNN achieves higher prediction accuracy while MLQ incurs smaller modeling costs.

1.5 Contributions

This paper makes the following main contributions. First, it proposes guidelines for self-tuning UDF cost modeling within a resource(i.e., memory and computation time)-limited environment with noisy data. Second, it presents two concrete techniques, MLKNN and MLQ, developed according to the guidelines. Third, it demonstrates the merits of the two techniques by comparing them against existing methods through extensive performance evaluations.

Although the focus of this paper is on the UDF cost modeling, the techniques proposed can be used in other application areas such as estimating program execution costs for job scheduling in parallel and/or distributed systems. It can also be used in other environments where resources are limited and dynamic value predictions are required at particular points in the data space.

1.6 Organization

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 guidelines for developing self-tuning cost modeling techniques in Section 4. In Section 5 we describe the MLKNN technique developed using the guidelines. In Section 6 we describe the MLQ method developed using the guidelines. In Section 7 we present the experiments conducted to evaluate the performances of MLKNN and MLQ against SH and KNN. Last, in Section 8 we conclude the paper and provide directions for further work.

2. RELATED WORK

We discuss related work in two areas: UDF cost modeling and self-tuning modeling.

2.1 UDF cost modeling

As already mentioned, the static histogram(SH) approach in [Boulos and Ono 1999] 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. Specifically, the UDF is executed for preset values of model variables to build a multi-dimensional histogram. The histogram is then used to predict the costs of future UDF executions.

Specifically, two different histogram construction methods are used in [Boulos and Ono 1999]: 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 with lower 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 [Boulos et al. 1997] Boulos et al. 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[Han and Kamber 2001], therefore inappropriate for query optimization in ORDBMSs [Boulos and Ono 1999]. For this reason, we do not compare our MLKNN and MLQ techniques with this approach in our experiments (Section 7).

2.2 Self-tuning modeling

Recently there have been efforts for building a self-tuning DBMS [Chaudhuri 1999], as exemplified by the automin project [Chaudhuri et al. 1999] at Microsoft Corporation. Self-tuning DBMSs are able to automatically tune themselves to application needs and hardware capabilities, thus significantly reducing the administration overhead. In this subsection we provide a brief survey of existing works that are using self-tuning techniques and distinguish them from our work.

Chaudhuri et al. in [Chaudhuri et al. 1999] discuss feedback-based self-tuning in the following four system issues: index selection for a given workload, memory management among concurrent queries, distribution statistics creation and updating, and dynamic storage allocation. Our work is also feedback-based, but it addresses a different system issue.

In [Lee et al. 2004], Lee et al. present a self-tuning technique for UDF cost modeling. It uses the same query feedback mechanism as that presented in this paper. The main difference is that multiple regression is used as the modeling technique. The regression coefficients are tuned incrementally based on the feedback from a batch of UDF executions. Limited memory is not a concern in their work because regression coefficients take very small amount of memory. However, although it has proven to be very feasible and achieving fairly accurate cost estimation for UDFs showing "smooth" cost variations, it is not generally applicable to UDFs with arbitrary cost variations.

There have been several papers presenting a self-tuning approach to estimating the selectivity of simple predicates (i.e., predicates on relational attributes) [Chen and Roussopoulos 1994; Aboulnaga and Chaudhuri 1999; Bruno et al. 2001]. As in our approach, their models are updated incrementally using query feedback. However, their works are for selectivity estimation instead of UDF cost estimation. Chen and Roussopoulos in [Chen and Roussopoulos 1994] use a curve fitting technique whereby a cumulative data distribution of the selection attribute value is updated based on query feedback. The selectivity is estimated from the distribution by computing the difference between the values at the two extreme points of the query range. Both STGrid[Aboulnaga and Chaudhuri 1999] and STHoles[Bruno et al. 2001] use multi-dimensional histograms as the modeling technique. STGrid uses a rectangular grid-based histogram that is dynamically split and merged based on the query feedback. STHoles improves on STGrid by allowing some buckets to be completely included inside others. In this way, the requirement that each bucket is rectangular is implicitly relaxed, and this results in buckets that more efficiently model complex regions of uniform tuple density. The idea behind both STGrid and STHoles is to allocate more memory in regions queried more frequently. This is similar to our aim of adapting to query distributions. However, there is a fundamental difference in that their feedback information is the actual number of tuples selected from the range query whereas our feedback information is the actual costs of the individual UDF executions. It is not obvious how to adapt their approaches[Aboulnaga and Chaudhuri 1999; Bruno et al. 2001] to work with UDF cost modeling.

In [Rahal et al. 2004], Rahal et al. present a technique that continuously updates the local query cost model in a dynamic multidatabase environment. They use multiple regression as the modeling technique. It periodically rebuilds the cost model after either one or a batch of query executions. (The batch approach is similar to the incremental update approach used in [Lee et al. 2004].) Their approach, however, is not self-tuning because there is no feedback loop that drives the update of a cost model. We may call it "self-managing" instead, as indicated in their paper.

In [Stillger et al. 2001], Stillger et al. presents a self-tuning approach to "repairing" an incorrect query execution plan. Each time a query is executed, the query execution plan used is analyzed based on the cost estimation errors to determine where in the plan the significant error occurred. The analysis results are

then used to adjust the data statistics and the estimation models of selectivity and cardinality. Unlike our approach which performs tuning at the level of a UDF, which is executed as only one step within a query, their approach performs tuning at the entire query level and, therefore, incurs higher overhead to collect the statistics need for the tuning. Moreover, different types of query predicates need separate tuning processes.

In [Lee et al. 2000], Lee et al. presents a self-tuning approach to data placement in a shared-nothing parallel database systems. If a load imbalance happens, it determines the amount of data to be moved from the overloaded node and integrate the moved data into selected destination nodes. Although called "self-tuning", this work is about dynamic resource allocations and is closer to a trigger-action mechanism.

3. PROBLEM FORMULATION

In this section, we formally define the general UDF cost modeling problem and the specific problem addressed in this paper.

3.1 UDF cost modeling

Let $f(a_1, a_2, ..., a_n)$ be a UDF that can be executed within an ORDBMS with a set of input arguments $a_1, a_2, ..., a_n$. Let $T(a_1, a_2, ..., a_n)$ be a transformation function that maps some or all of $a_1, a_2, ..., a_n$ to a set of "cost variables" $c_1, c_2, ..., c_k$, 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` calculated as `elapsed_time` = `end_time` − `start_time`.

Let us define *model variables* $m_1, m_2, .., m_k$ as either input arguments $a_1, a_2, ..., a_n$ or cost variables $c_1, c_2, ..., c_k$, 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 $m_1, m_2, ..., m_k$ and ec_{IO}, ec_{CPU} for a given UDF $f(a_1, a_2, ..., a_n)$. Each point in the model space has the model variables as its coordinates.

3.2 Problem definition

Let Q be a set of query points arriving in sequence. Then, given limited memory and computation time available and with cost fluctuating over time, the self-tuning UDF cost modeling technique aims to minimize the prediction error, prediction cost, and model update cost, measured considering all the queries in Q.

The prediction error is measured as the *normalized absolute error* (NAE) defined as

$$NAE(Q) = \frac{\sum_{\mathbf{q} \in Q} |PC(\mathbf{q}) - AC(\mathbf{q})|}{\sum_{\mathbf{q} \in Q} AC(\mathbf{q})} \tag{1}$$

where $PC(\mathbf{q})$ denotes the predicted (i.e., estimated) cost and $AC(\mathbf{q})$ denotes the actual cost at a query point $\mathbf{q} \in Q$. This metric is similar to the normalized absolute error used in [Bruno et al. 2001] for selectivity modeling[2].

The prediction cost and the model update cost are averaged over the queries in Q. Thus, the average prediction cost (APC) is measured as

$$APC = \frac{\sum_{\mathbf{q}} cost_{pred}(\mathbf{q})}{|Q|} \tag{2}$$

[2]We do not use the relative error because it is biased by a small number of query points with very low actual costs. We do not use the unnormalized absolute error either because it varies greatly across different UDFs/datasets, while in our experiments we do compare the errors across different UDFs/datasets.

where $cost_{pred}(\mathbf{q})$ is the cost of making a prediction at query point $\mathbf{q} \in Q$ and $|Q|$ denotes the cardinality of Q, and the average model update cost (AUC) is the summation of the average insertion cost (AIC) and the average compression cost (ACC). That is,

$$AUC = AIC + ACC \tag{3}$$

where AIC and ACC are defined as follows.

$$AIC = \frac{\sum_{\mathbf{d} \in D} cost_{ins}(\mathbf{d})}{|Q|} \tag{4}$$

where D is the set of data points inserted from the query points in Q and $cost_{ins}(\mathbf{d})$ is the cost of inserting a data point $\mathbf{d} \in D$.

$$ACC = \frac{\sum_{c \in C} cost_{comp}(c)}{|Q|} \tag{5}$$

where C is the set of compressions performed while predictions are made at the query points in Q and $cost_{comp}(c)$ is the cost of a compression $c \in C$.

Additionally, we define the *average modeling cost (AMC)* as the average cost of processing a set of queries – predictions and model updates (i.e., insertions, compressions). That is,

$$AMC = APC + AUC \tag{6}$$

4. GUIDELINES

The guidelines presented in this section are designed to address the problem defined in Section 3.2 while meeting the challenges described in Section 1.3. The guidelines are organized by the key operations: prediction, insertion, and compression.

4.1 Prediction guidelines

Since prediction is done using a model refined as a result of data point insertions, we can state this guideline in terms of the number of data points "used" for making the prediction.

Guideline P: *Use more data points for prediction in a region with higher noise level.*
This guideline is based on the common statistical technique for reducing the prediction error caused by noisy data, mentioned in Section 1.3.3. A prediction error consists of a variance error and a bias error [Bogartz 1994]. The former is caused by the variation of data value at one particular coordinate in the multi-dimensional space, and the latter is caused by the variation of data value across different coordinates. As the noise level increases, the variance error has more influence on the prediction error than the bias error. Therefore, more data points should be considered in order to reduce the prediction error in this case (even though it means using data points from a larger region). Conversely, as the noise level decreases, the bias error has more influence, and, therefore, data points in a smaller region should be used (which often leads to using fewer data points).

Figure 3(a) illustrates the guideline P. Suppose a prediction is to be made at a query point in the region 1-2. By Guideline P, the data points in the surrounding memory blocks A and C are used as well as those in B if the noise level in the region is significantly high, otherwise only block B is used.

4.2 Insertion (memory allocation) guidelines

The first insertion guideline has to do with the "pace" of inserting new data points.

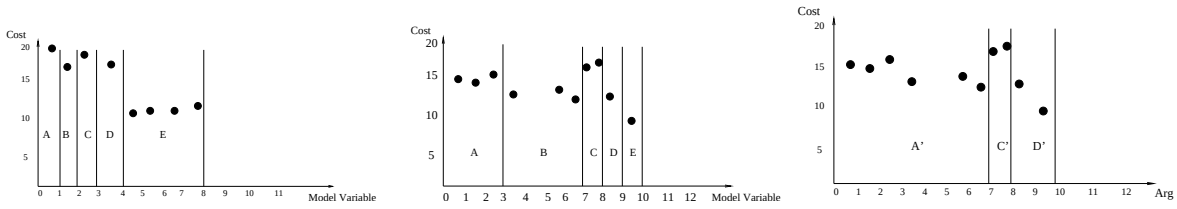
Guideline I: *Insert data points conservatively.*
A conservative insertion strategy contributes to reducing the frequency of data point insertion. This slows down the growth rate of the model and, consequently, reduces the frequency of compression as well. Thus, it leads to reducing both the average insertion cost (see Equation 4) and the average compression cost (see Equation 5).

Data point insertions lead to the refinement of a model one way or another, and this typically leads to consuming more memory. In this regard, the next two guidelines below are better stated in terms of memory allocation.

Guideline I1: *Allocate more memory to model regions with more complex cost variations.*
This allows for using a more refined model to predict values in those regions, thereby reducing the prediction errors.

Guideline I2: *Allocate more memory to model regions with more frequent queries.*
Assuming query patterns do not change rapidly, the regions recently queried frequently are likely to be queried more frequently in the near future. Thus, higher prediction accuracy can be achieved by refining the model in those regions; Refining the model calls for allocating more memory.

Figure 3(b) illustrates the guidelines I1 and I2 in light of the UDF cost modeling. (We use a one-dimensional model for ease of illustration, but it generalizes to any higher dimensional model.) Each point in the figure represents a previous query point. Some of the query points are inserted as data points, and others are not. Assume the resulting model is stored in five memory blocks (A, B, C, D, E) of the same size. Then, by the guidelines I1, more memory blocks are allocated to model the region 7-10 (blocks C, D, and E) than 0-7 (blocks A and B) because the UDF cost fluctuates more sharply in the region 7-10. Besides, by the guideline I2, no memory is allocated to model the region 10-12 because no query point has appeared there.



(a) Prediction. (b) Memory allocation. (c) Compression.

Fig. 3. Illustrations of the guidelines.

4.3 Compression (memory deallocation) guidelines

Since model compression is to deallocate memory allocated to the model as a result of data point insertion, its guidelines are contrasted with those of data point insertion.

Guideline C1: *Compress the model more aggressively in regions with less complex data distributions.*
This is because these regions can be modeled with similar accuracy as more complex regions without using as much memory.

Parameter	Description
K	the number of nearest neighbors used to make predictions.
T_{pe}	the prediction error threshold used to determine if data point should be inserted into the model.
MCR	the model compression ratio used as measure of how aggressively the model as a *whole* should be compressed.

Table I. Summary of parameters used in MLKNN.

Guideline C2: *Compress the model more aggressively in regions with less frequent queries.*
Assuming query patterns do not change rapidly, the regions recently queried less frequently are likely to be queried less frequently now. It is thus not an efficient use of computational time and memory to keep the models of these regions as refined as more frequently queried regions.

Figure 3(b)-(c) illustrates the guidelines C1 and C2. By the guideline C1, the memory blocks A and B of Figure 3(b) are compressed into the memory block A' because the UDF cost variation is smaller in the region 0-7 (blocks A and B) than the other regions. Besides, by the guideline C2, the memory blocks D and E of Figure 3(b) are compressed into the memory block D' because the number of query points in the region 8-9 (block D) and that in the region 9-10 (block E) are smaller (only one each) than those in the other regions. Furthermore, using a discerning strategy like the guidelines C1 and C2 contributes to reducing the frequency of compression compared with not using such a strategy, thereby contributing to reducing the average model compression cost (see Equation 5).

5. MEMORY-LIMITED K NEAREST NEIGHBORS (MLKNN)

As mentioned in Section 1.4, MLKNN is modified from KNN to be efficient in both computation and storage while preserving much of its merits – the prediction accuracy and cheap incremental training inherent in KNN. The efficiencies are achieved by limiting the number of data points stored, and the merits are preserved by keeping only the data points that are likely to increase the future prediction accuracy.

The idea of limiting the number of data points stored to improve KNN performance has already been extensively studied in the pattern classification literature[Chang 1974; Hart 1968; Wilson 1972] in the form of *edited K nearest neighbors (EKNN)*. They, however, use computationally expensive off-line methods to reduce the training dataset size by compressing the initial dataset. There are two important distinctions between that group of work and our work. First, their goal is to improve prediction speed at no cost to accuracy[Chang 1974; Hart 1968] or, sometimes, to improve the accuracy [Wilson 1972]. Naturally, they can not work with a memory limit. In contrast, we are willing to compromise accuracy in order to ensure that memory usage stays within a limit. Second, their approaches are static and, therefore, do not allow for incremental training. In contrast, our approach is dynamic and allows for low-cost incremental training.

In this section, we show the utility of the guidelines by presenting an instance-based cost modeling technique developed from it. We first describe the data structures used by MLKNN in Section 5.1. In Sections 5.2, 5.3, and 5.4, we elaborate on MLKNN's cost prediction, data point insertion, and model compression, respectively. Table I provides a summary of the parameters used in MLKNN.

5.1 Data structures

MLKNN makes use of two data structures: *point data structure (PData)* and a *multi-dimensional index*. PData stores the following information per data point: the coordinates, the UDF execution cost (either CPU or IO cost), and the utility value (to be defined formally in Equation 11).

The multi-dimensional index is used for fast retrieval of the PData of the K nearest neighbors and fast insertion of new PData. We can use any of the existing index structures designed for efficient KNN search [Beckmann et al. 1990; Cheung and chee Fu 1998; Kim et al. 2001; Yu et al. 2001]. Note that all those indexes will benefit from reduced retrieval and insertion costs and reduced memory usage when the number of data points stored is reduced. We have used the R*-tree[Beckmann et al. 1990] enhanced with Cheung and Fu's improved KNN search algorithm[Cheung and chee Fu 1998] in the experiments of this paper. The reasons for this choice are given in Section 7.1.1.

5.2 Cost prediction

For MLKNN, the prediction guideline P translates into automatically determining the number (K) of nearest neighbors to use depending on the noise-level, that is, setting K higher if the noise-level is higher. For this purpose, we maintain a set of the running sums of absolute prediction errors ($\{e_{K_1}, e_{K_2}, \cdots, e_{K_N}\}$) for a set of distinct values of K ($K_1 < K_2 < \cdots < K_N$) and, when making a cost prediction, use the K_s for which the e_{K_s} ($1 \leq s \leq N$) is the minimum among $e_{K_1}, e_{K_2}, \cdots, e_{K_N}$. This method is based on the assumption that the optimal K (among those considered) in the past is likely to be optimal now. Maintaining the multiple running sums incurs little additional run-time overhead compared with maintaining one running sum for K_N because, to compute each of the other running sums, we can simply reuse a subset of the K_N data points.

Given a particular K, the algorithm for predicting the cost is simple and straightforward, as outlined in Figure 4. Given a query point, the algorithm first finds its K nearest neighbors through the multidimensional index based on the Euclidean distance. Then, it calculates the weighted average of their costs (using Equation 7) and returns the result as the predicted costs. Most KNN searches using multi-dimensional indexes require the calculation of the distances between the query point and its neighbors. We reuse these distances (already calculated) to compute the weights.

MLKNN_Predict_Cost (K: number of nearest neighbors, MI: multi-dimensional index,
$\mathbf{q}$: query point)

1. Find the K nearest neighbors $\mathbf{p_1}, \mathbf{p_2}, ..., \mathbf{p_K}$ of query point $\mathbf{q}$ using the multi-dimensional index MI.
2. Return the weighted average of the costs for $\mathbf{p_1}, \mathbf{p_2}, ..., \mathbf{p_K}$. (See Equation 7.)

Fig. 4. Cost prediction algorithm of MLKNN.

We now give a more formal description of how the predicted cost is calculated for a given K. Let $\mathbf{p_1}, \mathbf{p_2}, \cdots, \mathbf{p_K}$ be the K nearest neighbors of a query point $\mathbf{q}$, and let their corresponding UDF execution costs and the Euclidean distances from $\mathbf{q}$ be $C(\mathbf{p_1}), C(\mathbf{p_2}), \cdots, C(\mathbf{p_K})$ and $D(\mathbf{p_1}), D(\mathbf{p_2}), \cdots, D(\mathbf{p_K})$, respectively. Then, the predicted cost, PC, at the query point $\mathbf{q}$ is calculated as

$$PC(\mathbf{q}) \;=\; \sum_{i=1}^{K} \frac{w(\mathbf{p_i})}{\sum_{i=1}^{K} w(\mathbf{p_i})}\, C(\mathbf{p_i}) \tag{7}$$

where $w(\mathbf{p_i})$ is the weight assigned to $\mathbf{p_i}$; weights are normalized by the total weight (in the denominator) so the sum of all weights equals 1. To compute the weight we use the *kernel smoothing* method, particularly the one based on the popular Epanechnikov kernel[Wand and Jones 1995], given as

$$\Phi(u) \;=\; \frac{3}{4}(1 - u^2) \tag{8}$$

where u is a real number between 0 and 1 (inclusive).

Using $\Phi(u)$, the weight $w(\mathbf{p_i})$ is expressed as

$$w(\mathbf{p_i}) \;=\; \Phi(\frac{\|\mathbf{q}-\mathbf{p_i}\|}{D(\mathbf{p_K})}) \tag{9}$$

There are a number of other weighting methods that can be used, such as uniform, triangular rank-based, and quadratic rank-based weights [Stone 1977]. We have chosen to use the Epanechnikov kernel smoothing method for its popularity[Wand and Jones 1995].

5.3 Data point insertion

The insertion algorithm follows the guideline I by inserting data points into the model only if prediction error (M_{pe}) exceeds a certain threshold T_{pe}. We call this *selective insertion*. Figure 5 outlines the algorithm. It first decides whether the query point should be inserted as a data point into the multi-dimensional index (Line 1). This is done by checking if the prediction error (defined in Equation 10) is above T_{pe}. Then, it updates the utility values of the data points used to make the prediction (i.e., the K nearest neighbors of the query point) using Equation 11.

MLKNN_Insert_Data_Point ($\mathbf{q}$: query point, $\mathbf{p_1}, \mathbf{p_2}, ..., \mathbf{p_K}$: K nearest neighbors of $\mathbf{q}$,
MI: multi-dimensional index, T_{pe}: prediction error threshold)

1. if ($M_{pe}(\mathbf{q}) > T_{pe}$) then begin
2. Insert $\mathbf{q}$ as a data point into MI.
3. end if.
4. Update the utility values of $\mathbf{p_1}, \mathbf{p_2}, ..., \mathbf{p_K}$. (See Equation 11.)

Fig. 5. Insertion algorithm of MLKNN.

M_{pe} is measured as follows.

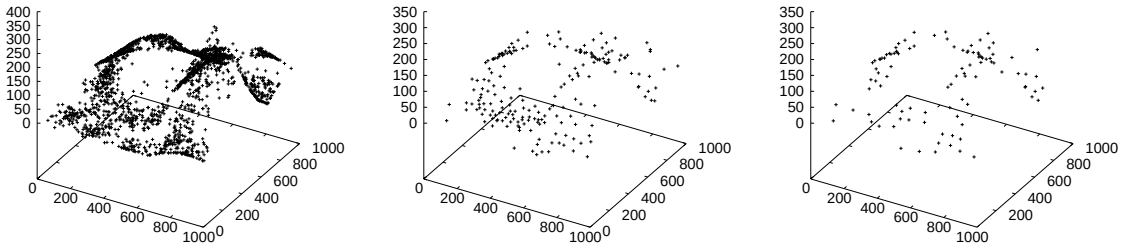
$$M_{pe}(\mathbf{q}) \;=\; \frac{|AC(\mathbf{q}) - PC(\mathbf{q})|}{max(AC(\mathbf{q}), PC(\mathbf{q}))} \tag{10}$$

where $AC(\mathbf{q})$ is the actual cost at a query point $\mathbf{q}$ and $PC(\mathbf{q})$ is the predicted cost at the same query point. The query point $\mathbf{q}$ is inserted into the model only if its $M_{pe}(\mathbf{q}) > T_{pe}$. Equation 10 allows T_{pe} to be set using a fraction, which is more intuitive for users than the absolute difference.

This selective insertion algorithm fulfills the guideline I1 because M_{pe} is more likely to exceed T_{pe} in regions with more complex cost variations and, consequently, more data points are inserted into these regions. This leads to more memory allocated to model these regions. The same algorithm fulfills the guideline I2 as well because more frequently queried regions are more likely to have their sub-regions with complex cost variations discovered and, thus, more data points are inserted into these regions.

Figure 6(a)-(b) illustrates the effect of selective insertion performed on an original dataset (Figure 6(a)) containing 2000 data points. After selectively inserting the first 1086 data points, only 200 are kept in the model. Figure 6(b) shows that this small set of 200 data points models quite precisely the complex shape formed by the original data points.

After each time a prediction is made, MLKNN updates the utility value of each of the K nearest neighbors of the query point. (These utility values are used by the compression algorithm in Section 5.4 to decide how aggressively each data points should be compressed.) Formally, the utility value of a given data point $\mathbf{p_i}$, denoted by $U(\mathbf{p_i})$, is defined as



(a) Original data points. (b) After selective insertion. (c) After RR compression.

Fig. 6. Example of the effect of model update.

$$U(\mathbf{p_i}) \;=\; \sum_{\mathbf{q} \in Q(\mathbf{p_i})} UI(\mathbf{p_i}, \mathbf{q}) \tag{11}$$

where $Q(\mathbf{p_i})$ is the set of query points for which $\mathbf{p_i}$ has been used to make predictions and $UI(\mathbf{p_i}, \mathbf{q})$ is an increment of $U(\mathbf{p_i})$ at the data point $\mathbf{p_i}$ used for prediction at the query point $\mathbf{q}$.

$UI(\mathbf{p_i}, \mathbf{q})$ is calculated as

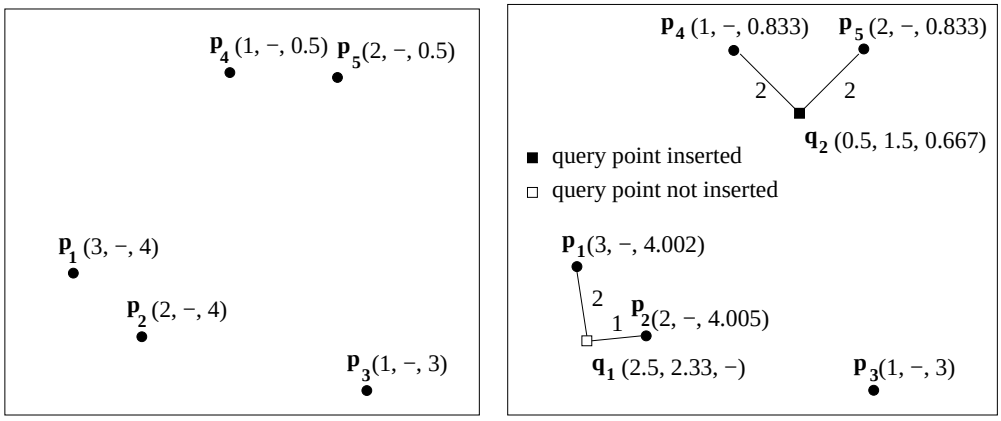
$$UI(\mathbf{p_i}, \mathbf{q}) \;=\; w(\mathbf{p_i}) \times M_{pe}(\mathbf{q}) \tag{12}$$

If the query point ($\mathbf{q}$) is inserted into the model as a new data point, then its utility value is initialized to $M_{pe}(\mathbf{q})$. Note that the utility value $U(\mathbf{p_i})$ is high if: (1) the frequency ($|Q(\mathbf{p_i})|$) of using $\mathbf{p_i}$ to make predictions is high, since $U(\mathbf{p_i})$ is increased every time $\mathbf{p_i}$ is used to make a prediction, (2) $\mathbf{p_i}$ is in a region in which there is highly complex cost variations, since points in these regions are more likely to give larger prediction errors ($M_{pe}(\mathbf{q})$), and (3) the distance between the data point $\mathbf{p_i}$ and the query point $\mathbf{q}$ is small, since $w(\mathbf{p_i})$ gives higher weight to data points closer to the query point $\mathbf{q}$ (Equation 9). This property allows the utility value to be used as a metric for determining how aggressively the model should be compressed (see Section 5.4).

Figure 7 illustrates the selective insertion of two new query points $\mathbf{q_1}$ and $\mathbf{q_2}$. Suppose the number (K) of nearest neighbors used for prediction is 2. Further suppose $M_{pe}(\mathbf{q_1})$ is below the threshold for insertion whereas $M_{pe}(\mathbf{q_2})$ is above the threshold. Then, $\mathbf{q_2}$ is inserted (with the utility value initialized to $M_{pe}(\mathbf{q_2})$, which equals 0.667 by Equation 10), but $\mathbf{q_1}$ is not inserted. The utility values of $\mathbf{p_1}$, $\mathbf{p_2}$, $\mathbf{p_4}$, and $\mathbf{p_5}$ (data points used to make the predictions) are increased according to Equation 12.

5.4 Model compression

The key idea behind MLKNN's compression algorithm is to compress data points with higher utility value less aggressively. This approach follows guideline C1 since data points with higher utility values are those that are in regions of higher cost variation complexity (as explained in Section 5.3). It also follows guideline C2 since data points that have higher utility values are those that have been frequently used for predictions.



(a) Before new query points. (b) After new query points.

$\mathbf{p_i}$(a,-,u), $\mathbf{q_i}$(a,p,u): a = actual cost, p = predicted cost, u = utility value.

Fig. 7. Selective insertion of query points.

During compression, existing data points are compressed to a smaller number of 'representative' points. The multi-dimensional index is then rebuilt[3] using only the representative points. The representative points are then treated in the same way as new data points inserted into the multi-dimensional index.

We consider two new compression algorithms of contrasting characteristics. The first algorithm is called *rank and remove(RR)* and the second algorithm is called *partition and merge(PM)*.

5.4.1 *Rank and remove(RR).* This algorithm is designed to be very computationally efficient by completely ignoring where the data points are located when performing the compression. Compression is only guided by the utility value. Figure 8 outlines the algorithm. It first sorts all existing data points in the decreasing order of the utility value. Then, it removes the data points in the bottom fraction, and selects the remaining data points as the representative points. The size of the removed fraction is determined by the model compression ratio (MCR).

MLKNN_RR_Compress (MI: multi-dimensional index, DP: the set of all data points inserted, MCR: model compression ratio)

1. Sort all data points DP in the decreasing order of the utility value.
2. Remove the bottom MCR fraction of the data points.
3. Select the remaining data points as representative data points.
4. Rebuild multi-dimensional index MI with representative data points.

Fig. 8. RR Compression algorithm of MLKNN.

[3]Typically, rebuilding is much more efficient than replacing the existing points through a sequence of deletions and insertions.

The complexity of this compression algorithm is dominated by the initial sorting of the data points (required to rank the data points by the utility value), which has the complexity of $O(n \log n)$ where n is the number of data points.

Figure 6(b)-(c) illustrates the effect of RR compression for MCR 50%. Figure 6(b) shows the 200 data points kept in the model before the compression, and Figure 6(c) shows the 100 data points retained after the compression. From the figures we can see the complex shape formed by the uncompressed data points are kept after RR compression.

5.4.2 *Partition and merge(PM).* This algorithm contrasts from RR by taking the spatial proximity of data points into consideration when deciding how to compress them. This algorithm aims to ensure that every region of the multi-dimensional space has a data point. Figure 9 outlines the algorithm. It first partitions the data points using a histogram approach and, then, merges the points in the same partition to a single representative point.

MLKNN_PM_Compress (MI: multi-dimensional index, DP: the set of all data points inserted, MCR: model compression ratio)

1. Let $|DP|$ be the number of data points in DP.
2. Partition all data points in DP into partitions of $(1 - MCR)|DP|$ points using a histogram approach.
3. For each partition p created begin
4. merge all data points in p to a representative point.
5. end for
6. Rebuild the multi-dimensional index MI with the representative data points.

Fig. 9. PM Compression algorithm of MLKNN.

In the partitioning step, it uses a simple extension of the equi-height histogram approach used in [Boulos and Ono 1999]. In [Boulos and Ono 1999], the height of a partition is defined as the number of data points in each partition. In our case, it is defined as the total *utility value* of all data points in the partition. The algorithm sorts the data points separately in each dimension based on the coordinate of the data points, generating d lists, where d is the number of dimensions. Then each list is partitioned into P partitions of equal height, thus creating P^d multi-dimensional partitions. The parameter P is determined by MCR as $P = \sqrt[d]{\text{MCR} \times |DP|}$. The complexity of the algorithm in a d-dimensional model space is $O(d|DP| \log |DP|)$. The complexity is dominated by the need to sort the data points by the coordinate in each dimension.

In the merge step, the coordinate of each representative point is computed as a weighted average of the coordinates of the data points in the same partition. The utility value of the data points are used as the weights. Formally, let $CO_i(\mathbf{p})$ be the value of the i^{th} component of point $\mathbf{p}$'s coordinate, let $\mathbf{p_1}, \mathbf{p_2}, ..., \mathbf{p_n}$ be the points in the partition merged to a representative point $\mathbf{r}$, and let $U(\mathbf{p_j})$ be the utility value of the point $\mathbf{p_j}$ for $j = 1, 2, ..., n$. Then, $CO_i(\mathbf{r})$ is computed as

$$CO_i(\mathbf{r}) \;=\; \sum_{j=1}^{n} \frac{U(\mathbf{p_j})}{\sum_{a=1}^{n} U(\mathbf{p_a})} CO_i(\mathbf{p_j}) \tag{13}$$

Let $C(\mathbf{r})$ be the execution cost associated with a representative point $\mathbf{r}$. Then, $C(\mathbf{r})$ is computed as a weighted average of the execution costs of the data points contained in the partition, i.e., $C(\mathbf{p}_1), C(\mathbf{p}_2), ..., C(\mathbf{p}_n)$,

as

$$C(\mathbf{r}) \;=\; \sum_{i=1}^{n} \frac{w(\mathbf{p_i})}{\sum_{i=1}^{K} w(\mathbf{p_i})}\, C(\mathbf{p_i}) \tag{14}$$

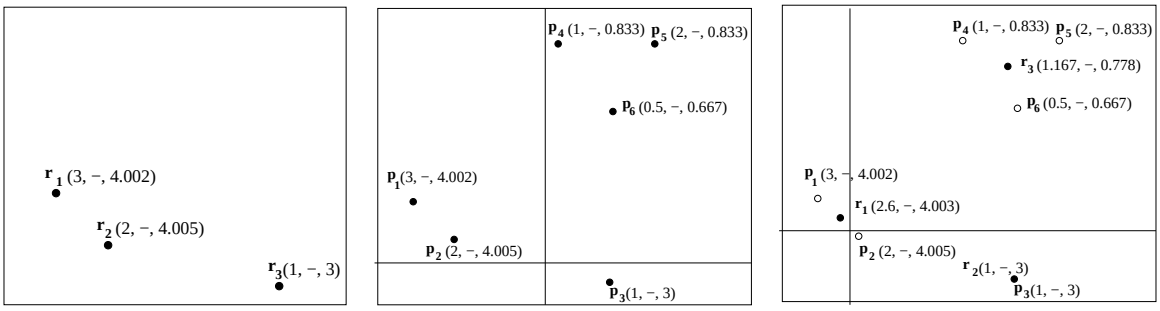
where $w(\mathbf{p_i})$ is determined by Equation 9.

The utility value of the representative point $\mathbf{r}$, $U(\mathbf{r})$, is computed as a weighted average of the utility values of the data points contained in the partition, i.e., $U(\mathbf{p}_1), U(\mathbf{p}_2), \cdots, U(\mathbf{p}_n)$, as

$$U(\mathbf{r}) \;=\; \sum_{i=1}^{n} \frac{w(\mathbf{p_i})}{\sum_{i=1}^{K} w(\mathbf{p_i})}\, U(\mathbf{p_i}) \tag{15}$$

Using Equation 13 causes the coordinate of the representative point to be skewed toward data points with high utility values, and using Equation 14 causes data points with a smaller Euclidean distance from the representative point to have a larger influence on its cost.

Figure 10 shows how the data points in Figure 7(b) are compressed using RR and PM compression algorithms. It can be observed that PM creates representative points that are more uniformly spaced across the indexed model space. This is because RR does not take spatial proximity into consideration, whereas PM does.



(a) After RR compression ($MCR = 0.5$).

(b) After partition in PM compression ($P = 2$, i.e., $MCR = 2/3$).

(c) After merge in PM compression.

$\mathbf{p_i}$(a,-,u), $\mathbf{r_i}$(a,-,u): a = actual cost, u = utility value.

Fig. 10. Model compression of data points in Figure 7 (b).

6. MEMORY-LIMITED QUADTREE (MLQ)

As mentioned in Section 1.4, MLQ is a summary-based technique as opposed to an instance-based technique like MLKNN. In MLQ, the quadtree is used to store summary information at *different resolutions* based on the complexity of UDF execution costs and the distribution of query points. The summary information is updated incrementally each time a new data point is inserted, and is used to calculate the predicted cost and to guide the compression of the quadtree when the memory limit is reached.

The quadtree is used as a summarization structure used in many application areas including approximate query processing[Lazaridis and Mehrotra 2001], selectivity estimation[Buccafurri et al. 2003], and image

processing[Tousidou and Manolopoulos 2000; Nardelli and Proietti 1994]. There are other summarization structures as well, such as other types of trees (e.g., dynamic KD-trees [Robinson 1981; Procopiuc et al. 2003], hB-trees [Evangelidis et al. 1997; Lomet and Salzberg 1990], R-trees [Beckmann et al. 1990; Guttman 1984]) and histograms [Aboulnaga and Chaudhuri 1999; Bruno et al. 2001; Poosala and Ioannidis 1997]. We have chosen the quadtree because of its support for fast retrieval and fast incremental update, as well as the multi-resolution model described next.

In this section, we describe the quadtree structures used by MLQ in Section 6.1, define the optimality criterion of the quadtree in Section 6.2, and elaborate on MLQ's cost prediction, data point insertion, and compression algorithms in Sections 6.3, 6.4, and 6.5, respectively. Table II summarizes the MLQ parameters, and Table III lists the statistics used by MLQ.

Parameter	Description
T_{ms}	the minimum support threshold, used to determine the minimum count (i.e., number of data points) needed to make a prediction
α	the scaling factor ($0 \leq \alpha \leq 1$) used to determine T_{SSE}, the SSE threshold used to decide when to split a quadtree node.
MCR	the model compression ratio, used as measure of how aggressively the model as a *whole* should be compressed.
λ	the maximum quadtree depth.

Table II. Summary of MLQ parameters.

Term	Description
$C(b)$	number of data points in block b.
$S(b)$	sum of the values of data points in block b.
$AVG(b)$	average of the values of data points in block b.
$SS(b)$	sum of squares of the values of data points in block b.
$SSE(b)$	sum of squared errors of the values of data points in block b.
$SSENC(b)$	SSE of the values of data points in block b excluding those in its children.
$TSSENC(qt)$	total $SSENC$ for all non-full blocks of quadtree qt.
$TSSENCG(b)$	TSSENC gained as a result of removing a block b from the quadtree.

Table III. Summary of statistics used in MLQ.

6.1 Quadtree structure

MLQ uses the conventional quadtree structure. 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. 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 11 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

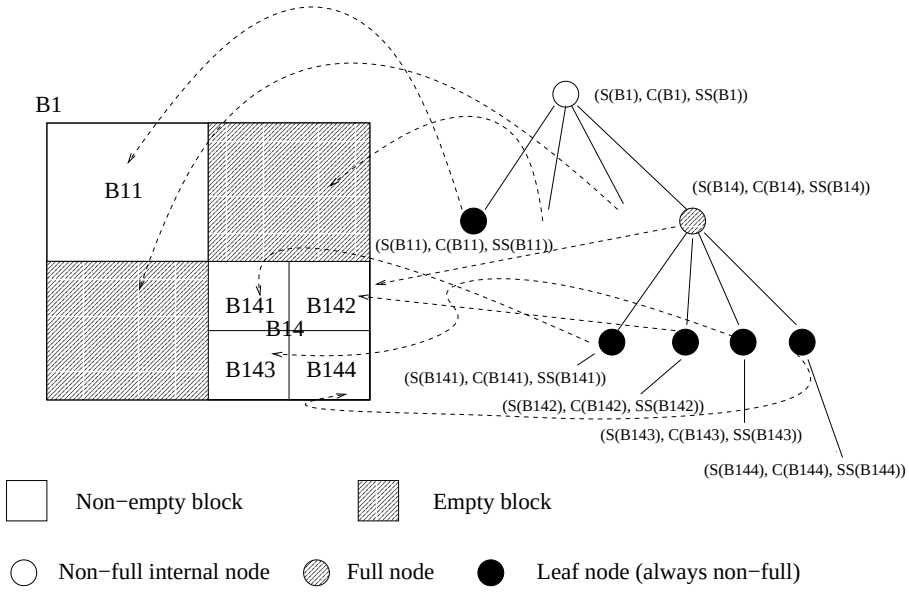


Fig. 11. The quadtree data structure.

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.

$$AVG(b) \;=\; \frac{S(b)}{C(b)} \tag{16}$$

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

$$\begin{aligned} SSE(b) \;&=\; \sum_{i=0}^{C(b)} (V_i - AVG(b))^2 \\ &=\; SS(b) - C(b)(AVG(b))^2 \end{aligned} \tag{17}$$

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

6.2 Optimal quadtree

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 used 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 values of data points in block b, excluding those in its children. That is,

$$SSENC(b) = \sum_{i=1}^{C(b_{nc})} (V_i - AVG(b))^2 \tag{18}$$

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}. Then, we define the optimal quadtree as the one that minimizes the *total* $SSENC$ ($TSSENC$) defined as follows.

$$TSSENC(qt) = \sum_{b \in NFB(qt)} (SSENC(b)) \tag{19}$$

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.

$SSENC(b)$ 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[Buccafurri et al. 2003]. It is used in [Buccafurri et al. 2003] 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.

6.3 Cost prediction

For MLQ, the prediction guideline P translates into automatically determining the value of T_{ms} for finding the quadtree node whose summary values are to be used for prediction. Since, by definition, T_{ms} is the minimum count value of the node to use for making a prediction, its value can be adjusted depending on the level of noise, that is, set higher if the noise-level is higher. Like MLKNN, MLQ maintains a set of running sums of absolute prediction errors ($\{e_{T_{ms_1}}, e_{T_{ms_2}}, \cdots, e_{T_{ms_N}}\}$) for N different T_{ms_i} values ($T_{ms_1} < T_{ms_2} < \cdots < T_{ms_N}$) and, when making a cost prediction, use the T_{ms_s} for which the $e_{T_{ms_s}} (1 \leq s \leq N)$ is the minimum among $e_{T_{ms_1}}, e_{T_{ms_2}}, \cdots, e_{T_{ms_N}}$. As in MLKNN, this method is based on the assumption that the optimal T_{ms} (among those considered) in the past is likely to be optimal now. Maintaining the multiple running sums incurs little additional run-time overhead compared with maintaining one running sum for $e_{T_{ms_1}}$ because, to compute each of the other running sums, we can simply reuse one of the nodes traversed on the way to the node for $e_{T_{ms_1}}$.

Given a particular T_{ms}, the algorithm for predicting the cost (using the quadtree structure) is simple and straightforward, as outline in Figure 12. The algorithm first finds the lowest level node that the query point maps into and that has the count value of at least T_{ms}. It then returns the average calculated using the summary values $S(b)$ and $C(b)$ stored in the node.

MLQ_Predict_Cost (QT: quadtree, **q**: query point, T_{ms}: minimum support threshold)

1. Find the lowest level node of QT such that **q** maps into the block of the node and the count in the node $\geq T_{ms}$.
2. Return sum/count from the node found.

Fig. 12. Cost prediction algorithm of MLQ.

Figure 13 illustrates cost prediction in MLQ. It shows three query points $\mathbf{q}_1$, $\mathbf{q}_2$, and $\mathbf{q}_3$. If $T_{ms} = 3$, then $\mathbf{q}_1$ returns a predicted cost of 100 (= 300/3) from B14, $\mathbf{q}_2$ returns 40 (= 200/5) from B12, and $\mathbf{q}_3$ also returns 40 from B12. Note that $\mathbf{q}_3$ does not return the average from B124 because the leaf node that $\mathbf{q}_3$ maps into has a count less than 3.

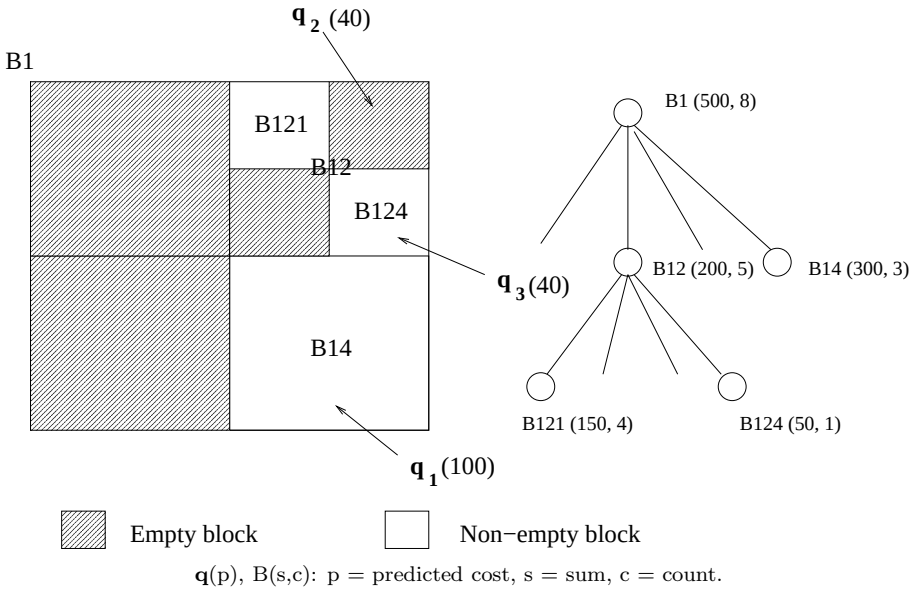


Fig. 13. An example of cost prediction in MLQ ($T_{ms} = 3$).

6.4 Data point insertion

The insertion algorithm follows the insertion guideline I provided in Section 4.2 by partitioning a quadtree block only if its SSE reaches a certain threshold (T_{SSE}) as a result of inserting a data point. Figure 14 shows the algorithm. First, it traverses the quadtree top down to a leaf node while updating the summary values stored in every node the data point maps into (Lines $1 \sim 6$). Then, if the SSE value stored in the leaf node is larger than T_{SSE} and if the depth of the leaf node is smaller than the maximum allowed quadtree height (λ), then it creates a new child node that the data point maps into and initializes the summary values to all zeros (Lines $7 \sim 10$).

```
Insert_point ( DP: data point, QT: quadtree, T_SSE: SSE threshold, λ: maximum depth )
1.  cn = the current node being processed, initialized to be the root node of QT.
2.  update sum, count, and sum of squares stored in cn.
3.  while (cn is not a leaf node) begin
4.      cn = the child of cn that DP maps into.
5.      update sum, count, and sum of squares in cn.
6.  end while
7.  if ((SSE(cn) ≥ T_SSE) and (depth of cn < λ)) then begin
8.      cn_new = create the child in cn that DP maps into.
9.      initialize sum, count, and sum of squares in cn_new to zeros.
10. end if
```

Fig. 14. Insertion algorithm of MLQ.

T_{SSE} is defined as follows.

$$T_{SSE} \;=\; \alpha\; SSE(r) \tag{20}$$

where r is the root block and the parameter α is a *scaling factor* provided by users to set the value of T_{SSE}. The value of SSE in the root node indicates the degree of cost variations in the *entire* data space. For this reason, the value of T_{SSE} can be set relative to the value of $SSE(r)$. As a slight variation, MLQ sets T_{SSE} to zero initially until the memory limit is reached (and the first compression occurs), before it starts updating T_{SSE} according to Equation 20. Setting T_{SSE} to zero makes the quadtree blocks to partition to the maximum depth at every data point insertion. This enables the algorithm to achieve adequate prediction accuracy during the initial period of inserting data points while the data space is sparsely populated.

In MLQ, it is through the partitioning of quadtree blocks (following the guideline I) that more memory is allocated in regions where data points are inserted. Therefore, the insertion algorithm fulfills the guideline I1 because SSE values are higher in the blocks located in regions with more complex cost variations. The same algorithm fulfills the guideline I2 as well because SSE values are higher in the blocks located in regions into which more data points are inserted.

Figure 15 illustrates how the quadtree is changed as two new data points $\mathbf{p}_1$ and $\mathbf{p}_2$ are inserted. In this example, we use $T_{SSE} = 8$ and $\lambda = 5$. When $\mathbf{p}_1$ 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 T_{SSE}. Next, when $\mathbf{p}_2$ is inserted, B14 is partitioned since its updated SSE of 52.67 becomes greater than the T_{SSE}.

6.5 Model compression

The key idea of compression in MLQ is to remove some nodes of the quadtree so that the removal results in the minimum increase of the quadtree's $TSSENC$ while freeing at least the memory amounting to MCR. Here, removing the leaf nodes before the internal nodes allows the compression to be done incrementally, since removing an internal node automatically removes all its children nodes as well. To implement this idea, we insert all leaf nodes into a priority queue, keyed by the TSSENC *gain* (TSSENCG) defined as follows.

$$TSSENCG(b) \;=\; SSENC(p_{ac}) - (SSENC(b) + SSENC(p_{bc})) \tag{21}$$

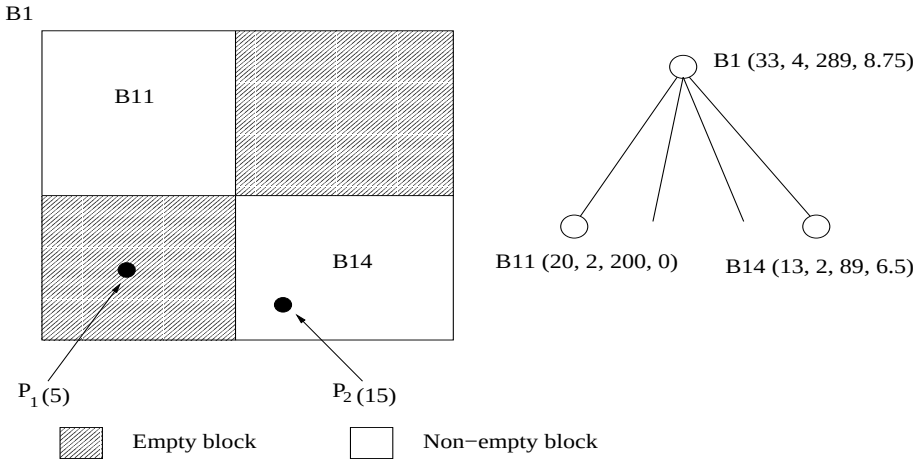
where b is a quadtree block, p_{bc} and p_{ac} refer to the states of the parent block of b before and after the removal of b, respectively. Thus, $TSSENCG(b)$ is the $TSSENC$ gain resulting from removing the block b.

Using the derivation shown in Appendix A, Equation 21 can be simplified to the following equation.

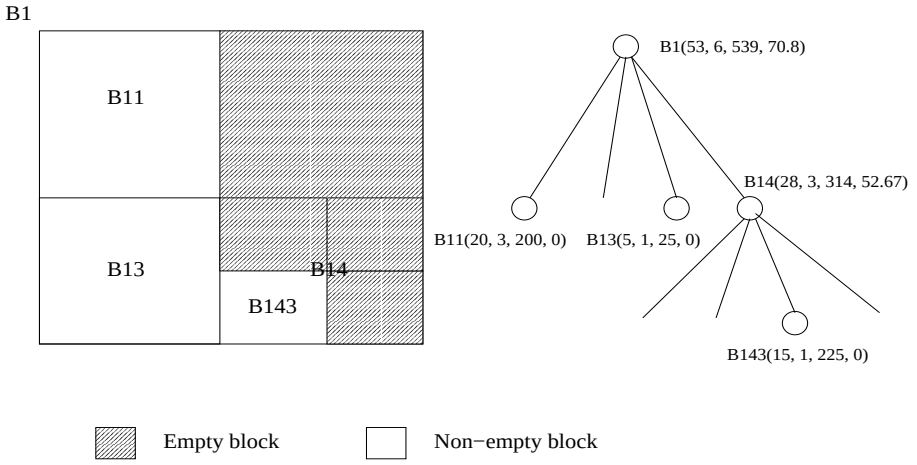
$$TSSENCG(b) \;=\; C(b)(AVG(p) - AVG(b))^2 \tag{22}$$

where p is the parent block of b. Using Equation 22 as the basis of compression offers three desirable properties, two of which fulfills the guidelines C1 and C2. First, it favors the removal of a leaf node that shows a smaller difference between the average cost for the node and the average cost for its parent (i.e., smaller $(AVG(p) - AVG(b))^2$). This fulfills the guideline C1 because regions with lower complexity of cost variations have leaf nodes with smaller differences in the average costs from those for their parents. Second, it favors the removal of a leaf node that had fewer data points inserted into (i.e. smaller $C(b)$). This fulfills the guideline C2. Third, the computation of $TSSENCG(b)$ is efficient as it can be done using the sum and count values already stored in the quadtree nodes.

Figure 16 outlines the compression algorithm. First, it inserts all the leaf nodes into the priority query (PQ) keyed by their $TSSENCG$ values (Line 1). Then, it removes the leaf nodes from the PQ, the one with the smallest $TSSENCG$ first (Lines 2 - 10). If the removal of a leaf node results in its parent becoming a leaf node, then the parent node is inserted into the PQ (Lines 5 - 7). The algorithm stops removing the leaf nodes when either the PQ becomes empty or the fraction of freed memory reaches at least MCR.



(a) Before insertion of new data points.



(b) After insertion of new data points.

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

Fig. 15. An example of data point insertion in MLQ.

```
MLQ_Compress_Model (QT: quadtree, MCR: model compression ratio,
                    total_mem: the total amount of memory allocated)
1.  Traverse QT and insert every leaf node into a priority queue PQ keyed by its TSSENCG.
2.  while (PQ is not empty) and (memory_freed / total_mem < MCR) begin
3.      remove the minimum-TSSENCG lead node from PQ and return it as 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) then begin
6.          insert parent_node into PQ keyed by its TSSENCG.
7.      end if
8.      deallocate memory used by current_leaf.
9.      memory_freed = memory_freed + size of current_leaf
10. end while
```

Fig. 16. Compression algorithm of MLQ.

Figure 17 illustrates how MLQ performs compression. Figure 17(a) shows the state of the quadtree before the compression. Either B141 or B144 can be removed first since they both have the lowest $TSSENCG$ value of 1. The tie is arbitrarily broken, resulting in 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.

7. EXPERIMENTAL EVALUATION

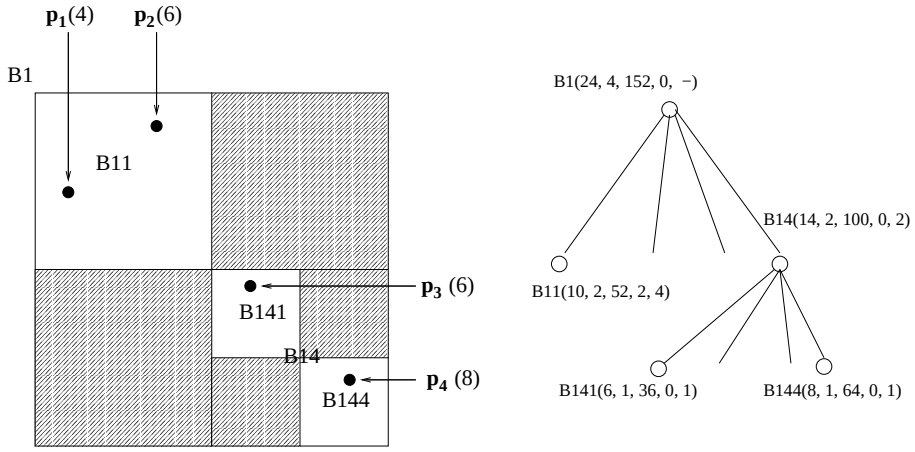
We evaluate the performances of MLKNN, MLQ, and KNN against the existing technique SH. In this section, we first describe the experimental setup in Section 7.1 and, then, present the experimental results in Section 7.2.

7.1 Experimental setup

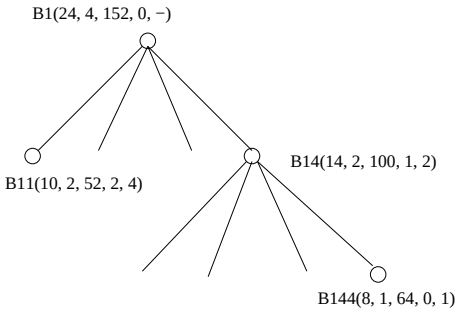
7.1.1 *Modeling techniques.* We compare the performances of two MLKNN variants and MLQ against two variants of SH and the KNN technique. Specifically, the following techniques are compared: (1) **MLKNN-SRR**, MLKNN using selective insertion and RR compression, (2) **MLKNN-SPM**, MLKNN using selective insertion and PM compression, (3) **MLQ**, (4) **SH-H**[Boulos and Ono 1999] using equi-height histograms, (5) **SH-W**[Boulos and Ono 1999] using equi-width histograms, and (6) **KNN**, the K nearest neighbor technique storing all data points.

For the techniques listed above, we train and test the models in different manners depending on whether the techniques are static or dynamic. The SH techniques are static. We build a model (i.e., histogram) a priori using training queries. Then, we use the model to make predictions during testing using testing queries. In contrast, the KNN-based and MLQ techniques are dynamic. We build a model incrementally (i.e., one query point at a time) during testing. Since a dynamic technique can be effective only after the model has been trained adequately, we first build an initial model statically using training queries and, then, update it during testing using test queries. For both the static and dynamic algorithms the test queries have the same distribution as the training queries. More specifics of query generation will appear in Section 7.1.4.

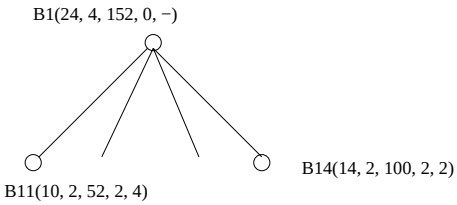
The multi-dimensional index used for the KNN-based techniques is the R*-tree[Beckmann et al. 1990] enhanced with Cheung and Fu's improved KNN search algorithm[Cheung and chee Fu 1998]. This algorithm aggressively prunes the search space, thereby resulting in fewer R*-tree node accesses. Moreover, the R*-tree allows for incremental insertions without the need for periodic rebuilding unlike other index structures (e.g., Δ-tree [Cui et al. 2003]) that require periodic rebuilding after many insertions.



(a) Before compression.



(b) After compression algorithm removes block B141.



(c) After compression algorithm removes block B144.

p(e), 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. 17. An example of MLQ compression.

In all the experiments except the one with varying memory limit, we have limited the amount of allocated memory to 10 KB (with the exception of KNN, which retains *all* data points). This amount is similar to that allocated in existing work for selectivity estimation of range queries [Bruno et al. 2001; Deshpande et al. 2001; Poosala and Ioannidis 1997].

We have tuned the KNN-based and MLQ techniques to achieve the best overall performance, and used the resulting parameter values. In the case of the SH methods, there is no tuning parameter except the number of buckets used, which is determined by the given allocated memory size. The following is a specification of the parameters used for the KNN-based techniques (including MLKNN parameters in Table I): K = $1 \sim 10$ (automatically adjusted to the noise-level), $T_{pe} = 0.1$, and $MCR = 0.5$. In addition, R*-tree fanout is set

to 20 for MLKNN techniques and 10 for KNN, and R*-tree fill factor is set to 0.9 for MLKNN techniques and 0.7 for KNN. The following is a specification of the parameters used for MLQ (in Table II): $T_{ms} = 1 \sim 10$ (automatically adjusted to the noise-level), $\alpha = 0.05$, $MCR = 0.1$, and $\lambda = 6$.

7.1.2 *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 model space. The coordinates of the peaks have the uniform distribution, and the heights (i.e., execution costs) of the peak have the Zipf distribution[Zipf 1949]. 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.

The following is a specification of the parameters used to generate the data sets: the number of dimensions $d = 3$, the range of values in each dimension $= 0 \sim 1000$, the maximum cost at the highest peak $= 10000$, the Zipf parameter (z) value $= 1$, the standard deviation for the Gaussian decay function $= 0.2$, and the distance $D = 10\%$ of the Euclidean distance between two extreme corners of the multi-dimensional model space.

7.1.3 *Real UDFs/datasets.* In this subsection, we introduce the real UDFs used in the experiment and outline determining their model variables. We use two kinds of real UDFs: three keyword-based text search functions and three spatial search functions. 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 [Pennsylvania 2003].

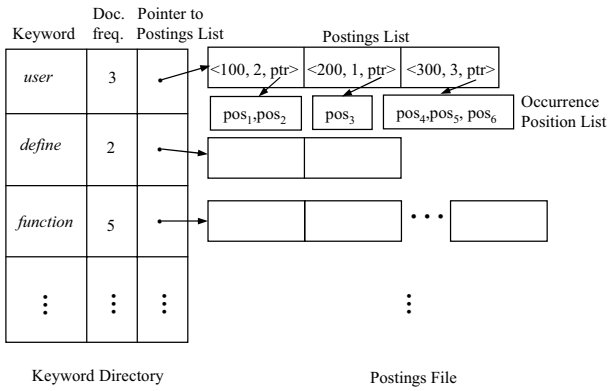
Table IV lists the prototypes, model variables, and the dimensionality (i.e. the number of model variables) of the real UDFs. The first three UDFs are keyword-based text search functions, and the last three are spatial search functions. *Simple Text Search (STS)* retrieves documents that contain all the keywords on the list. *Threshold Text Search (TTS)* retrieves documents that have the keywords appearing at least the threshold number of time; The retrieved documents do not need to contain all the keywords on the list. *Proximity Text Search (PTS)* retrieves documents in which all the keywords on the list appear at least once and at most max_span words apart. *K-Nearest Neighbors (KNN)* takes a two-dimensional reference point and the number (K) of neighbors and, then, retrieves the K nearest neighbors. *Window (WIN)* retrieves all objects contained in or overlapping a query window; The window is specified as the two-dimensional coordinates of the bottom left (x_1, y_1)and the top right (x_2, y_2) corners of the window. *Range (RAN)* retrieves all objects within a specified distance from a given two-dimensional reference point. Each of these UDFs accesses the entire table of tuples containing queried texts or geometric objects. The resulting cost, therefore, should be divided by the number of tuples in the table to obtain the differential cost as needed in [Hellerstein 1998; 1994; Chaudhuri and Shim 1999].

Determining the model variables of a UDF appears complicated. The complexity may come from the input arguments unsuitable for direct transformation to model variables, the underlying data structures (e.g., indexes), etc. We believe, however, the task is simpler that it appears because it is the *semantics*, not the syntax (e.g., input arguments), of a UDF that determines the model variables. Moreover, usually only a few model variables influence the costs predominantly[4][Lee et al. 2003; VanHorn et al. 2003; Jiang et al. 2003]. This often renders the task of determining model variables feasible. After all, the users, who write UDFs, are likely to understand the semantics of their UDFs well.

[4]This is consistent with the principle of Occam's razor [Thorburn 1915].

UDF	Function Prototype	Model Variables	d
STS	STS(<list of keywords>)	num_docs, num_keywords	2
TTS	TTS(<list of keywords>, int threshold)	num_docs, num_keywords, threshold	3
PTS	PTS(<list of keywords>, int max_span)	num_docs, num_keywords, max_span	3
KNN	KNN(reference_point, int K)	reference_x, reference_y, K	3
WIN	WIN(float x1, float y1, float x2, float y2)	x1, y1, x2, y2	4
RAN	RAN(reference_point, float distance)	reference_x, reference_y, distance	3

Table IV. Real UDF prototypes, model variables, and dimensionality (d).



For each keyword, the index stores the document frequency (i.e., the number of documents in which the keyword appears) and a postings list. Each element in the postings list is a posting record consisting of the identifier of the document, the number of occurrences of the keyword within the document, and a list of the positions of those occurrences. In this example, the document frequency of the keyword *user* is 3; this keyword occurs twice in the document with ID 100, once in the document with ID 200, and three times in the document with ID 300.

Fig. 18. An example text search index.

A good example is the text search functions shown in Table IV. For instance, a text query STS("cat dog fight") takes a sequence of token words, all nominal, as the input argument. It may seem impossible to derive any numeric model variable from them. However, users with the basic understanding of a text search index (see Figure 18) would be able to figure out what influences the execution cost most – the number of documents retrieved (num_docs).

Given the search keywords of a text search function, it is straightforward to obtain the value of num_docs from the postings list.[5] In the interest of space, we do not describe the details here, and refer interested readers to [VanHorn et al. 2003].

The spatial search functions are another example. The model variables are determined straightforward from the semantics of the functions with little or no transformation of the input arguments. The underlying spatial index structures (e.g., R-tree, quadtree) have relatively insignificant effect on the costs. For example, the first two model variables of KNN, reference_x and reference_y, come directly from the input argument *reference point* as its coordinates, and the other variable, K, is the input argument K itself. Likewise, the model variables of WIN come directly from the input arguments. Interested readers are referred to [Jiang et al. 2003] for details.

[5]We have added a few more model variables (e.g., num_keywords, threshold, max_span) for different text search functions. All of them have much less effects on the costs than num_docs.

7.1.4 *Query distributions.* Query points are generated using three different random distributions of their coordinates: uniform, Gaussian-random, and Gaussian-sequential. In the uniform distribution, we generate query points uniformly in the entire multi-dimensional model space. We need two parameters for the Gaussian distribution: the number (c) of centroids and the number (n) of query points. In the case of Gaussian-random, we first generate the c Gaussian centroids with 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 pick each c 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 for the n query points.

The following is a specification of the parameters used to generate the query points with the Gaussian distributions: $c = 3$ and standard deviation $= 0.05$, and $n = 2500$.[6] Half the query points are used for training and the other half for testing. We repeat each experiment five times using different random seeds (for creating different queries that follow the same distribution) and report the average result. We have used the same query distribution for training and testing because typically query distribution does not change so fast. The same assumption has been used in [Aboulnaga and Chaudhuri 1999; Bruno et al. 2001] for self-tuning histograms for selectivity estimation.

7.1.5 *Computing platform.* In the experiments involving real datasets, we use Oracle9i on SunOS5.8, 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. We set up Oracle to use direct IO and thus bypass the operating system cache. This is often recommended for higher system performance and also allows us to perform controlled flushing of database pages from memory.

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, with 512 MB RAM, and 40 GB hard disk.

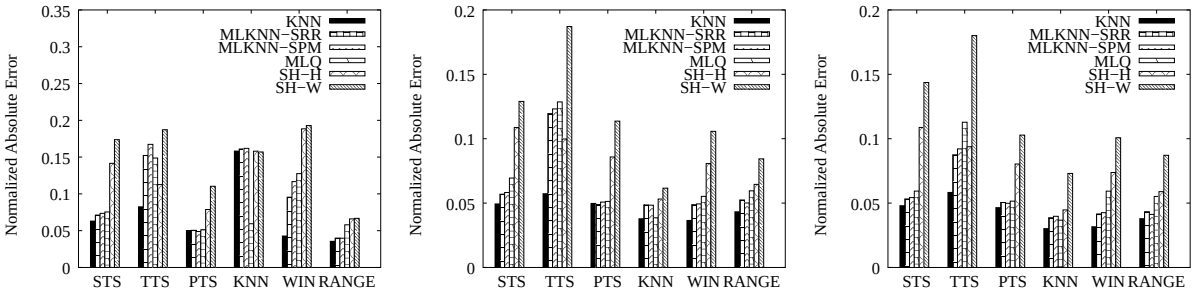
7.2 Experimental results

We have conducted seven different sets of experiments to compare (1) the prediction accuracies for various query distributions and UDFs/datasets, (2) the prediction accuracies for varying memory size, (3) the modeling costs, (4) the prediction accuracies in the presence of noises, (5) the prediction accuracies for varying K (for KNN-based techniques) and T_{ms} (for MLQ), (6) the prediction accuracies for an increasing number of query points processed starting from a cold start (i.e., without training the model a-priori), and (7) the prediction accuracies for an increasing number of query points processed over changing costs. For the real UDFs, we consider two cost metrics: CPU cost, measured as the time spent on the CPU, and disk IO cost, measured as the number of physical disk pages fetched into the buffer. Cost metrics are irrelevant for synthetic UDFs.

7.2.1 *Experiment 1: prediction accuracy for various query distributions and UDFs/datasets.* In this experiment, we compare the prediction accuracies across different UDFs/datasets and query distributions. Figure 19 shows the prediction errors for the CPU costs of the real UDFs/datasets. (The results for the disk IO costs will appear in Section 7.2.4.) From the figure, we make three observations. First, both MLQ and MLKNN outperform both SH techniques in a majority (15 out of 18) of the test cases of query distributions and real UDFs/datasets. Second, both MLQ and MLKNN perform close to KNN in a majority (14 out of 18) of the test cases. This is impressive, considering that KNN retains all the data points that have been inserted into the model. Third, between MLKNN and MLQ, MLKNN outperforms MLQ in a majority (13 out 18)

[6] 2500 query points are sufficient in our experiments because, as will be shown in Figure 25, the prediction error reaches its minimum well before the 2500^{th} query point (except for KNN which does not compress the model).

of cases. This indicates the advantage of an instance-based technique over a summary-based technique for prediction accuracy.



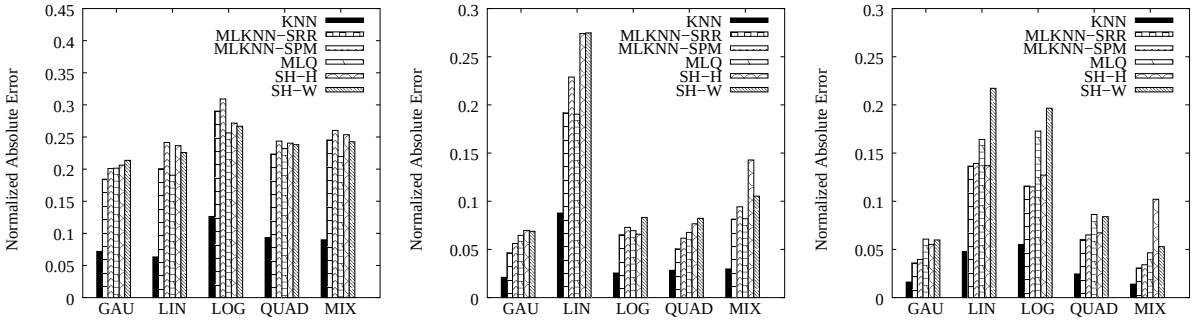
(a) Uniform query distribution

(b) Gaussian-random query distribution

(c) Gaussian-sequential query distribution

Fig. 19. Prediction accuracy for various real UDFs/datasets.

Figure 20 shows the prediction errors for the CPU costs of the synthetic UDFS/datasets. The UDFs/datasets are generated using the following decay functions: linear (LIN), Gaussian (GAU), log of base 2 (LOG), quadratic(QUAD), and a random mixture of them (MIX). From the figure, we make observations similar to those for the real UDFs/datasets, except that KNN outperforms the other techniques by a larger margin.



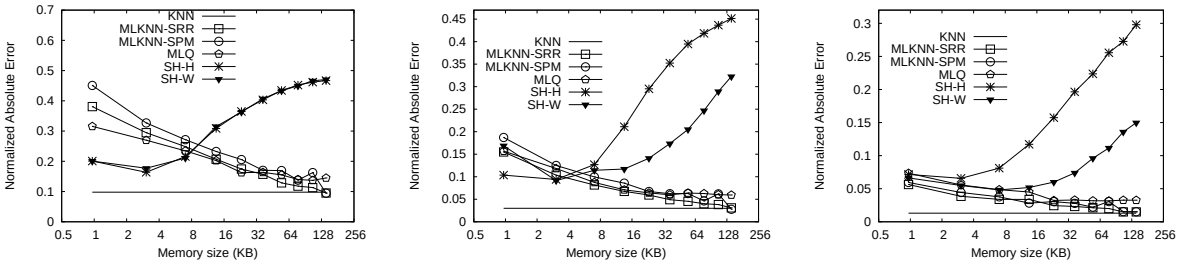
(a) Uniform query distribution

(b) Gaussian-random query distribution

(c) Gaussian-sequential query distribution

Fig. 20. Prediction accuracy for varying decay functions (for synthetic data).

7.2.2 *Experiment 2: prediction accuracy for varying memory size.* In this experiment, we compare the influences of memory limitation on the prediction accuracies. Figure 21 shows the prediction errors for the costs of the synthetic UDF/dataset as the memory size increases from 1 KB to 128 KB. This experiment does not apply to KNN, which does not limit the amount of memory; We thus show the normalized absolute error of KNN as a flat line in the figure.



(a) Uniform query distribution

(b) Gaussian-random query distribution

(c) Gaussian-sequential query distribution

Fig. 21. Prediction accuracy for a varying memory size (for synthetic data).

In the figure, we see that, initially, the performances of both SH techniques stay the same or improve slightly as memory size increases, but then starts deteriorating after a certain point. In order to explain the reason for this trend, we need to first describe the relationship between the memory size (m) and the grid resolution (r) of the two types (i.e., SH-W, SH-W) of histograms. That is,

$$m = \begin{cases} d(r-1)f + r^d f & \text{for SH-H} \\ r^d f & \text{for SH-W} \end{cases} \tag{23}$$

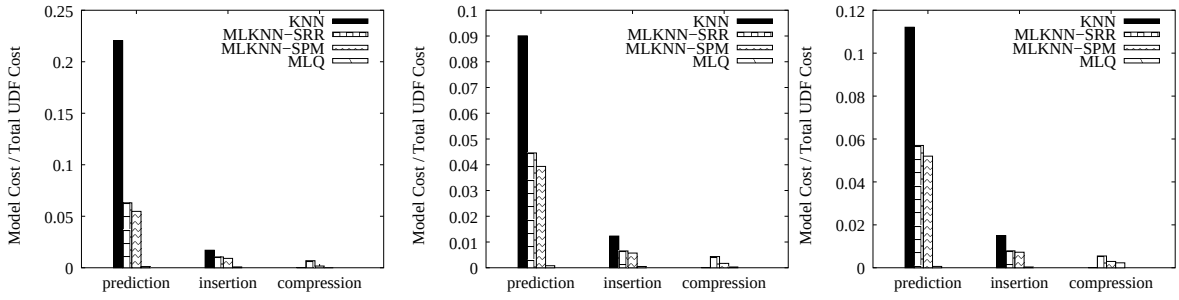
where d is the dimensionality of the model space, r is the grid resolution (i.e., the number of partitions in each dimension), and f is the size of a floating point number. The first term of Equation 23 (for SH-H) refers to the size of memory for storing the coordinates of grid partition boundaries, and the second term of Equation 23 (for SH-H) refers to that for storing the average cost in each bucket of the histogram. Note that SH-H needs both terms, whereas SH-W needs only the second term because its bucket width is fixed. (Equation 23 for SH-H and SH-W are the same as those used in Bruno et al. [2001] to determine the grid resolution for a given a memory size in a static grid-based histogram.)

Now, we can explain the reason for the trend. From Equations 23 for SH-H and SH-W we see that an increase of the memory size (m) leads to an increase of in the grid resolution (r). This in turn leads to an increase in the number of empty buckets in the histogram. In the SH techniques, when a query point maps into an empty bucket, the cost is estimated using the average cost of all data points in the training data set. This obviously causes a significant prediction error once the number of empty buckets exceeds a certain value. In contrast, KNN-based techniques and MLQ do not have such a problem. In the case of the KNN-based techniques, as the memory size increases, they can store more data points and, consequently, find closer neighbors. This improves the prediction accuracy. In the case of MLQ, if a query point maps into an empty node, then it can predict the cost using the immediate parent node or an ancestor node at only a few level higher than the empty node and, consequently, avoid such a significant prediction error as in the

SH techniques. Thus, the prediction errors decrease consistently with increasing memory size for both the KNN-based techniques and MLQ.

7.2.3 *Experiment 3: modeling costs.* In this experiment, we compare the overheads of cost modeling. We show the overheads as the *ratio* of the modeling costs to the execution costs of a real UDF. We use this metric since it allows us to view the overhead in terms of the amount of time spent executing the real UDF. (We do not show the results for the synthetic UDFs/datasets because they have no notion of execution costs.) This experiment is not applicable to SH due to its static nature and, therefore, we compare only among the KNN-based and MLQ techniques.

Figure 22 shows the modeling costs. The results are similar between the CPU costs and the disk I/O costs, and the figure is for the CPU costs. The modeling costs shown are those accumulated over all test query points separately for the prediction costs, insertion costs, and compression costs. Results from the other real UDFs/datasets and results from the other query distributions show similar trends.



(a) Real UDFs/datasets (STS) (b) Real UDFs/datasets (PTS) (c) Real UDFs/datasets (WIN)

Fig. 22. Total modeling cost (using uniform query distribution).

We make three observations from the figure. First, the total modeling costs (prediction + insertion + compression) of the MLKNN and MLQ techniques are between 0.1% and 8% of the execution costs of the real UDFs/datasets. We believe these overheads are within acceptable limits. Second, the prediction costs of both MLKNN techniques are significantly lower than that of KNN. This is because MLKNN stores much fewer data points and, thus, incurs much fewer distance computations when searching for the K nearest neighbors.

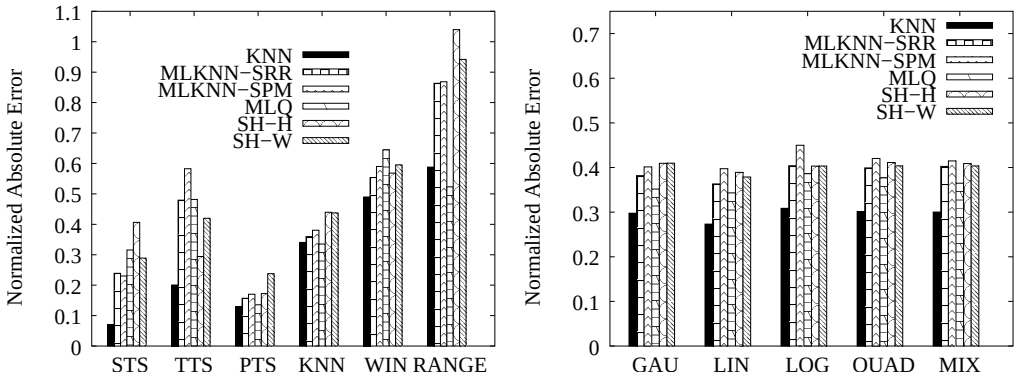
Third, the prediction costs of MLQ are much lower than those of both MLKNN techniques. This comes from the inherently lower cost of a quadtree search compared with the cost of a K-nearest neighbor search.

7.2.4 *Experiment 4: prediction accuracy in the presence of noise.* In this experiment, we compare the prediction accuracies after deliberately "injecting noise" simulating the caching effects. We focus on the disk I/O costs because they are more susceptible to the caching effect than the CPU costs.

Noises are injected in the following manner. For the synthetic UDFs/datasets, we return a random value instead of the true value with 80% probability for each query point. (Results from using other probabilities show similar relative performances among the different techniques.) The true value is calculated by adding the contributions from the decay functions, and the random value is generated using the uniform distribution in the range between 0.0 and the true value (inclusive). Here, the true value simulates the number of disk

pages *accessed*, and the random value simulates the number of disk pages actually *fetched* into main memory. For the real UDFs/datasets, we flush a random portion of the database buffer cache with 20% probability. (Results from using other probabilities show the same relative performances.)

Figures 23(a) and 23(b) show the results from using the real UDFs/datasets and synthetic UDFs/datasets, respectively, for query points with the uniform distribution. (Results for the query points of other distributions show similar relative performances.) We see that the normalized absolute errors of the two MLKNN techniques and MLQ are lower than or similar (within 0.1) to those of the two SH techniques in a majority of UDFs/datasets, specifically, five out of six real UDFs/datasets and all six synthetic UDFs/datasets. This indicates the merit of automatically adjusting the values of K (in MLKNN) and T_{ms} (in MLQ) to the level of noise (as described in Section 5.2 and Section 6.3).



(a) Real UDFs/datasets

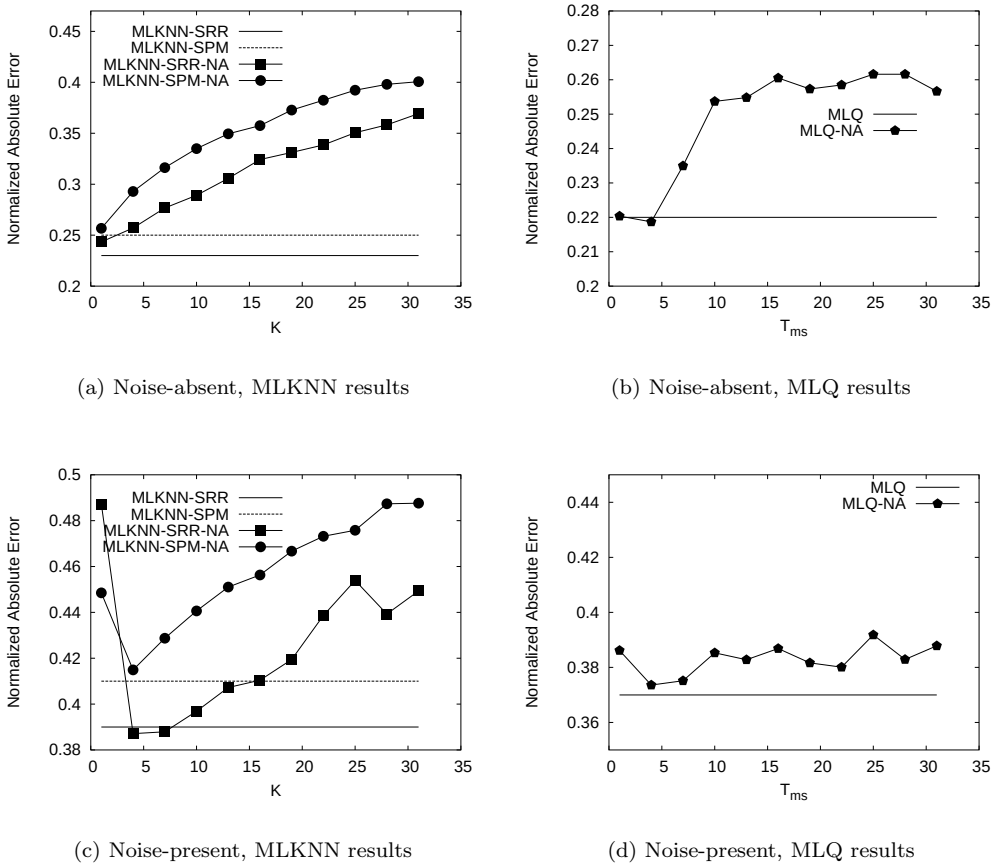
(b) Synthetic UDFs/datasets

Fig. 23. Prediction accuracy in the presence of noise (using uniform query distribution).

7.2.5 *Experiment 5: prediction accuracy for varying K and T_{ms}.* As described in Section 5.2 and Section 6.3, the parameters K (for MLKNN) and T_{ms} (for MLQ) are used to adjust to a varying level of noise, namely, they are "noise-tuning parameters". In this experiment, we examine the effects of varying the values of these parameters on the prediction accuracy and observe how effective the cost prediction algorithms are in automatically finding the optimal values of the parameters.

Figure 24 shows the results obtained using the synthetic UDFs/data sets – specifically, Figures 24(a) and 24(b) when there is no noise and Figures 24(c) and 24(d) when there *is* noise. (The results from using the real UDFs/datasets show similar trends.) The noise is injected in the same manner as in Section 7.2.4. In the figure, the techniques with "-NA" in their labels do not adjust the parameters (i.e., K, T_{ms}) to the noise level but use fixed values. Note that the values of the parameters are irrelevant to the performances of techniques that do the adjustment (i.e., with no "NA" in the label). We, thus, show their prediction errors as flat lines[7]. We observe from the figure that the prediction accuracies of both MLKNN and MLQ with noise tuning are (nearly) the minimum among those achieved for different values of parameters without noise tuning. This indicates that the automatic noise tuning works well.

[7]In the actual measurements, the results show slight fluctuations because they are obtained as an average from using five



(a) Noise-absent, MLKNN results

(b) Noise-absent, MLQ results

(c) Noise-present, MLKNN results

(d) Noise-present, MLQ results

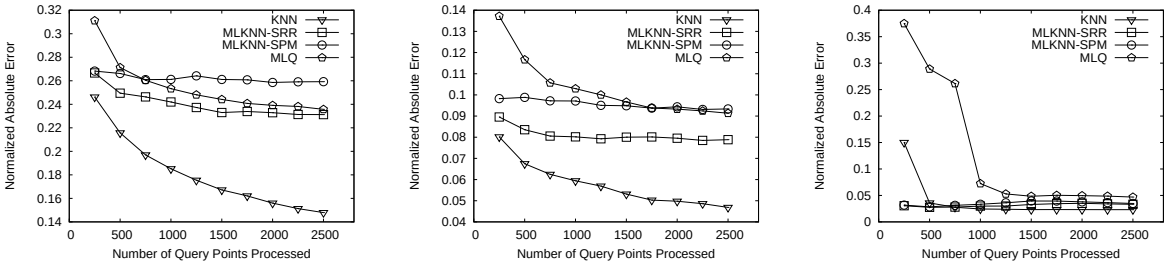
Fig. 24. Prediction accuracy for varying K and T_{ms} (using uniform query distribution).

7.2.6 *Experiment 6: prediction accuracy for an increasing number of query points processed from a cold start.* The objective of this experiment is to see how the prediction error changes as the number of query points processed increases (from a cold start). This experiment is not applicable to SH because it is not dynamic.

Figure 25 shows the prediction accuracy for the synthetic UDFs/datasets using query points with the uniform, Gaussian-random, and Gaussian-sequential distributions. ¿From the figure, we make three observations. First, the prediction accuracies of all the memory limited algorithms converge to the highest values before or by around the 1000^{th} query. Second, the prediction errors for the MLKNN algorithms stay relatively constant with the number of query points processed, whereas the prediction error for MLQ tends to decrease continuously. This indicates that the MLKNN algorithms, being instance-based, consume the available memory quicker than the summary-based MLQ; This in turn causes the MLKNN algorithms to compress the model earlier than MLQ, after which the prediction accuracy stays relatively constant. Third,

different training and testing query sets (randomly generated using the same distribution).

for KNN, the prediction error keeps decreasing as the number of query points processed increases. This is because KNN inserts all query points as new data points and never compresses the model; Therefore, KNN always finds closer K nearest neighbors as more data points are inserted, which results in a lower prediction error.



(a) Uniform query distribution

(b) Guassian-random query distribution

(c) Guassian-sequential query distribution

Fig. 25. Prediction error with an increasing number of query points processed from a cold start (using synthetic UDFs/datasets).

7.2.7 *Experiment 7: prediction accuracy for an increasing number of query points processed over a change of cost.* The objective of this experiment is to see how the prediction error changes as the UDF cost changes. As in Experiment 6, this experiment is not applicable to SH because it is not dynamic. We simulate the change of costs by adding some random variations to the true costs of data points (calculated by adding the contributions from the decay functions). The range of variations used is 50% to 150% of the true costs. Specifically, we first train the model a-priori with queries against the randomly varied costs and, then, test the model using queries against the true costs.

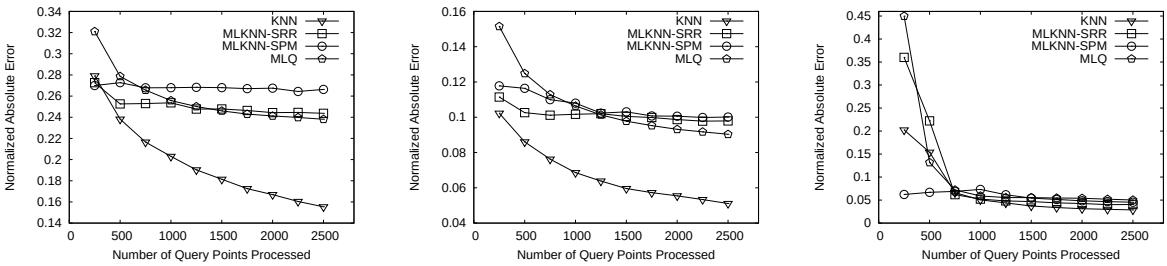
Figure 26 shows the prediction accuracy for the synthetic UDFs/datasets using query points with the uniform, Gaussian-random, and Gaussian-sequential distributions. The results are very similar to those obtained in Experiment 6. That is, the prediction accuracies of all the memory limited algorithms converge to the highest values in about the same number of queries (between 500 and 1000 queries) and the relative prediction errors among different memory limited algorithms look the same.

8. CONCLUSIONS

8.1 Summary

In this paper, we have addressed modeling the executions costs of user-defined functions using self-tuning techniques, with the ORDBMS query optimizer as the main application. For this purpose, first we have proposed a set of guidelines designed to develop the techniques that make fast and accurate predictions while incurring small model update costs under the constraints of limited memory, limited computation time, and fluctuating costs.

Then, we have presented two concrete techniques, MLKNN and MLQ, developed following the guidelines. MLKNN is an instance-based technique. It stores selected query points as data points into a model. The model used in MLKNN consists of a multi-dimensional index tree (particularly the R*-tree) and a data



(a) Uniform query distribution

(b) Guassian-random query distribution

(c) Guassian-sequential query distribution

Fig. 26. Prediction error with an increasing number of query points processed over a change of costs (using synthetic UDFs/datasets).

structure containing information about the data points stored. MLQ is a summary-based technique. It uses the quadtree for maintaining a summary of inserted data points at multiple resolution levels. The model used in MLQ consists of the quadtree and the summary values (i.e., count, sum, sum of squares) stored in each node of the quadtree. For each of these two techniques, we have presented algorithms for predicting the cost at a query point while adjusting to the level of noise (i.e., cost fluctuation), inserting a query point into the model as a new data point, and compressing the model when the memory limit is reached.

We have demonstrated the merits of MLKNN and MLQ through experiments conducted using both various real and synthetic UDFs/datasets and using query points with various distributions. The results show that, first, MLKNN and MLQ techniques achieve higher prediction accuracies than SH – the only existing UDF cost modeling techniques usable in a query optimizer – in most test cases; Second, both MLKNN and MLQ incur modeling costs amounting to only 0.1% to 8% of the execution costs of the real UDFs/datasets; Third, between MLKNN and MLQ, MLKNN is more accurate but MLQ incurs lower modeling costs while it appears MLQ shows relatively better performance overall.

8.2 Open issues for practical applicability

The framework proposed in this paper leaves some open issues to enhance the practical applicability. We summarize them here.

Currently, users are required to identify the model variables of a UDF. As mentioned in Section 1.2, model variables are those that predominantly influence the UDF execution costs. Moreover, sometimes they are not input arguments of the UDF but variables resulting from user-defined transformations. Thus, in order to find appropriate model variables, users need to know the semantics of the UDF, and in some cases high-level understanding of an internal structure used (e.g., text inverted index), as discussed using example real UDFs in Section 7.1.3. It will be useful to provide users with a tool that facilitates this process.

If the cost fluctuates (e.g., due to caching) too quickly, the techniques may not adapt to the change fast enough. (As shown in the experiment in Section 7.2.7, it took 500 to 1000 queries for the tested algorithms to adapt to changes in costs.) It will be interesting to improve the techniques to be more agile to a rapid change of cost.

Currently the techniques proposed in this paper do not support model variables that are nominal (or categorical). It would be useful to extend the techniques to support nominal variables. One (somewhat

naive) approach is to build separate models, each involving ordinal model variables only, for different values of such nominal variables. If the cardinality of a nominal variable is too high, then only the models for recently-used values can be kept. This simple idea has proved adequate in our previous work [Lee et al. 2003] for the UDFs used. Further study will be needed in a more general case.

The "curse of high dimensionality" will not be so forgiving to either KNN-based methods or quadtree-based methods; both run-time overhead and storage space overhead may become very high as the dimensionality of model space increases. It will be thus worthwhile to address this issue using dimensionality reduction techniques, such as principal component analysis(PCA)[Jolliffe 1986] and multidimensional scaling(MDS)[Morrison et al. 2003]. The challenge in this case will lie in adapting the existing techniques to a *dynamic* modeling environment.

8.3 Future work

For our immediate further work, we plan to extend MLKNN and MLQ to use the same model data structure (e.g., R*-tree, quadtree) for multiple UDFs instead of one. This is likely to improve the efficiency of utilizing the system resources. Additionally, we have identified two areas for future work. The first one is to use the guidelines to develop additional cost modeling techniques. This may produce techniques more effective under the constraints mentioned above. The second one is to apply MLKNN and MLQ to other applications like estimating program execution costs for job scheduling in parallel and distributed systems.

As mentioned in the Introduction, query optimization involves selectivity estimation of a UDF predicate as well as cost estimation. In this regard, a framework for selectivity estimation would be useful. Better yet, a unified framework for both selectivity and cost estimations of UDF predicates would be desirable. This also remains as our future work.

It would be interesting to see the impact of our cost modeling techniques on actual query execution costs. This can be done by comparing the query execution times measured with and without using our cost model. Conducting these experiments is another important future work.

Acknowledgments

We would like to thank Li Chen for her contributions to the coding of the experimental framework used for this paper. We thank Songtao Jiang and David Van Horn for setting up the real UDFs/datasets used in the experiments, Xindong Wu and Hill Zhu for insightful technical discussions, Marios Hadjieleftheriou for making the R*-tree code freely available on the Internet, and the Reuters Limited for providing Reuters Corpus, Volume 1, English Language, "Reuters Corpus, Volume 1, English language, 1996-08-20 to 1997-08-19" for use in the experiments. This research has been supported by the National Science Foundation through Grant No. IIS-0415023 and the US Department of Energy through Grant No. DE-FG02-ER45962.

Appendix A

Here we show the derivation of Equation 22 from Equation 21. We first give an equation needed for the derivation.

$$SSENC(p_{ac}) = SSENC(p_{bc}) + \sum_{i=0}^{C(b)} (V_i - AVG(p))^2 \tag{24}$$

where p is the parent block of b (the block being removed) and V_i is the i^{th} data point that maps into block b. Equation 24 is interpreted as that the $SSENC$ of block p after the compression equals the sum of the $SSENC$ before the compression and the new SSE introduced by the compression.

We now show the derivation of Equations 22 from Equation 21 and 24. Note that, $AVG(p)$ is the same before and after the compression.

$$TSSENCG(b) = SSENC(p_{ac}) - (SSENC(b) + SSENC(p_{bc}))$$

Using Equation 24, this can be written as

$$SSENC(p_{ac}) - (SSENC(b) + SSENC(p_{bc})) = \sum_{i=0}^{C(b)} (V_i - AVG(p))^2 - SSENC(b) \quad (25)$$

Since block b is for a leaf node, which has no child node, $b_{ac} = b$. Hence, using Equation 18, Equation 25 can be rewritten as:

$$TSSENCG(b) = \sum_{i=0}^{C(b)} (V_i - AVG(p))^2 - \sum_{i=0}^{C(b)} (V_i - AVG(b))^2 \quad (26)$$

It is straightforward to derive Equation 22 from Equation 26. We show some intermediate steps here.

$$\begin{aligned} TSSENCG(b) &= C(b)SS(b) - 2S(b)AVG(p) + C(b)(AVG(p))^2 - C(b)SS(b) + 2S(b)AVG(b) - C(b)(AVG(b))^2 \\ &= C(b)(AVG(p))^2 - 2AVG(b)C(b)((AVG(p) - AVG(b)) - C(b)(AVG(b))^2 \\ &= C(b)((AVG(p))^2 - 2AVG(b)AVG(p) + (AVG(b))^2) \\ &= C(b)(AVG(p) - AVG(b))^2 \end{aligned} \quad (27)$$

REFERENCES

Aboulnaga, A. and Chaudhuri, S. 1999. Self-tuning histograms: building histograms without looking at data. In *Proceedings of the 1999 ACM International Conference on Management of Data (SIGMOD 99)*. 181–192.

Beckmann, N., Kriegel, H., Schneider, R., and Seeger, B. 1990. The R*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 90)*. 322–331.

Bogartz, R. S. 1994. *An Introduction to the Analysis of Variance*. Praeger Publishers.

Boulos, J. and Ono, K. 1999. Cost estimation of user-defined methods in object-relational database systems. *SIGMOD Record 28,* 3, 22–28.

Boulos, J., Viemont, Y., and Ono, K. 1997. A neural network approach for query cost evaluation. *Transactions in Information Processing Society of Japan 38,* 12, 2566–2575.

Bruno, N., Chaudhuri, S., and Gravano, L. 2001. STHoles: a mulidimensional workload-aware histogram. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 01)*. 211–222.

Buccafurri, F., Furfaro, F., Sacca, D., and Sirangelo, C. 2003. A quad-tree based multiresolution approach for two-dimensional summary data. In *Proceedings of the 15th International Conference on Scientific and Statistical Database Management (SSDBM 2003)*. Cambridge, Massachusetts, USA, 127–140.

Chang, C. L. 1974. Finding prototypes for nearest neighbor classifiers. *IEEE Transactions on Computers C-23,* 11, 1179–1184.

Chaudhuri, S. 1999. Self-tuning databases and application tuning. *IEEE Data Engineering Bulletin 22,* 2, 3–46.

Chaudhuri, S., Christensen, E., and Graefe, G. 1999. Self-tuning technology in microsoft sql server. *IEEE Data Engineering Bulletin 22,* 2, 20–26.

Chaudhuri, S. and Shim, K. 1999. Optimization of queries with user-defined predicates. *ACM Trans. Database Syst. 24,* 2, 177–228.

Chen, M. C. and Roussopoulos, N. 1994. Adaptive selectivity estimation using query feedback. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 94)*. 161 – 172.

Cheung, K. L. and chee Fu, A. W. 1998. Enhanced nearest neighbour search on the R-tree. *SIGMOD Record 27,* 3, 16–21.

Cui, B., Ooi, B. C., Su, J., and Tan, K.-L. 2003. Contorting high dimensional data for efficient main memory knn processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 03)*. 479 – 490.

Deshpande, A., Garofalakis, M., and Rastogi, R. 2001. Independence is good: dependency-based histogram synopses for high-dimensional data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 01)*. 199–210.

Evangelidis, G., Lomet, D., and Salzberg, B. 1997. The hb -tree: A multi-attribute index supporting concurrency, recovery and node consolidation. *The VLDB Journal 6*, 1–25.

Guttman, A. 1984. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 84)*. 47–57.

Han, J. and Kamber, M. 2001. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, Chapter 7, 303, 314–315.

Hart, P. E. 1968. The condensed nearest neighbor rule. *IEEE Transactions on Information Theory IT-14,* 3, 515–516.

He, Z., Lee, B. S., and Snapp, R. 2004. Self-tuning UDF cost modeling using the memory limited quadtree. In *Proceedings of the International Conference on Extending Database Technology (EDBT 04)*. 513–531.

Hellerstein, J. 1994. Practical predicate placement. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 94)*. 325–335.

Hellerstein, J. and Stonebraker, M. 1993. Predicate migration: Optimizing queries with expensive predicates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 93)*. 267–276.

Hellerstein, J. M. 1998. Optimization techniques for queries with expensive methods. *ACM Trans. Database Syst. 23,* 2, 113–157.

Jiang, S., Lee, B. S., and He, Z. 2003. The cost modeling of spatial operators using nonparametric regression. Tech. Rep. CS-03-17, Department of Computer Science, University of Vermont. (submitted for publication).

Jolliffe, I. 1986. *Principal component analysis*. Springer-Verlag.

Kim, K., Cha, S., and Kwon, K. 2001. optimizing multidimensional index trees for main memory access. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 01)*. 139–150.

Lazaridis, I. and Mehrotra, S. 2001. Progressive approximate aggregate queries with a multi-resolution tree structure. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 01)*. 401–413.

Lee, B. S., Chen, L., Buzas, J., and Kannoth, V. 2004. Regression-based self-tuning modeling of smooth user-defined function costs for an object-relational database management system query optimizer. *The Computer Journal 47,* 6 (November), 673–693.

Lee, B. S., Kannoth, V., and Buzas, J. 2003. A statistical cost-modeling of financial time series functions for an object-relational DBMS query optimizer. Technical Report CS-03-10, Department of Computer Science, University of Vermont, Burlington, VT 05405. March. (submitted for publication).

Lee, M., Kitsuregawa, M., Ooi, B., Tan, K., and Mondal, A. 2000. Towards self-tuning data placement in parallel database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 00)*. 225–236.

Lomet, D. B. and Salzberg, B. 1990. The hB-tree: A multiattribute indexing method with good guaranteed performance. *ACM Transaction on Database Systems 15,* 4, 625–658.

Morrison, A., Ross, G., and Chalmers, M. 2003. Fast multidimensional scaling through sampling, springs and interpolation. *Information Visualization 2,* 1, 68–77.

Nardelli, E. and Proietti, G. 1994. A hybrid pointerless representation of quadtrees for efficient processing of window queries. In *Proceedings of the International Workshop on Advanced Information Systems: Geographic Information Systems*. 259–269.

Pennsylvania. Last viewed:6-18-2003. PSADA - Data Download - Urban Areas. URL:http://www.pasda.psu.edu/access/urban.shtml.

Poosala, V. and Ioannidis, Y. 1997. Selectivity estimation without the attribute value independence assumption. In *Proceedings of the 23th International Conference on Very Large Data Bases (VLDB 97)*. 486–495.

Procopiuc, O., Agarwal, P. K., Arge, L., and Vitter, J. S. 2003. Bkd-tree: a dynamic scalable kd-tree. In *Proceedings of the 8th International Symposium on Spatial and Temporal Databases*. 46–65.

Rahal, A., Zhu, Q., and Larson, P.-A. 2004. Evoltionary techniques for updating query cost models in a dynamic multi-database environment. *The VLDB journal 13,* 2, 162–176.

Robinson, J. T. 1981. The K-D-B-tree: A search structure for large multidimensional dynamic indexes. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD 81)*. 10–18.

Stillger, M., Lohman, G., Markl, V., and Kandil, M. 2001. LEO - DB2's LEarning optimizer. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB 01)*. 19–28.

Stone, C. J. 1977. Consistent nonparametric regression. *The Annals of Statistics 5,* 4, 595–645.

Thorburn, W. M. 1915. Occam's razor. *Mind 24*, 287–288.

Tousidou, E. and Manolopoulos, Y. 2000. A performance comparison of quadtree-based access methods for thematic maps. In *Proceedings of the 2000 ACM Symposium on Applied Computing*. 381–388.

VanHorn, D., Lee, B. S., Buzas, J., and Thompson, P. 2003. Metadata-based generation of statistical cost functions for text search. Tech. Rep. CS-03-13, Department of Computer Science, University of Vermont.

Wand, M. P. and Jones, M. C. 1995. *Kernel smoothing monographs on statistics and applied probability.* Chapman & Hill.

Wilson, D. L. 1972. Asymptotic properties of nearest neighbor rules using edited data. *IEEE Transactions on Systems, Man and Cybernetics SMC-2,* 4, 408–421.

Yu, C., Ooi, B. C., Tan, K.-L., and Jagadish, H. V. 2001. Indexing the distance: an efficient method to KNN processing. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB 01)*. 421–430.

Zipf, G. K. 1949. *Human behavior and the principle of least effort.* Addison-Wesley.