Regression-Based Self-Tuning Modeling of Smooth User-Defined Function Costs for an Object-Relational Database Management System Query Optimizer

Byung S. Lee¹, Li Chen¹, Jeff Buzas², Vinod Kannoth³
¹Department of Computer Science, University of Vermont, Burlington, VT 05405, USA
²Department of Mathematics and Statistics, University of Vermont, Burlington, VT 05405, USA
³Federal Reserve Information Technology, 701 East Byrd Street, Richmond VA 23219, USA
E-mails: {bslee, lchen, buzas}@emba.uvm.edu, vinod.kannoth@frit.frb.org

Abstract

We present a new approach to modeling the execution costs of user-defined functions (UDFs) for the query optimizer of an object-relational DBMS (ORDBMS). Our approach self-tunes a cost model incrementally based on the costs of the recent executions of a UDF. The approach is centered on a feedback loop in which the feedback information comprises individual UDF execution records. Each execution record contains the cost variable values used in the execution and the resulting CPU and disk I/O costs. This feedback information is saved in the execution log and used in a batch to update the cost model. Furthermore, our approach handles nominal cost variables by maintaining separate cost models for recently used values of the variables. We have built a framework that implements the feedback loop in a commercial ORDBMS. Then, we have performed experiments using common database UDFs with smooth cost variations and incrementally modeling the data using multiple regression. The experimental results demonstrate the adaptive accuracy that makes the cost model stabilize quickly while incurring small errors in cost estimations. Our approach has the advantages of incurring little overhead while tuning the cost model continuously throughout the UDF executions.

Keywords: object-relational database system, user-defined function, query optimization, cost estimation, multiple regression

1 Introduction

1.1 Motivation

The objective of a cost-based query optimization is to choose an efficient query execution plan, which involves systematically estimating the costs of alternative execution strategies using predefined cost functions and selectivity functions. In this regard, the availability and accuracy of these two functions are crucial to an efficient query processing.

Today’s object-relational database management systems (ORDBMSs) support complex database application by allowing users to define their own functions, or user-defined functions (UDFs), and use them as if they were built-in functions. If these UDFs are specified in the query condition (e.g., “where UDF¹(args¹) op¹ const¹ AND UDF²(args²) op² const²”), the cost-based query optimizer needs the cost functions and selectivity functions of the UDFs to determine the order of predicate evaluations. Unfortunately, however, *The corresponding author, e-mail: bslee@cs.uvm.edu, phone: (802)656-1919, fax: (802)656-0696*
those functions cannot be known at the time the DBMS is developed. Therefore, the responsibility is passed on to the users who develop the UDFs. This task, however, is overwhelming for most users. At least they need a tool that facilitates the task.

1.2 Problem formulation

This paper concerns generating the cost functions of database UDFs. This issue has not been addressed actively, although several researchers addressed ordering predicates involving UDFs (i.e., UDF predicates) provided with their cost functions [1, 2, 3]. Presumably, this is because of the complexity of the problem. The only published one we find is a preliminary work done by Boulos and Ono [4], and we have done a similar work as well [6]. Both works use the “parade-of-runs” (PoR) approach. This approach executes a UDF repeatedly for different combinations of the sample values of the variables influencing the cost (called “cost variables”) and generates the cost data coupled with the cost variable values used. Then, a data analysis builds a cost function by applying a data reduction technique [7] to the generated cost data sets. For this purpose, a statistical regression model is used in [6] whereas a multi-dimensional histogram in [4].

This PoR approach is simple, hence easy to implement, and generates a fairly precise cost model provided with a suitable modeling technique and a sufficiently large cost data set. However, it has a number of problems. First, the computational overhead increases exponentially with the number of cost variables because a UDF is typically executed for every combination of the values of all cost variables. This overhead may be significant enough to render the approach impractical. Second, the generated cost function is fixed and, therefore, does not adapt to the changes of the environments like the data statistics (e.g., the number of tuples, the number of distinct column values, index height) and the system configurations (e.g., buffer size, blocking factor). Third, it assumes that there exists a finite range of the values of each cost variable. This assumption is not valid in general. Moreover, the generated cost function is not valid outside the subrange chosen by a user. Last, nominal variables are excluded from consideration. In theory we can build separate cost functions for different values of a nominal variable. However, this is infeasible in practice unless the cardinality is of a manageable size. In this paper we propose a novel approach that resolves these problems.

1.3 Objective and our approach

The objective of our approach is two-fold: (1) to dispense with the parade of runs and (2) to facilitate handling nominal cost variables. The first objective is met by updating a cost function incrementally based on the actual costs of recent UDF executions, and the second objective is met by building separate models for only the recently used values of a nominal variable. Thus, our approach accomplishes a self-tuning modeling (STM) of the costs.

We build a cost function as a statistical regression model as in [6]. Initially, no cost model is available for a new UDF and, therefore, default values are used for cost estimation by the query optimizer. Then, each time the UDF is executed, its costs, i.e., the CPU time and the number of fetched disk pages (or, disk I/O count), are captured and written to an execution log. (The logging incurs negligible overhead and can be done in the background.) The logged cost data are then used to build a new cost model or refine an existing cost model. Thus, the cost model adapts to the recently-logged cost data. This adaptive model building progresses incrementally as illustrated in Figure 1. As a new batch of data is added, a cost model is adjusted incrementally based on the entire data set from the past. The effects of old data diminishes as the iteration progresses, and eventually the model becomes stable as sufficient data are considered. If a change occurs in the costs afterward, the model adapts again incrementally.

Our experience suggests that a nontrivial number of common database UDFs show smooth cost variations with respect to the cost variables. (Thus far, we have used various financial time series functions, text search functions, and spatial search functions.) Hence, this paper focuses on those UDFs. For these UDFs, we choose
parametric regression, particularly multiple regression, as the modeling technique. Multiple regression is simple and yet adequately precise for data varying reasonably smooth. More importantly, it allows for an incremental update of the model with additional data, thus allowing for discarding the data while keeping only the new model. Its disadvantage could be the risk of overfitting or underfitting the data, but this is not a serious problem in our case because query predicate ordering is quite tolerant of UDF cost estimation errors.

1.4 Experimental summary

We have built a framework that implements the STM approach. It has been built using a commercial ORDBMS Oracle\(^1\) while leveraging the extensible query optimization capabilities available through its Data Cartridge\(^2\) interfaces. We have selected two kinds of experimental UDFs: aggregate financial time-series functions and keyword-based text-search functions. A full quadratic regression model suffices for these UDFs because their costs vary smoothly and monotonously with the values of cost variables. A higher order model may well be used for cost variations showing more than one extrema.

The experimental results show that the median relative error of cost estimation is within 20\% for the time-series UDFs and within 40\% for the text-search UDFs on average. These are quite accurate considering the unpredictable adverse effects of data caching in the buffer. Typically the CPU time is estimated more accurately than the disk I/O counts because the CPU time is not affected by data caching. The cost models become stable quickly in a few feedback cycles with 50 data points in each batch of the logged cost data set.

1.5 Contributions

This paper makes three main contributions. First, it proposes a new method for providing the cost functions of database UDFs. As far as we know, this paper is the first one addressing incremental and adaptive cost modeling of UDFs. Second, it demonstrates the practicality of the proposed approach by incorporating it into a commercial ORDBMS and using real UDFs supported by the ORDBMS. Third, it presents an incremental model update approach using multiple regression. To our knowledge, this kind of model update is a new technique.

1.6 Organization of the paper

The rest of the paper is organized as follows. Section 2 provides some background information. Section 3 describes the specifics of our approach. Section 4 presents the experiments and their results. Section 5

---

\(^1\)Oracle is a trademark of Oracle, Inc.  
\(^2\)Data Cartridge is a trademark of Oracle, Inc.
discusses related work, and Section 6 concludes the paper.

# 2 Background

In this section we provide an overview of an extensible query optimizer in an ORDBMS and the PoR approach to UDF cost modeling used in the previous work[6].

## 2.1 ORDBMS’s extensible query optimizers

In a database system, a query typically has many possible execution strategies and a query optimizer chooses the most efficient one. There are two kinds of query optimizers: the rule-based and the cost-based. The rule-based one alone is not sufficient in most database systems. The cost-based query optimizer uses a traditional optimization technique that searches the space of alternative execution plans for one that minimizes the estimated query execution cost. The cost typically consists of the CPU cost, disk I/O cost, and network I/O cost.

If a query specifies UDF predicates, the query optimizer determines the order of evaluating them based on the costs of executing the UDFs and the resulting selectivity of the predicates. The following example query specifies one UDF predicate on the financial time series function ‘median’ and another on the text search function ‘contains’. Users are required to provide the cost functions of median and contains (as well as selectivity functions).

```sql
select c.name, e.name from Employee e, Company c
where e.work-for = c.id
and median(c.ticker, ‘JAN-01-1902’, ‘DEC-31-2002’) > 50
and contains(e.resume, “UNIX and NT”, l) > 0;
```

In order to incorporate the cost functions into its query optimizer, an ORDBMS provides an extensible framework such as Oracle Data Cartridge and DB2 Extender. Three cost functions are needed for each UDF, one for each of the CPU cost, the disk I/O cost, and the network I/O cost. Their cost metrics are generic so the estimated costs are immune to the changes of system workload and environment. For example, Oracle’s extensible query optimizer uses the following metrics: (1) CPU cost as the number of machine instructions executed by the CPU, (2) disk I/O cost as the number of data pages transferred from disk to main memory buffer, and (3) network I/O cost as the number of data packets transmitted via the network. We use the first two cost metrics in our work. The third metric is not considered here because it is not used by any ORDBMS yet.

## 2.2 The parade-of-runs approach

In the PoR approach used in [6], a user determines the cost variables, and either provides a model based on one’s understanding of the UDF or lets the system build a default regression model using the variables. Then, the system calibrates the regression coefficients by fitting the model to a cost data set generated through a parade of runs.

The experiments use aggregate financial time series functions including NthGrpMavg(groupsymbol, startdate, enddate, windowsize, n). Given a group symbol (e.g., NASIND1), NthGrpMavg returns the n-th minimum moving average of the group average time series calculated within a specified range of dates. Here, a group average time series is generated by taking the average of all ticker prices in the same group on each day. The UDF is implemented in Oracle PL/SQL. (A simplified code is shown in Appendix A.2.) Its cost varies smoothly and monotonously with respect to the cost variables, and is modeled precisely using a quadratic regression model.
Determining cost variables
groupsize = cardinality(groupsymbol);
daterange = enddate – startdate;
windowsize;

Mapping cost variables to model variables
Y = cost; X1 = groupsize – mean(groupsize);
X2 = daterange – mean(daterange);
X3 = windowsize – mean(windowsize);

User-defined function
NthGrpMavg(groupsymbol, startdate, enddate, windowsize, n)

Parade of runs
Based on the samples of groupsize, daterange, and windowsize

Cost model
Y = a0 + a1X1 + a2X2 + a3X1X3 + a4X12 + a5X22 + a6X32 + a7X1X2 + a8X2X3 + a9X3X1

Model-based data analysis

Cost data set
{<groupsize,daterange,windowsize,CPU,disk I/O>}

Cost model parameters
a0, a1, a2, a3, a4, a5, a6, a7, a8, a9

Figure 2: Cost modeling using the parade-of-runs approach.
Figure 2 illustrates the process of generating the cost function using the PoR approach while using NthGrpMavg as an example UDF. It first derives three cost variables: groupsize as the number of ticker symbols belonging to the group denoted by groupsymbol, daterange as the interval between startdate and enddate, and windowsize as provided. The input argument ‘n’ has no effect on the execution cost and, therefore, does not derive any cost variable. Then, a parade of runs is performed using the three cost variables to generate a cost data set. In parallel, the cost variables are centered\(^3\) to become the model variables \(X_1, X_2,\) and \(X_3\). Then, by performing a regression analysis on the cost data set, we obtain the values of the model parameters, i.e., the regression coefficients \(a_0\) through \(a_9\).

The experimental results show very small errors (less than 1.5% mean and median relative errors) for both the CPU cost and the disk I/O cost. Besides, the resulting cost functions are easily incorporated into the Oracle query optimizer. However, the overhead of parades of runs is significant, especially because NthGrpMavg is an expensive UDF that involves mergesort. In addition, the experiment does not consider the unpredictable caching effect on the disk I/O cost and, therefore, the accuracy of the disk I/O cost estimation is misleading.

### 3 Self-Tuning Modeling Approach

As mentioned in Section 1.3, the self-tuning modeling (STM) approach replaces the one-time process of parade of runs with a continuous, incremental tuning process based on the most recent runs, and this incremental approach also allows for handling nominal cost variables by building separate models for the most recently used values. In this section, we describe the STM framework with a focus on its feedback loop and elaborate on important modeling issues.

#### 3.1 Overview of the STM framework

Figure 3 shows an overview of the STM framework. The rectangles depict executable modules, the ovals depict data generated and used in main memory, and the drums depict data stored in and retrieved from tables in the database. The XOR in the upper left corner denotes an exclusive-or, meaning “use either the estimated costs or the default costs.”

The STM framework is centered on a feedback loop in which the feedback information comprises individual UDF execution records. Each execution record contains the cost variable values used in the execution and the resulting CPU and disk I/O costs. This feedback information is saved in the execution log and used in a batch to update the cost model parameters. The resulting cost model is then used by the query optimizer to estimate the execution cost.

Formally, let \(B_i\) be a set of UDF execution records \(\{<v_1, v_2, \ldots, v_k, c, d>\}\) used in the \(i\)-th batch, where \(v_1, v_2, \ldots, v_k\) denote the values of cost variables and \(c\) and \(d\) respectively denote the CPU cost and the disk I/O cost. Additionally let \(\beta_i^C\) and \(\beta_i^D\) denote the vectors of regression coefficients for estimating the CPU and disk I/O costs in the \((i+1)\)-th batch, respectively. Then, these costs are calculated as \(\hat{C}_{i+1} = V_{i+1}\hat{p}_i^C\) and \(\hat{D}_{i+1} = V_{i+1}\hat{p}_i^D\), where \(V_{i+1}, \hat{C}_{i+1},\) and \(\hat{D}_{i+1}\) denote the vectors of, respectively, \(<v_1, v_2, \ldots, v_k>\) records used, the estimated CPU costs, and the estimated disk I/O costs in the \((i+1)\)-th batch. Each row of \(V_{i+1}, \hat{C}_{i+1}, \hat{D}_{i+1}\) are stored in a log side by side, and the resulting set of records \(B_{i+1}\) is feedback to upgrade \(\beta_i^C\) and \(\beta_i^D\) to \(\beta_{i+1}^C\) and \(\beta_{i+1}^D\), respectively.

We have built the STM framework using a commercial ORDBMS. In the remainder of this section, we describe its functional components and modeling capabilities.

\(^3\)Centered data reduces the fitting error by reducing the collinearity between power terms (e.g., \(X\) and \(X^2\)). [14]
3.2 Functional components of the STM

The STM has three functional components: UDF cost model registration, UDF execution cost recording, and UDF cost model update. We describe each component in this section.

3.2.1 UDF cost model registration

Figure 4 shows the steps involved in registering a UDF and its cost model.

**Parsing the XML script:** In our implementation, XML is used as the interface language for registering a cost model. Figure 5 shows an example XML script, which contains the same information as in Figure 2 except the parade-of-runs part. That is, it contains general information such as the UDF’s name (e.g., NthGrpMavg).
modeling method (e.g., multiple regression), cost metrics (e.g., CPU time, disk I/O page count) and their default values, as well as model-specific information such as the regression variable (e.g., X2) of each ordinal cost variable (e.g., daterange) and how the variable is determined (e.g., enddate—startdate), model parameters (e.g., regression coefficients a1, a2, etc.) and the derivation methods (e.g., direct, computed) of the associated terms (e.g., X1*X1).

```xml
<?xml version='1.0' encoding='ISO-8859-1' standalone="yes"?>
<!DOCTYPE UDF SYSTEM "file:config.dtd">
<UDF name='NthGrpMavg' method='multiple_regression'>
  <CPU default='2000'>
    <Model>
      <Ordinal name='X1' value='groupsize' type='computed'/>
      <Ordinal name='X2' value='enddate-startdate' type='computed'/>
      <Ordinal name='X3' value='windowsize' type='direct'/>
      <Term coefficient='a0' value='1' type='constant'/>
      <Term coefficient='a1' value='X1' type='direct'/>
      <Term coefficient='a2' value='X2' type='direct'/>
      <Term coefficient='a3' value='X3' type='direct'/>
      <Term coefficient='a4' value='X1*X1' type='computed'/>
      <Term coefficient='a5' value='X2*X2' type='computed'/>
      <Term coefficient='a6' value='X3*X3' type='computed'/>
      <Term coefficient='a7' value='X1*X2' type='computed'/>
      <Term coefficient='a8' value='X2*X3' type='computed'/>
      <Term coefficient='a9' value='X1*X3' type='computed'/>
    </Model>
  </CPU>
  <IO default='1000'>…</IO>
</UDF>
```

Figure 5: An example XML script for registering a cost model.

**Updating the database:** Registering a model involves parsing the XML script and saving information about the cost model in the configuration tables. The information includes the model formula, ordinal cost variables, nominal cost variables, cost model parameters, default costs, and the feedback control information. The feedback control information includes the maximum and minimum numbers of observations required before a model update.

**Generating the statistics object:** In addition, the STM creates a new object type (e.g., NthGrpMavg_stat) having cost functions as members and associate this type with the Statistics object of the ORDBMS. This allows the generated cost functions to be incorporated into the ORDBMS’s extensible query optimizer.

### 3.2.2 UDF execution cost recording

Figure 6 shows the steps involved in recording the execution costs of a UDF during run-time.

**Estimating the execution cost:** While generating a query execution plan, the query optimizer uses the current cost model obtained from the configuration tables to estimate the CPU and disk I/O costs. In case no cost model is available (because initially no model parameters are available), the default costs are used.

**Executing the UDF:** Each time a UDF is executed, the values of cost variables used and the observed CPU and disk I/O costs are recorded in the execution cost log table. These records constitute the feedback information used to adapt the cost models to the recent executions of the UDF. The STM system captures the observed costs by taking snapshots of the execution session immediately before and after the UDF execution and calculating the difference of the CPU and disk I/O usage.
Figure 6: Recording execution costs during the run-time.

Figure 7: Updating a cost model using multiple regression.
3.2.3 UDF cost model update

Figure 7 shows the steps for updating the cost model parameters. Specifically, it shows using multiple regression as the modeling technique.

**Incremental update of cost model parameters:** A new set of cost model parameters are calculated from the parameter values stored in the configuration tables and the new cost data set in the log. The resulting updated parameter values replace the old ones and are available for the query optimizer. Section 3.3.1 describes the algorithm STM uses for this incremental model update.

3.3 Modeling capabilities of the STM

As mentioned in Section 1.3, the STM updates the model incrementally and can handle nominal cost variables. Moreover, the STM deals with two cases of concerns in regression techniques: outliers and multi-collinearity. In this section we elaborate on the techniques the STM has adopted to handle these issues.

3.3.1 Incremental updates of cost model parameters

Our incremental model update algorithm is founded upon the following property of multiple regression coefficients. (We omit the proof of this property.)

**Property 3.1**  Consider a multiple regression model with $p$ coefficients.

\[ Y = X\beta + E \]

Let $<X_1, Y_1>$ be a data set and $\hat{\beta}_1 = (X_1^T X_1)^{-1}X_1^T Y_1$ be the least squares estimate of $\beta$. Then, given an additional data set $<X_a, Y_a>$, the new vector of estimated least square coefficients $\hat{\beta}_{\text{new}}$ is derived as

\begin{equation}
\hat{\beta}_{\text{new}} = (X_1^T X_1 + X_a^T X_a)^{-1}(X_1^T X_1 \hat{\beta}_1 + X_a^T Y_a)
\end{equation}

In the following algorithm, $\hat{C}$ and $\hat{\beta}$ are respectively initialized to all-zero $p \times p$ and $p \times 1$ matrices at start-up and updated every time UpdateModel is invoked.

**Algorithm 3.1 (UpdateModel)**

*Input:*

- Old $\hat{C}$
  - Old vector of estimated regression coefficients $\hat{\beta}$
- Additional data set $<X_a, Y_a>$

*Output:*

- New $\hat{C}$
  - New vector of estimated regression coefficients $\hat{\beta}$

*Begin*

1. $\hat{\beta} := (\hat{C} + X_a^T X_a)^{-1}(\hat{C} \beta + X_a^T Y_a)$;
2. $\hat{C} := \hat{C} + X_a^T X_a$;

*End*

3.3.2 Handling nominal cost variables

Three types of nominal variables are considered in the STM.

- Type 1: Nominal variables that have no effect on the cost. These variables are ignored.
• Type 2: Nominal variables that have effect on the cost but only indirectly as an ordinal variable derived from them. These variables are substituted with the ordinal variable. An example is the startdate and enddate deriving daterange for NthGrpMavg in Section 2.2.

• Type 3: Nominal variables that have random effect on the cost. In this case, separate cost models are built for different values of the variable. That is,

\[
\text{model}(O : n) = \sum_{i \in \text{domain}(n)} I(n = i) \text{model}_i(O)
\]

where \( O \) denotes a set of ordinal variables, \( n \) denotes a nominal variable, and \( I \) denotes a function that returns 1 if \( n = i \) and 0 otherwise. This can be easily extended to the case of multiple nominal variables. If the cardinality of a nominal variable is too large, then only the most recently used values are considered.

Based on these types, we categorize the cost model registration into the following three cases based on the number of models registered and the associated sets of model parameters.

• Case 1: Register one model and one set of parameters. This case is used if there is no nominal cost variable, that is, all nominal variables are of Type 1 or Type 2. In this case, the model may be either a user-provided model or the default model.

• Case 2: Register one model and multiple sets of parameters. This case is used if there are nominal cost variables while no user-provided models are provided. In this case, the default model is used for all possible values of a nominal variable. Each set of parameters is associated with each recently-used value of the variable.

• Case 3: Register multiple models and one set of parameters per model. This case is used if there are nominal cost variables while user-provided models are provided. In general, a user-provided model varies depending on the value of a nominal variable. Hence, one model and one set of parameters are associated with each value of the variable. Furthermore, the default model can be used for nominal variable values not associated with user-provided models.\(^4\)

Case 3 is most general, but it involves user-provided models. If this is not feasible, we resort to Case 2.

### 3.3.3 Removing outliers

Outliers are extreme observations. Under the method of least squares, a fitted equation may be pulled disproportionately towards an outlying observation. For our purpose of cost modeling, an outlier should be detected and removed from consideration. We adopt an approach based on the notion of “semistudentized residuals”\(^5\) [13]. Semistudentized residual is defined as the ordinary residual divided by the root mean square error. The common rule of thumb is to regard a data point as an outlier if the semistudentized residual of the data point is greater than 4.

**Definition 3.1 (Semistudentized residuals)**

Let \( < X, Y > \) denote a data set and \( \hat{\beta} \) denote the vector of estimated regression coefficients based on the data set. Then, given the vector of residuals \( E \)

\[
\hat{E} = Y - \hat{Y} = Y - X \hat{\beta}
\]

\(^4\)Not implemented in our system yet.

\(^5\)We use semistudentized instead of studentized residuals because the latter one requires saving all the previous data.
and the mean square error $\text{MSE}$

$$\text{MSE} = \frac{1}{n - p} \hat{E}^T \hat{E}$$

where $n$ is the number of observations in the data set and $p$ is the number of regression coefficients (in $\hat{\beta}$), the vector of semistudentized residuals is defined as

$$\frac{\hat{E}}{\sqrt{\text{MSE}}} = \left( \begin{array}{c} \frac{y_1 - \hat{y}_1}{\sqrt{\text{MSE}}} \\ \vdots \\ \frac{y_n - \hat{y}_n}{\sqrt{\text{MSE}}} \end{array} \right)$$

where $y_1, y_2, \ldots, y_n$ constitute $Y$.

The following algorithm summarizes the procedure for removing outliers from an additional data set used to update the model. It calculates the semistudentized residual ($\delta$) of each data point in the data set and checks if the resulting value is greater than the threshold.

**Algorithm 3.2 (RemoveOutlier)**

**Input:**
Additional data set $<X_a, Y_a>$ used to update the model
Sum of squared errors SSE and the number of observations $N$

**Output:**
Data set $<X_a, Y_a>$ with no outlier
New SSE and $N$

**Begin**

1. $N_1 :=$ the number of new observations in data set $X_a$
2. $N := N + N_1$
3. $SSE_1 := \sum_{i=1}^{N_1} (Y_i - \hat{Y}_i)^2$; //SSE of the additional data set
4. $SSE := SSE + SSE_1$; //Update SSE for the entire data set.
5. $\text{MSE} := \left( \frac{1}{N - p} \right) SSE$
6. for $i = 1$ to $p$, //Discard if $\delta > 4$
   if $\left| \frac{y_i - \hat{y}_i}{\sqrt{\text{MSE}}} \right| > 4$ then { $X_a := X_a - x_i$; $Y_a := Y_a - y_i$; }

**End**

### 3.3.4 Multi-collinearity

The multi-collinearity problem can happen for a couple of reasons. One reason is too few distinct values of a cost variable. In the case of a polynomial model, fitting requires at least one more data points than the degree of polynomial. For example, a quadratic model needs at least three. The other reason is too close correlation among regression terms. For example, $X$ and its power term $X^2$ may be collinear, or an interaction term $X \log X$ may be highly correlated with $X$ and $\log X$.

If, for example, a user executes NthGrpMavg repeatedly with the same group symbol and window size but possibly different start and end date and the STM uses the collected data set to update a cost model, then the multi-collinearity problem will happen.

The STM detects the multi-collinearity case by testing if the calculation of $X^T X$ generates a singular matrix error, and handles it by postponing the update of the cost model until more data points are available. The default costs would be used until then. If the problem persists after several tries, a user-interaction is called for, and STM checks the cause and suggests a remedial action to the user.
4 Experiments

We have implemented the STM in Oracle9i and evaluated it using UDFs with different characteristics. The experiments focused on the accuracy of cost estimations tuned over repeated feedback cycles and the efficacy of handling nominal cost variables. In this section we present the experiments performed, specifically, the experimental UDFs in Section 4.1, the data in Section 4.2, the cost models in Section 4.3, the setup in Section 4.4, and the results in Section 4.5.

4.1 Experimental UDFs

Two kinds of database UDFs have been used: two aggregate functions on financial time series data and three keyword-based search functions on text data. All these UDFs are implemented in Oracle PL/SQL.

4.1.1 Time-series UDFs

The two UDFs have the following signatures.

- \( \text{MinGrpMavg}(\text{groupsymbol}, \text{sdate}, \text{edate}, \text{windowsize}) \);
- \( \text{NthGrpMavg}(\text{groupsymbol}, \text{sdate}, \text{edate}, \text{windowsize}, \text{n}) \);

Given a group symbol, MinGrpMavg returns the minimum of group moving average calculated within a specified data range whereas NthGrpMavg returns the n-th minimum of group moving average. Both UDFs are extensions of the conventional moving average functions[15]. These UDFs are white boxes, and their simplified codes are shown in Appendix A.

4.1.2 Text-search UDFs

The three UDFS have the following signatures.

- \( \text{SimpleTextSearch}(\text{list of keywords}) \);
- \( \text{ProximityTextSearch}(\text{list of keywords, max_span}) \);
- \( \text{ThresholdTextSearch}(\text{list of keywords, threshold}) \);

Given a list of keywords connected with AND or OR, SimpleTextSearch searches the text documents and returns the number of documents containing the keywords. ProximityTextSearch returns the number of documents containing the keywords within the proximity of max_span words. ThresholdTextSearch returns the number of documents containing the keywords with the score of at least the threshold. These UDFs invoke a built-in Oracle Text function Contains[16] and are black boxes because we do not know their implementations.

4.2 Experimental data

4.2.1 Time-series data

A time series can be regular or irregular. In a regular time series, data arrives predictably at predefined intervals whereas, in an irregular time series, unpredictable bursts of data arrive at unspecified points in time.

Figure 8 shows the schema of the financial ticker time-series data used in the experiment. The schema tdev contains three tables. The table ticker_index is an index to ticker symbols that are members of a group, the table tsquick_tab is a table that contains the regular time-series data, and the table tsquick_irtab contains the irregular time-series data. The table schema of tsquick_irtab is identical to that of tsquick_tab. The cardinalities of the tables ticker_index, tsquick_tab, and tsquick_irtab are 68, 1629144, and 1585642,
respectively. Irregular data are alterations of these regular data by removing random intervals of data at random time stamp.

4.2.2 Text data

Figure 9 shows the schema of the text data used in the experiment. The schema ctxdev contains only the table text_dataset, which stores text documents as CLOBs. We use XML data of news articles obtained from the Reuters. The cardinality of the table text_dataset is 36422.

4.3 Experimental UDF cost models

Since the time series UDFs are white boxes, they allow for determining the cost variables and, further, building a model based on the code. In contrast, since text search functions are black boxes, they require a
hypothesis-and-test process to determine the cost variables. In this section we outline the cost model building processes and present the resulting cost models.

4.3.1 Time-series UDF cost models

The cost models differ between regular and irregular time-series data.

**Regular time-series** From the two UDFs shown in Appendix A, we derive the user-provided (User) models and the default full quadratic (Default) models as shown below. In these equations, \(a_0, a_1, \ldots, a_9\) denote coefficients (with different values for different models) and \(d, w, g\) denote date range, window size, and group size, respectively.

- **MinGrpMavg**
  - [Default] \[cost = a_0 + a_1 g + a_2 d + a_3 w + a_4 g^2 + a_5 d^2 + a_6 w^2 + a_7 g d + a_8 g w + a_9 d w\]
  - [User] \[cost = a_0 + a_1 d + a_2 g d + a_3 g w + a_4 d w\]

- **NthGrpMavg**
  - [Default] \[cost = a_0 + a_1 g + a_2 d + a_3 w + a_4 g^2 + a_5 d^2 + a_6 w^2 + a_7 g d + a_8 g w + a_9 d w\]
  - [User] \[cost = a_0 + a_1 g (d + w + 1) + a_2 (d + 2) w + a_3 (d + w + 1) + a_4 (d + 2) \log_2 (d + 2)\]

**Irregular time-series** Because irregular time-series have data at different time stamps for different ticker symbols, group size is not constant across time stamps and, therefore, cannot be used as a cost variable. Instead, we introduce group symbol as a nominal cost variable and build separate models for different group symbols. Either Case 2 or Case 3 in Section 3.3.2 applies here. Note that we use the default models in Case 2 and user-provided models in Case 3. For the UDFs considered, one user-provided model is shared by all group symbols. The cost models are reduced from those for the regular time-series by removing the cost variable group size (i.e., \(g\)).

- **MinGrpMavg**
  - [Case 2: Default] \[cost = a_0 + a_1 d + a_2 w + a_3 d^2 + a_4 w^2 + a_5 d w\]
  - [Case 3: User] \[cost = a_0 + a_1 d + a_2 w + a_3 d^2\]

- **NthGrpMavg**
  - [Case 2: Default] \[cost = a_0 + a_1 d + a_2 w + a_3 d^2 + a_4 w^2 + a_5 d w\]
  - [Case 3: User] \[cost = a_0 + a_1 (d + w + 1) + a_2 (d + 2) w + a_3 (d + w + 1) + a_4 (d + 2) \log_2 (d + 2)\]

4.3.2 Text-search UDF cost models

Since the text search UDFs are black boxes, we determine candidate cost variables based on the well-known text indexing and search mechanism [17] and test the sensitivity of the actual costs to the variables. This process leads to one cost variable predominant for all three UDFs: the number of documents containing the searched keyword phrase (abbreviated to numdocs).

For the SimpleTextSearch, we can easily obtain the value of numdocs from the Oracle Text index prior to executing it. For the other UDFs, it requires parsing the BLOBs containing the position lists (i.e., \([docid_1, loc_{11}, loc_{12}, \ldots, docid_2, loc_{21}, loc_{22}, \ldots]\)) of individual keywords but, unfortunately, its byte format is not known to us. Therefore, in our experiments, numdocs of a multi-keyword SimpleTextSearch is estimated based on the numdocs of the individual keywords. Besides, max_span of ProximityTextSearch and threshold of ThresholdTextSearch are used as additional cost variables as if they were uncorrelated to numdocs.
Here, we describe the numdocs estimation for a multi-keyword SimpleTextSearch. Let us define the frequency of a search keyword phrase $K$ as

$$\text{freq}(K) = \frac{\text{numdocs}(K)}{\text{total\_number\_of\_documents}}$$

where $\text{numdocs}(K)$ denotes the number of documents containing $K$. As the denominator is a constant, estimating $\text{numdocs}(K)$ is tantamount to estimating $\text{freq}(K)$. For simplicity, let us assume that keywords have uniform and independent probabilities of occurrences in the text documents. Then,

$$\text{freq}(K_1 \ \text{AND} \ K_2) = \text{freq}(K_1) \times \text{freq}(K_2)$$

$$\text{freq}(K_1 \ \text{OR} \ K_2) = \text{freq}(K_1) + \text{freq}(K_2) - \text{freq}(K_1) \times \text{freq}(K_2)$$

where $K_1$ and $K_2$ denote keyword phrases. Hence, for an arbitrary search keyword phrase $K$, for example, $K = (K_1 \ \text{AND} \ K_2) \ \text{OR} \ (K_3 \ \text{AND} \ K_4)$, $\text{freq}(K)$ is calculated from $\text{freq}(K_1), \text{freq}(K_2), \text{freq}(K_3),$ and $\text{freq}(K_4)$ using the above two equations.

The following equations show the full quadratic cost models of the three UDFs. Note that user-provided models are not applicable because the UDFs are black boxes. In these equations, $a_0, a_1, \ldots, a_5$ denote regression coefficients (with different values for different models), and $n, m,$ and $t$ denote numdocs, max_span, and threshold, respectively.

**SimpleTextSearch:**

[Default] $\text{cost} = a_0 + a_1n + a_2n^2$

**ProximityTextSearch:**

[Default] $\text{cost} = a_0 + a_1n + a_2m + a_3n^2 + a_4m^2 + a_5nm$

**ThresholdTextSearch:**

[Default] $\text{cost} = a_0 + a_1n + a_2t + a_3n^2 + a_4t^2 + a_5nt$

### 4.4 Experimental setup

This section describes various issues pertaining to the setup for the experiments.

**Platform:** Experiments are performed using Oracle9i on SunOS5.8, installed on Sun Ultra Enterprise 450 with four 276MHz CPUs, 1024 Mbyte RAM, and 55 Gbyte hard disk. Oracle is configured to use 16M database buffer cache.

**Programming:** We use C shell script to generate other C shell scripts that, when executed for a particular UDF, generate PL/SQL codes for generating "statistics" objects through Oracle Data Cartridge Interface(ODCI) and for executing the UDF and recording the cost. The codes for model building and update are written in Java.

**STM system data:** We store the STM system data (or, metadata) in Oracle tables. Example data are the registered UDF’s model parameters, cost variables, and the associated configuration and control data shown in Figure 3. For simplicity of the implementation, the cost log data are also stored in a table instead of a file.

**Cost data set distributions:** In a multi-dimensional space defined by cost variables, the distribution of cost data points is determined by the actual values of cost variables used in the UDF executions. We consider two kinds of data distribution: uniform and normal. For the uniform distribution, we assume a finite range of values for each ordinal cost variable. In the experiments, we set the following range of values: the date range of 0 to 100 years, the window size of 1 to 100 days, and the group size of 5 to 12 ticker symbols. The normal distribution simulates the values of cost variables concentrated in local regions. For this distribution, we pick the mean (for the centroid) randomly from the ranges of the cost variables. The standard deviations are set to 200 days for the date range and 3 days for the window size, respectively. We use a product of one-dimensional normal distributions instead of one joint normal distribution, under the assumption of uncorrelated cost variables.

**Performance metric:** As the metric of cost estimation accuracy, we use the relative error defined as the ratio of the absolute difference between the observed and the estimated costs to the observed cost. In our
experiments, \textit{median} relative error is more meaningful than mean relative error because the mean is biased by a small number of excessive relative errors attributed to cases with small values of CPU or disk I/O cost.

\textbf{Caching effect:} In a computer system with dynamic system load, the caching effect on the cost is quite random. We simulate this random effect by flushing database buffer by a random portion at a random interval. For this purpose, the following two parameters are used: interval and percentage. “Interval” controls the number of UDF invocations between two I/O-intensive dummy runs that clear the data buffer, and “percentage” controls the amount of buffer cleared each time. In our experiments, interval is randomly selected from the range [1,20], and percentage from the range [0,100]. Note that in a real system appropriately configured, the caching effect typically ranges in an interval narrower than [0, 100]. That is, we are simulating the worst possible scenario.

\textbf{Workload effect:} Although we use generic cost metrics immune to the fluctuations of workload, an excessive workload may still incur additional disk I/O. This happens when some pages in the buffer are invalidated (if not modified) or flushed to disk (if modified) because of a buffer overflow. The pages must be refetched from disk if needed again, thus incurring additional disk I/O. We ignore this effect because it rarely happens in a system configured adequately (e.g., buffer space, swap space).

\textbf{Negative estimate cost:} When multiple regression is used, some of the predicted values may be negative. This is not desired in cost estimation. Hence, if a negative cost is ever estimated for a query, we simply use zero instead.

### 4.5 Experimental results

We conduct three types of experiments with a focus on the following aspects: (1) the accuracy of cost estimation, (2) handling nominal cost variables, and (3) adapting to the changes of cost variable values. We use the default models for the time-series UDFs. The results are almost the same as those from the user-defined models.

#### 4.5.1 Experiment 1: cost estimation accuracy

We use the two financial time series UDFs on regular time-series data and the three text search functions. For each UDF, Figure 10 shows the median relative errors of the UDF cost estimations from the first through twentieth feedback cycle, with 50 data points each. They also show the distributions of relative errors in the estimated CPU costs and disk I/O costs.

From the figures in the first column of Figure 10, it appears as if the errors did not decrease over repeated cycles. In fact, they do, and very quickly. As the cost models are so suitable to the experimental UDFs, the errors are as small as they can be after the first cycle, and the models are stable.

Table 1 summarizes the percentage rates of cost estimations with relative errors smaller than 30% for the CPU and disk I/O costs. It shows that the disk I/O cost has higher estimation errors more often than the CPU cost. The reason for this is the random effect of caching, which affects only the disk I/O cost.

<table>
<thead>
<tr>
<th>UDF</th>
<th>CPU cost</th>
<th>disk I/O cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>MinGrpMavg</td>
<td>99.7%</td>
<td>79.9%</td>
</tr>
<tr>
<td>NthGrpMavg</td>
<td>95.25%</td>
<td>80.25%</td>
</tr>
<tr>
<td>simpleTextSearch</td>
<td>85.5%</td>
<td>69.7%</td>
</tr>
<tr>
<td>proximityTextSearch</td>
<td>99.0%</td>
<td>43.4%</td>
</tr>
<tr>
<td>thresholdTextSearch</td>
<td>88.9%</td>
<td>38.4%</td>
</tr>
</tbody>
</table>
Execution cost estimation errors of MinGrpMavg

(a) MinGrpMavg on regular time-series data

Execution cost estimation errors of NthGrpMavg

(b) NthGrpMavg on regular time-series data

Execution cost estimation errors of SimpleTextSearch

(c) SimpleTextSearch

Execution cost estimation errors of ProximityTextSearch

(d) ProximityTextSearch

Execution cost estimation errors of ThresholdTextSearch

(e) ThresholdTextSearch

Figure 10: Cost estimation errors for UDFs
Figure 11: Cost estimation errors for MinGrpMavg on irregular time-series data

Table 1 also shows that the text search UDFs incur higher cost estimation errors more often than the financial time series UDFs. The reasons for this is the assumption of independent and uniformly-distributed keywords. In reality, the occurrences of two keywords are correlated in many cases. Thus, the assumption introduces an error in the estimated value of the cost variable numdocs.

Furthermore, compared with SimpleTextSearch, the disk I/O cost estimation errors are larger for both ProximityTextSearch and ThresholdTextSearch. This is caused by ignoring the correlation between numdocs and each of max_span and threshold (in Section 4.3.2). Note that inaccurate estimations of numdocs have more impact on the disk I/O costs than the CPU costs.

4.5.2 Experiment 2: handling nominal cost variables

We use MinGrpMavg on irregular time-series data to evaluate our approach in handling nominal cost variables. As mentioned in Section 4.3.1, with irregular time series the nominal input argument groupsymbol becomes a nominal cost variable that can not be converted to an ordinal one. We use both Case 2 and Case 3 described in Section 3.3.2. The default quadratic model (for Case 2) and the user-provided model (for Case 3) are shown in Section 4.3.1. The text search functions are not applicable here because they have no nominal cost variable.

Figure 11 shows the medians and distributions of relative errors for Case 2 and Case 3. We collect 100 data points in each of the 20 iterations. Table 2 summarizes the percentage rates of cost estimations with relative errors smaller than 30%. The errors are quite comparable to those without nominal cost variables despite the irregularity of the data.
Table 2: Rate of cost estimations with relative errors < 30%

<table>
<thead>
<tr>
<th>Case number</th>
<th>CPU cost</th>
<th>disk I/O cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case 2</td>
<td>94.94%</td>
<td>80.33%</td>
</tr>
<tr>
<td>Case 3</td>
<td>98.42%</td>
<td>79.95%</td>
</tr>
</tbody>
</table>

4.5.3 Experiment 3: adapting to cost variable values

We use the two financial time series UDFs on localized cost data sets simulated with the normal distribution. Text search UDFs are not considered because the localization does not apply to keywords, which are essentially nominal.

In Figure 12, we see that the cost estimation errors are quite large during the first a few cycles. This happens because the data set used to fit the model does not cover the entire ranges of values of the cost variables. For example, the cost model based on one localized data set is not so useful to predict the costs of another localized data set. However, the errors show the tendency of decreasing over the feedback cycles as the covered region expands. Eventually, the errors become comparable to those from the uniformly distributed data. Thus, this experiment demonstrates the self-tuning ability of STM more clearly than in Experiment 1 by, ironically, tuning more slowly.

![Figure 12: Cost estimation errors on localized data sets.](image)

5 Related Work

We find related work in the following two categories: UDF cost modeling and self-tuning cost modeling.
5.1 UDF cost modeling

Our work relates to those in [4] and [5] in terms of UDF cost modeling. In [4] Boulos and Ono use a parade-of-runs for a simple text search UDF. They use the number of keywords and the total size (i.e., bytes) of the text documents as the cost variables, and use multidimensional histogram as the cost model. In their work, however, the data set becomes too voluminous to be stored and processed with reasonable performance. They mitigate this problem by sampling the generated data sets but, as a result, incur high errors in the cost estimation. In [5], Boulos, Viemont and Ono present a curve fitting technique based on neural networks. However, this approach provides a very complex solution that cannot be incorporated into an ORDBMS.

5.2 Self-tuning cost modeling

Our work joins the efforts of building a self-tuning DBMS [8], as exemplified by the automin project [9] at Microsoft Corporation. Self-tuning databases are able to automatically tune themselves to application needs and hardware capabilities, thus reducing the administration overhead significantly. In [9], Chaudhuri et al. 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 addresses a different system issue.

Particularly in [10], Aboulnaga and Chaudhuri present self-tuning histograms, which are built and maintained to estimate the selectivity without accessing the actual data. Each time the selectivity is estimated for a query using the histogram, the estimated value is compared with the actual selectivity and the estimation error is used to refine the histogram. Our work shares the concept of self-tuning estimation, but is distinct for estimating the \textit{UDF execution cost} instead of selectivity.

In [11], Stillger et al. presents a self-tuning approach to repairing an incorrect query execution plan (QEP). Each time a query is executed, the used QEP 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 selectivity and cardinality estimation models. This approach aims at 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 [12], 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. This work is essentially about dynamic resource allocations and is closer to a trigger-action mechanism than a continual self-tuning mechanism.

6 Conclusion

We have presented a self-tuning approach to building and maintaining the cost functions of user-defined functions (UDFs) in an ORDBMS. Our approach is incrementally adaptive in that a cost model is adjusted continuously based on a new data set collected from the recent UDF executions. Additionally, it handles a nominal cost variable by keeping cost functions separately for recently-used values of the variable.

We have built a framework that iterates through three functional components in a feedback loop and updates a user-provided cost model at each cycle. The three components are registering a UDF cost model, recording the UDF execution costs, and updating the UDF cost model. Currently we use multiple regression as the cost model considering UDFs with smooth cost variations. This model allows for an incremental update of the model as a new data set becomes available at each cycle. The framework is also capable of removing outliers and handling multi-collinearity problems.

We have performed experiments in the framework, using two aggregate financial time series UDFs and three text search UDFs. As the financial time series UDFs are white boxes, we have used both the user-
provided models and the default (full quadratic) models. In contrast, as the text search UDFs are black boxes, we have used only the default models. The experimental results show the cost models stabilizing immediately after the first cycle for uniformly distributed data sets and around the third cycle for normally distributed data sets. In addition, the cost models are quite accurate, especially considering the adverse effect of data buffer caching. At least 80% of cost estimations incur lower than 30% relative errors in all experimental cases except the disk I/O costs of text search UDFs, which are black boxes.

The framework works well for UDFs with smooth cost variations. Currently we are developing the next version of the STM that aims at modeling arbitrary (i.e., non-smooth) cost variations while compromising the model precision to an acceptable degree. The key idea is to partition the data space and model data in each partition separately. Multiple regression is not usable in this case and, therefore, we are using a different technique. The future work includes semi-automating the determination of cost variables, by which the STM selects the most influential cost variables among those the user provides or suggests the user to provide other cost variables.

Acknowledgments

We thank Reuters Limited for providing “Reuters Corpus, Volume 1, English language, 1996-08-20 to 1997-0819” for our use in the experiments with text search functions.

A  
Financial Time-Series UDF’s

We show the codes simplified from the original PL/SQL codes.

A.1 MinGrpMavg

FUNCTION MinGrpMavg(groupsymbol IN CHAR, sdate IN CHAR, edate IN CHAR, windowsize IN NUMBER)

RETURN NUMBER IS
    tsdev.ticker_index.ticker_index_id%TYPE;
    tsdev.tsquick_tab.tstamp%TYPE;
    tsdev.tsquick_tab.close%TYPE;
    tickdummy VARCHAR2(10); tickcount NUMBER := 0;
    reccount NUMBER := 0;
BEGIN

    // Cost ~ (groupsize * (daterange + windowsize) + daterange * windowsize)
    OPEN CUESOR c1 FOR
        (SELECT x.ticker_index_id, x.tstamp, sum(x.group_close)/count(x.group_close)
            FROM (SELECT a.ticker_index_id, b.tstamp, sum(b.close)/count(b.close) AS group_close
                FROM tsdev.ticker_index a, tsdev.tsquick_tab b
                WHERE a.ticker_index_id = groupsymbol AND a.ticker = b.ticker
                AND b.tstamp >= sdate - windowsize AND b.tstamp <= edate
                GROUP BY a.ticker_index_id, b.tstamp) x
            WHERE (x.tstamp between to_date(x.tstamp,'DD-MON-YY')-windowsize
                AND to_date(x.tstamp, 'DD-MON-YY'))
                OR (x.tstamp between to_date(edate,'DD-MON-YY')
                AND to_date(edate, 'DD-MON-YY'))
                OR (x.tstamp between to_date(sdate,'DD-MON-YY')
                AND to_date(sdate, 'DD-MON-YY'))
                OR (x.tstamp between to_date(edate,'DD-MON-YY')
                AND to_date(sdate, 'DD-MON-YY')).

    --Cost ~ (daterange)
LOOP UNTIL c1%NOTFOUND
    FETCH C1 INTO ti,ts,tm;
    IF (tickcount = 0) OR (tickcount > tm) THEN tickcount := tm;
    END IF;
END LOOP;
RETURN(tickcount);
END MinGrpMavg;

A.2 NthGrpMavg

FUNCTION NthGrpMavg(groupsymbol: CHAR,sdate: DATE, edate:DATE, windowsize:NUMBER,n:NUMBER) RETURN NUMBER IS
    temp:TABLE OF tsdev.tsquick_tab.close%TYPE;
    tempavg:TABLE OF tsdev.tsquick_tab.close%TYPE;
    j, k, tot: NUMBER;
    BEGIN
        //Read the data of all ticker symbols within the group and
        //build a group average time series. Store the result in temp.
        //This involves opening a cursor and fetching records in a loop.
        -- Cost ~ groupsize(daterange+windowsize+1)
        OPEN CURSOR c1 FOR
            (SELECT b.tstamp,sum(b.close)/count(b.close) AS group_close
                FROM tsdev.ticker_index a,tsdev.tsquick_tab b
                WHERE a.ticker_index_id = groupsymbol AND a.ticker = b.ticker
                AND b.tstamp >= sdate - windowsize AND b.tstamp <= edate
                GROUP BY b.tstamp);
        -- Cost ~ daterange+windowsize+1
        LOOP UNTIL c1%NOTFOUND
            FETCH c1 INTO c1_rec;
            temp(i) := c1_rec.group_close;
            i := i + 1;
        END LOOP;
        //Calculate the moving average of group average time series
        //and store the result in tempavg.
        -- Cost ~(daterange+2)windowsize
        -- Note: temp.count = daterange+windowsize+1
        FOR j = 1 TO (temp.count - windowsize + 1)
            BEGIN
                tot := 0;
                FOR k = j TO (j + windowsize - 1) tot := tot + temp(k);
                tempavg(j) := tot/windowsize;
            END;
        //Mergesort tempavg and return the n-th of the sorted tempavg.
        -- Cost ~(daterange+2)log(daterange+2)
        Mergesort(tempavg, 1, tempavg.count);
        RETURN(tempavg(n));
    END NthGrpMavg;

References


