The Computer Journal, Vol. 47 No. 6, © The British Computer Society; all rights reserved

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

BYUNG SUK LEE[1], LI CHEN[1], JEFF BUZAS[2] AND VINOD KANNOTH[3]

[1]*Department of Computer Science, University of Vermont, Burlington, VT 05405, USA*
[2]*Department of Mathematics and Statistics, University of Vermont, Burlington, VT 05405, USA*
[3]*Federal Reserve Information Technology, 701 East Byrd Street, Richmond, VA 23219, USA*
Email: {bslee, lchen, buzas}@emba.uvm.edu, vinod.kannoth@frit.frb.org

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.

Received 10 May 2003; revised 27 March 2004

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. This paper concerns the cost function.

Today's object-relational database management systems (ORDBMSs) support complex database applications 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 $\mathrm{UDF}_1(\mathrm{args}_1)$ op_1 const_1 AND $\mathrm{UDF}_2(\mathrm{args}_2)$ op_2 const_2'), the cost-based query optimizer needs the cost functions of the UDFs (as well as the selectivity functions of the predicates involving the UDFs) to determine the order of predicate evaluations. Besides, in some ORDBMS research prototypes [1], the query optimizer needs the cost functions to order multiple UDFs invoked in combination (e.g. select $\mathrm{UDF}_1(\mathrm{UDF}_2(\mathrm{UDF}_3(\mathrm{args})))$ from ...). Unfortunately, however, 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.

Traditionally, cost functions are defined for individual algebraic operations (e.g. scans, joins, selections, projections) of query processing. Then, at the time of generating a query execution plan, the cost of executing each alternative plan is calculated as the summation of the costs of executing the algebraic operations that constitute the plan. Each such cost function is defined as a function of parameters such as data profile parameters (e.g. table cardinality, column selectivity, index height, join selectivity), hardware parameters (e.g. disk page size, main memory buffer size) and derived parameters (e.g. disk page blocking factor). We call this approach the analytic approach in this paper. In order to build a cost function using this approach, the user must have thorough knowledge of the query processing mechanism inside the DBMS. This burden is overwhelming for most users. Furthermore, it is a nearly impossible task to build the cost function of a UDF using the analytic approach. We will explain the reason in detail in Section 2.2.

1.2. Problem formulation

This paper concerns generating the cost functions of database UDFs created as stored procedures (or functions). This problem has not been addressed actively, although several researchers addressed ordering predicates involving UDFs (i.e. UDF predicates) [2, 3, 4] or ordering multiple UDFs invoked in combination [5, 6] assuming their cost functions are provided. Presumably, this is because of the complexity of the problem inherent in the analytic approach.

The only published ones we find are works by Boulos and Ono [7, 8]. We have done some works as well [9, 10]. These works, except [8], use statistical approaches. These approaches execute a UDF repeatedly for different combinations of the sample values of the variables influencing the cost (called 'cost variables') and generate cost data coupled with the cost variable values used. Then, a data analysis builds a cost function by applying a data reduction technique [11] to the generated cost data sets. For this purpose, a regression model is used in [9, 10] whereas a multi-dimensional histogram is used in [7]. In [7], this approach is called the 'parade-of-runs' (PoR) approach based on the cost data set collection method. We use the same term in this paper as well.

This PoR approach is simple, thus easy to implement. Besides, it requires very little from the user compared with the analytic approach. That is, users need only to provide the cost variables and the specification for sampling the values of cost variables. The resulting cost function (or cost model) is fairly precise provided with a suitable modeling technique and a sufficiently large cost data set. However, the PoR approach 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 data statistics (e.g. the number of tuples, the number of distinct column values, index height) and 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 always valid. Moreover, the generated cost function is not valid outside the range 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: (i) to dispense with the parade of runs and (ii) 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 self-tuning modeling (STM) of the costs.

First, we build a cost function as a statistical regression model as in [9, 10]. 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—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 overhead is typically insignificant relative to the costs of database UDFs. Besides, the logging 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 diminish as the iteration progresses, and eventually the model becomes stable as sufficient data are considered. If a change occurs in the costs afterwards, the model adapts again incrementally.

1.4. Scope of the work

In this paper we focus on those UDFs whose costs vary smoothly with respect to cost variables. We have used built-in Oracle Data Cartridge functions such as financial time series functions [12], text search functions [13], and spatial search functions [14]. Despite unpredictable cost variations apparent from the semantics of the UDFs, the actual cost variations proved to be smooth enough given the cost variables. Our experiences with these UDFs suggest that a non-trivial number of common database UDFs show smooth cost variations. It is also indicated in [6] that many simple UDFs are added by database users.

For these UDFs, regression suits well as the modeling technique. Compared with the analytic approach, regression achieves more accurate cost estimation due to its ability of fitting complex and often erroneous data to minimize the overall estimation error. We use parametric regression in this paper. Parametric regression achieves precise fitting efficiently provided with an appropriate modeling function. We particularly use multiple regression for the two types (time series and text search) of UDFs used in our experiments. Multiple regression is simple and yet adequately precise for data varying reasonably smoothly. More importantly, it allows for an incremental update of the model with additional data, thus allowing for discarding the data while keeping

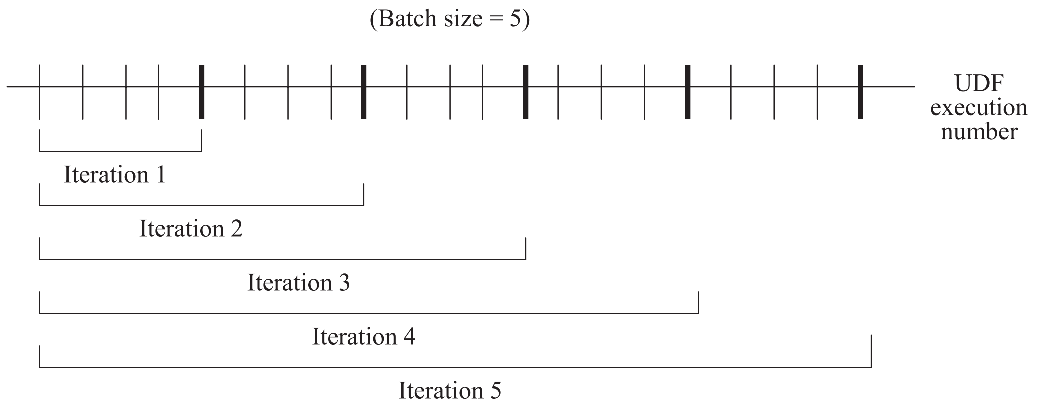


FIGURE 1. Incrementally adaptive model building.

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.

In case a modeling function cannot be provided, non-parametric regression may be used instead. The spatial search functions fit this case. For these UDFs, it is hard for users to provide a parametric modeling function because the cost variation depends heavily on the distributions of spatial data, which can be quite arbitrary. Currently we do not consider non-parametric regression because, as far as we know, they do not allow for an incremental model update. Besides, non-parametric techniques are computationally more expensive than parametric techniques and perform poorly as the number of cost variables (or the dimensionality of the data set) increases.

1.5. 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. To reflect a realistic system environment, we have deliberately included a database buffer caching effect in the experiments, although even the state-of-the-art query optimizers do not consider it [15].

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 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] Oracle is a trademark of Oracle, Inc.

[2] Data Cartridge is a trademark of Oracle, Inc.

1.6. 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 approach to incrementally updating a model (using multiple regression) without saving the entire cost data set. This enables the ORDBMS to maintain the cost models of many UDFs with only the memory needed for storing the regression coefficients.

1.7. 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 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, show the difficulty of generating a UDF cost function using the analytic approach, and describe the PoR approach to UDF cost modeling used in the previous work [9, 10].

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

```
Input:
  Func: an object of type ODCIFuncInfo used by Oracle (for an internal processing)
  Ticker, StartDate, EndDate and WindowSize: input arguments of MinMavg
Output:
  Cost: an object of type ODCICost, consisting of CPU cost, I/O cost, and network cost
  Success:  a flag indicating a successful completion (function of package ODCIConst)
Procedure ODCIStatsFunctionCost:
{
   Initialize Cost to null;
   range = select (EndDate – StartDate) from dual;
   Cost.CPUCost = a function of cost variables Range and WindowSize;
   Cost.IOCost = a function of cost variables Range and WindowSize;
   Cost.NetworkCost = –1;
   return(Success, Cost);
}
```

FIGURE 2. Registering the cost functions of a UDF MinMavg through ODCIStatsFunctionCost.

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 'MinMavg' and another on the text search function 'Contains'. Users are required to provide the cost functions of MinMavg and Contains (as well as selectivity functions).

```
select c.name, e.name
from Employee e, Company c
where e.work-for = c.id
and MinMavg(c.ticker, 'JAN-01-1902',
            'DEC-31-2002', 30) > 50
and Contains(e.resume, ''UNIX and NT'', 1) > 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: (i) CPU cost as the number of machine instructions executed by the CPU, (ii) disk I/O cost as the number of data pages fetched from disk to main memory buffer and (iii) 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.

Provided with cost functions, the extensible query optimizer orders the UDF predicates based on their estimated execution costs. Oracle facilitates it by providing Oracle Data Cartridge Interface (ODCI) through which users can register cost functions as the components of a 'statistics object'. Figure 2 sketches registering the CPU and disk I/O cost functions of MinMavg. Its PL/SQL implementation is shown in Appendix B, where the cost functions are hard-coded as regression equations.[3] In the figure, ODCIStatsFunctionCost is an ODCI function. If no cost function is registered, Oracle uses its own default costs.

In [16] it is stated that Oracle query optimizer evaluates the predicates specified in the 'where' clause in the following order.

(i) Non-UDF predicates, in the order specified in the clause.
(ii) UDF predicates with associated cost functions, in an increasing order of the costs.
(iii) UDF predicates without associated cost functions, in the order specified in the clause.
(iv) Predicates not specified in the clause but transitively generated by the optimizer.
(v) Predicates with sub-queries, in the order specified in the clause.

2.2. Analytic approach to building a cost function

As mentioned in Section 1.1, it is very hard for common users to build a cost function of a UDF in the form of an analytic function of parameters acquired from database profiles and system configurations. The following example is from [17], and is intended to demonstrate the complexity of the analytic approach.

EXAMPLE 2.1. Consider a single loop equijoin $R \bowtie_{R.A=S.B} S$ between two relations R and S. There are different cost functions for this equijoin operation depending on the index structure used for accessing the tuples of the table S. For instance, if a clustering index is used, the cost function is

$$\begin{aligned}\text{Cost}_{\text{clustering\_index}} &= b_R + (|R| \times (x_B + s_B/bfr_S)) \\ &\quad + ((js \times |R| \times |S|)/bfr_O)\end{aligned}$$

[3]In STM, the cost function is not hard-coded but is stored in relational tables.

and, if a hash index is used, then the cost function is

$$\text{Cost}_{\text{hash\_index}} = b_R + (|R| \times h) + ((js \times |R| \times |S|)/bfr_O)$$

Here, b_R denotes the number of disk pages needed to store the relation R and b_S denotes that for the relation S; x_B denotes the height of the index on the column B of the relation S; s_B denotes the selection cardinality of B, i.e. the average number of records of S that satisfy the equality condition on the attribute B; js denotes the join selectivity of the equijoin between R and S; h denotes the average number of pages accessed to retrieve a record given its hash value; $|R|$ and $|S|$ respectively denote the cardinalities of the relations R and S; bfr_S denotes the blocking factor (i.e. the number of records in one disk page) of the data file storing the relation S; bfr_O denotes the blocking factor of the output file storing the join result.

This example shows only two of several possible cost functions for the same operation. Building the cost function this way is difficult for most naive users. Furthermore, this operation is only one step in executing a query statement as explained in the following example.

EXAMPLE 2.2. Consider the following SQL query statement for finding the employees working for the Research department and working on projects run by the same department with the project budget more than one million dollars.

```
select Employee.name, Project.budget
from Employee, Department, Project
where Employee.workfor = Department.dnumber
and Project.runby = Department.dnumber
and Employee.workon = Project.pnumber
and Project.budget >1000000
and Department.deptname = ''Research'';
```

Executing this query involves the following algebraic operations:

- Equijoin between the `Employee` table and the `Department` table by the join condition `Employee.dno = Department.dnumber`.
- Equijoin between the `Project` table and the `Department` table by the join condition `Project.runby = Department.dnumber`.
- Equijoin between the `Employee` table and the `Project` table by the join condition `Employee.workon = Project.pnumber`.
- Selection from the `Project` table by the column selection condition `Project.budget > 1000000`.
- Selection from the `Department` table by the column selection condition `Department.deptname = ''Research''`.
- Projection of the column `name` from the `Employee` table and the column `budget` from the `Project` table.

Additionally, there are other operations for processing the intermediate results during the query execution.

To make it worse, a UDF like the one shown in Appendix A involves one or more SQL statements embedded in a stored function. The internal operation of such a UDF is far too complicated to allow for generating the cost model in the form of an analytic function. Note that the query optimizer can only use the cost function of the entire UDF treated as one atomic operator. That is, it cannot make use of the cost functions of individual algebraic operations constituting the SQL statements within the UDF.

2.3. The parade-of-runs approach

In the PoR approach used in [9, 10], 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. In this section we present the results from using two kinds of UDFs: aggregate financial time series functions and keyword-based text search functions.

The former UDFs include NthGrpMavg (groupsymbol, startdate, enddate, windowsize, *n*). An example is NthGrpMavg(NASIND1, 'JAN-1-1980', 'DEC-31-2002', 30, 2). Given the group symbol NASIND1, NthGrpMavg returns the 2nd minimum moving 30-day window average of the group average time series calculated within the range of dates JAN-1-1980 $\sim$ DEC-31-2002. 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.)

The latter UDFs include SimpleTextSearch(text-documents, query). An example is SimpleTextSearch (news_articles, 'election AND poll AND candidates'). Given the multi-keyword Boolean query 'election AND poll AND candidates', the SimpleTextSearch returns all documents containing all three keywords 'election', 'poll' and 'candidates' from the news_articles document set. The UDF is implemented in PL/SQL and calls a built-in Oracle Text function 'Contains', which is a black box to the user.

The cost variables of the times series UDFs are determined directly from the input arguments based on the semantics of the UDFs. Those of the text search UDFs are determined straightforwardly from the inverted text index structure [18] well known in the information retrieval area. We will present the details of determining the cost variables in Section 4.3. For both kinds of UDFs, their costs vary smoothly and monotonously with respect to the cost variables, and are modeled quite precisely using a quadratic regression model.

Figure 3 illustrates the process of generating the cost function using the PoR approach, 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,

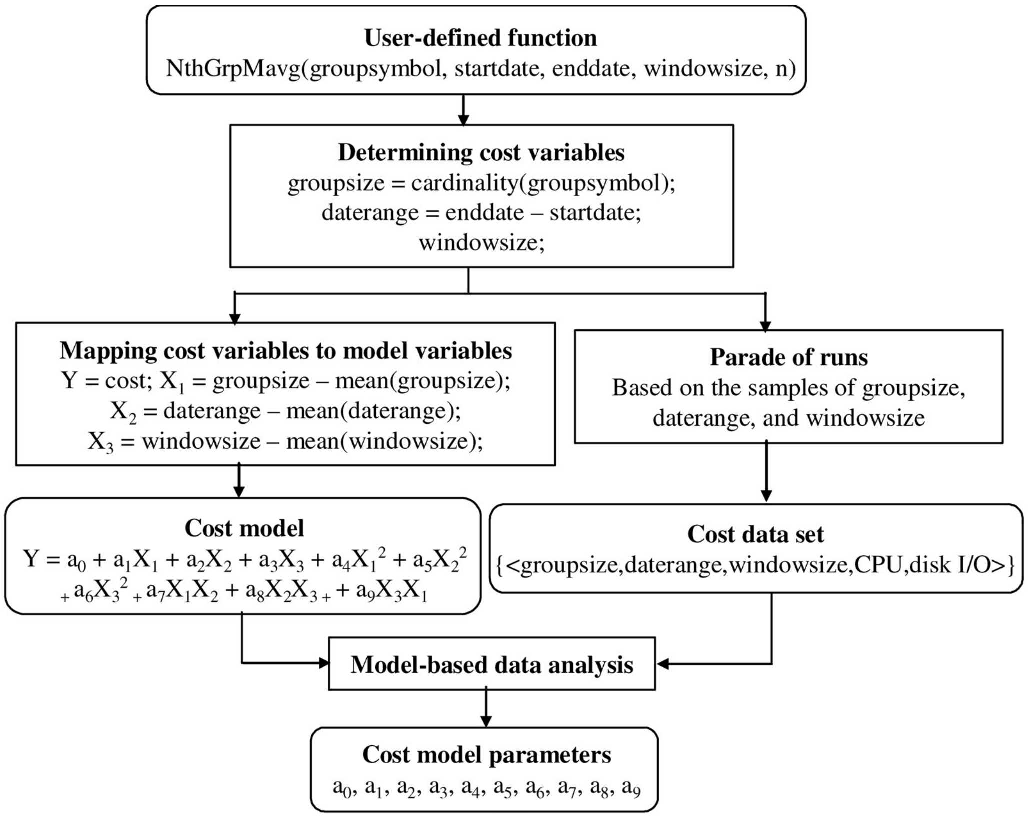

User-defined function
NthGrpMavg(groupsymbol, startdate, enddate, windowsize, n)
Determining cost variables
groupsize = cardinality(groupsymbol);
daterange = enddate – startdate;
windowsize;
Mapping cost variables to model variables
Y = cost; X_1 = groupsize – mean(groupsize);
X_2 = daterange – mean(daterange);
X_3 = windowsize – mean(windowsize);
Parade of runs
Based on the samples of groupsize, daterange, and windowsize
Cost model
$Y = a_0 + a_1X_1 + a_2X_2 + a_3X_3 + a_4X_1^2 + a_5X_2^2 + a_6X_3^2 + a_7X_1X_2 + a_8X_2X_3 + a_9X_3X_1$
Cost data set
{<groupsize,daterange,windowsize,CPU,disk I/O>}
Model-based data analysis
Cost model parameters
$a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9$

FIGURE 3. Cost modeling using the parade-of-runs approach.

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[4] 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 for the time series UDFs. Specifically, the mean and median relative errors are lower than 1.5% for both the CPU cost and the disk I/O cost. In the case of the text search UDFs, the value of the cost variable should be estimated due to the lack of access to Oracle Text index structure. Overall the resulting mean relative errors are 5–21% for the CPU cost and 9–62% for the disk I/O cost. These errors are acceptable considering several causes of errors including the OS buffer caching effects and the cost variable estimation errors.

Besides, the resulting cost functions are easily incorporated into the Oracle query optimizer. However, the overhead of parades of runs is significant, especially for NthGrpMavg because it is an expensive UDF that involves mergesort. In addition, the experiment does not consider the database buffer caching effect on the disk I/O cost. Since a disk page access does not incur a physical page fetch from disk if the page is already cached in the database buffer, the actual disk I/O cost varies depending on which pages are cached in the database buffer. Therefore, the accuracy of the disk I/O cost estimation is misleading unless the effect of caching is taken into consideration.

[4]Centered data reduces the fitting error by reducing the collinearity between power terms (e.g. X and X^2) [19].

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 4 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

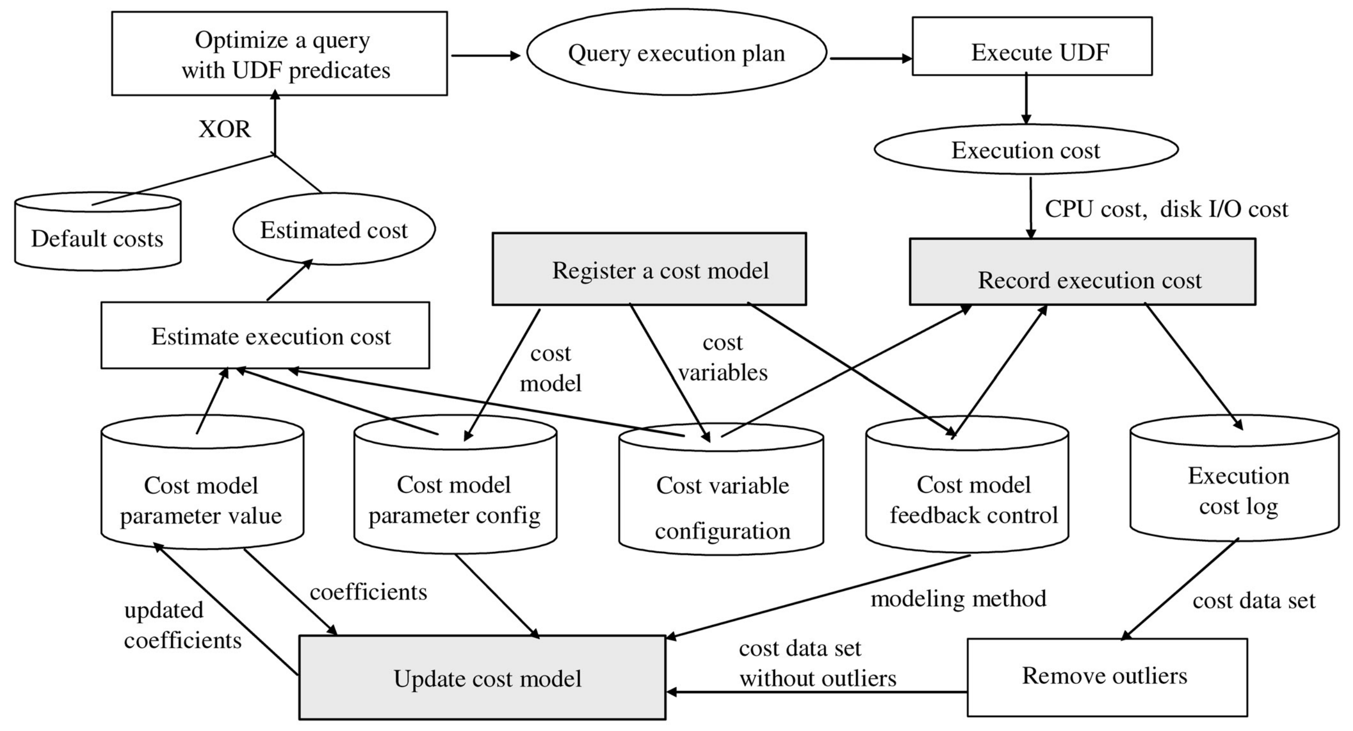


FIGURE 4. The STM framework.

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 $\{\langle v_1, v_2, \ldots, v_k, c, d\rangle\}$ 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 β_i^C and β_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 $\hat{C}_{i+1} = V_{i+1}\hat{\beta}_i^C$ and $\hat{D}_{i+1} = V_{i+1}\hat{\beta}_i^D$, where V_{i+1}, $\hat{C}_{i+1}$, and $\hat{D}_{i+1}$ denote the vectors of, respectively, $\langle v_1, v_2, \ldots, v_k\rangle$ 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}, $\hat{C}_{i+1}$, $\hat{D}_{i+1}$ is stored in a log side by side, and the resulting set of records B_{i+1} is feedback to upgrade β_i^C and β_i^D to β_{i+1}^C and β_{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.

3.2. 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 5 shows the steps involved in registering a UDF and its cost model.

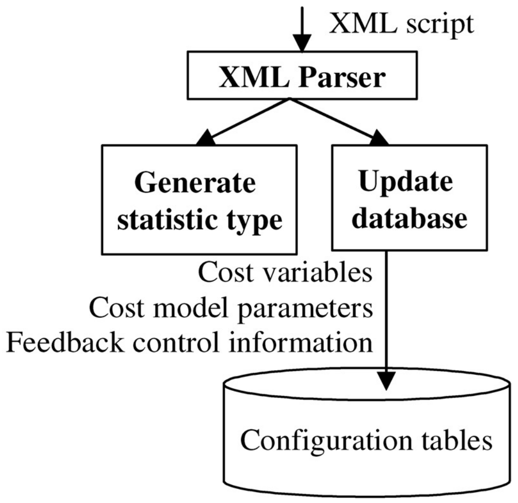


FIGURE 5. Registering a cost model.

Parsing the XML script. In our implementation, XML is used as the interface language for registering a cost model. Figure 6 shows an example XML script, which contains the same information as in Figure 3 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

```
<?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>
<Ordinal name=‘X2’ value=‘enddate-startdate’ type=‘computed’ ></Ordinal>
<Ordinal name=‘X3’ value=‘windowsize’ type=‘direct’></Ordinal>
<Term coefficient=‘a0’ value=‘1’ type=‘constant’></Term>
<Term coefficient=‘a1’ value=‘X1’ type=‘direct’></Term>
<Term coefficient='a2' value='X2' type='direct'></Term>
<Term coefficient='a3' value='X3' type='direct'></Term>
<Term coefficient=‘a4’ value=‘X1*X1’ type='computed'></Term>
<Term coefficient=‘a5’ value=‘X2*X2’ type='computed'></Term>
<Term coefficient=‘a6’ value=‘X3*X3’ type='computed'></Term>
<Term coefficient=‘a7’ value=‘X1*X2’ type='computed'></Term>
<Term coefficient=‘a8’ value=‘X2*X3’ type='computed'></Term>
<Term coefficient=‘a9’ value=‘X1*X3’ type='computed'></Term>
</Model>
</CPU>
<IO default=‘1000’>···</IO>
</UDF>
```

FIGURE 6. An example XML script for registering a cost model.

methods (e.g. direct, computed) of the associated terms (e.g. X1$*$X1).

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 associates 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 7 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.

3.2.3. UDF cost model update

Figure 8 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 is 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.)

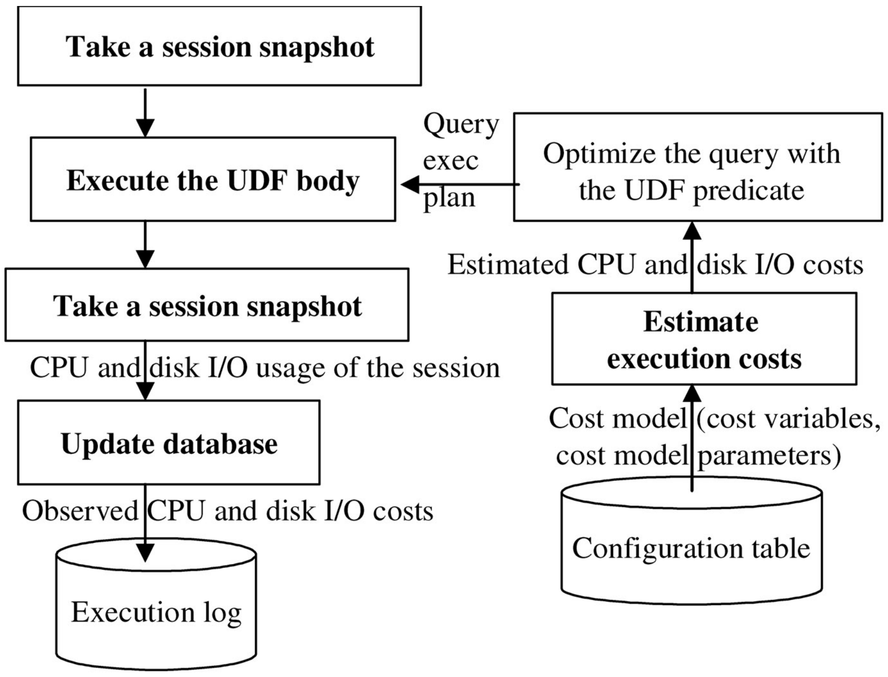

Take a session snapshot
Execute the UDF body
Query exec plan
Optimize the query with the UDF predicate
Estimated CPU and disk I/O costs
Take a session snapshot
CPU and disk I/O usage of the session
Estimate execution costs
Update database
Cost model (cost variables, cost model parameters)
Observed CPU and disk I/O costs
Configuration table
Execution log

FIGURE 7. Recording execution costs during the run-time.

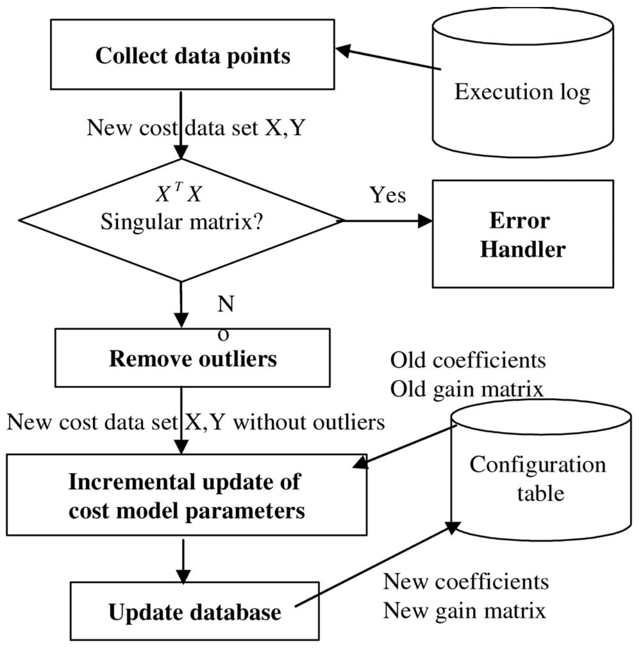


FIGURE 8. Updating a cost model using multiple regression.

PROPERTY 3.1. *Consider a multiple regression model with p coefficients.*

$$Y = X\beta + E$$

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

$$\hat{\beta}_{new} = (X_1^T X_1 + X_a^T X_a)^{-1}(X_1^T X_1 \hat{\beta}_1 + X_a^T Y_a) \quad (1)$$

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 $\langle X_a, Y_a\rangle$
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
```

As mentioned in Section 1, using this algorithm the ORDBMS can maintain the cost models of many UDFs with only the memory needed for storing the regression coefficients, i.e. without storing the entire cost data set.

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 an 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 2.3.
- 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, i.e. 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.[5]

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 semi-studentized residuals[6] [20]. A semi-studentized 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 semi-studentized residual of the data point is greater than 4.

DEFINITION 3.1. (Semi-studentized residuals) *Let $\langle X, Y\rangle$ 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*

$$\hat{E} = Y - \hat{Y} = Y - X\hat{\beta}$$

and the mean square error 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 semi-studentized residuals is defined as

$$\frac{\hat{E}}{\sqrt{\text{MSE}}} = \begin{pmatrix} \frac{y_1-\hat{y}_1}{\sqrt{\text{MSE}}} \\ \vdots \\ \frac{y_n-\hat{y}_n}{\sqrt{\text{MSE}}} \end{pmatrix}$$

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 semi-studentized residual (δ) 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 $\langle X_a, Y_a\rangle$ used to update the model
  Sum of squared errors SSE and the number of observations
N
Output:
  Data set $\langle X_a, Y_a\rangle$ 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. $\text{SSE}_1 := \sum_{i=1}^{N_1} (Y_i - \hat{Y}_i)^2$; //SSE of the additional data
     set
  4. $\text{SSE} := \text{SSE} + \text{SSE}_1$; //Update SSE for the entire data
     set.
  5. $\text{MSE} := (\frac{1}{N-p})\text{SSE}$;
  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 point than the degree of polynomial. For example, a quadratic model needs at least three data points. 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^TX generates a singular matrix error, and

[5]Not implemented in our system yet.

[6]We use semi-studentized instead of studentized residuals because the latter requires saving all the previous data.

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.

- MinGrpMavg(groupsymbol, sdate, edate, windowsize);
- NthGrpMavg(groupsymbol, sdate, edate, windowsize, 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 [12]. 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.

- SimpleTextSearch (text documents, list of keywords);
- ProximityTextSearch (text documents, list of keywords, max_span);
- ThresholdTextSearch (text documents, 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 [13] and are black boxes because we do not know the implementation of Contains.

4.2. Experimental data

4.2.1. Time-series data

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

Figure 9 shows the schema of the financial ticker time-series data used in the experiment. The schema tsdev 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 1512775, respectively. The sizes of the tables are about 2 KB, 56 MB and 52 MB, 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 10 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 character large objects (CLOBs). We use XML data of news articles obtained from Reuters. The cardinality of the table text_dataset is 36422, and the table size is 72 MB. The text index size after loading the document set is 56.8 MB.

4.3. Experimental UDF cost models

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 and g denote daterange, windowsize and groupsize, respectively.

MinGrpMavg:

$$\text{[Default]} \quad cost = a_0 + a_1 g + a_2 d + a_3 w + a_4 g^2 + a_5 d^2 + a_6 w^2 + a_7 gd + a_8 gw + a_9 dw$$

$$\text{[User]} \quad cost = a_0 + a_1 d + a_2 gd + a_3 gw + a_4 dw$$

NthGrpMavg:

$$\text{[Default]} \quad cost = a_0 + a_1 g + a_2 d + a_3 w + a_4 g^2 + a_5 d^2 + a_6 w^2 + a_7 gd + a_8 gw + a_9 dw$$

$$\text{[User]} \quad cost = a_0 + a_1 g(d + w + 1) + a_2 (d + 2) w + a_3 (d + w + 1) + a_4 (d + 2) \log_2 (d + 2)$$

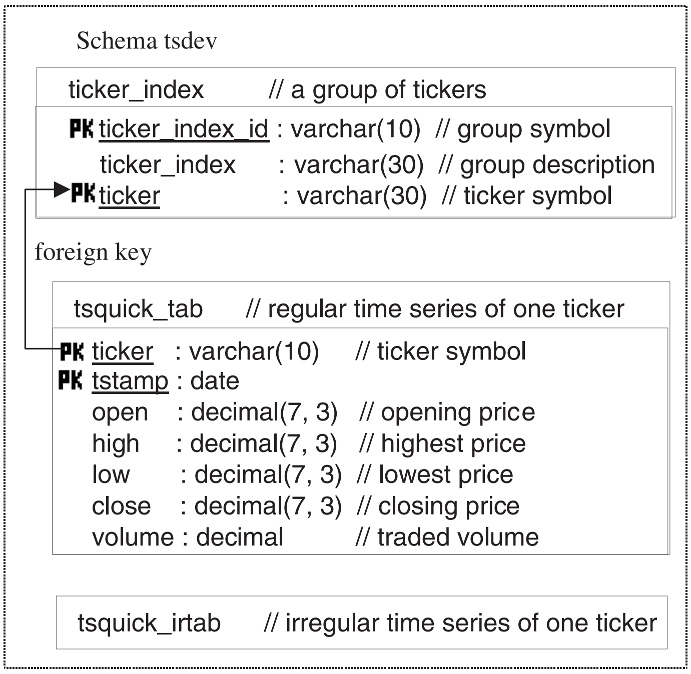

Schema tsdev
ticker_index // a group of tickers
PK ticker_index_id : varchar(10) // group symbol
ticker_index : varchar(30) // group description
PK ticker : varchar(30) // ticker symbol
foreign key
tsquick_tab // regular time series of one ticker
PK ticker : varchar(10) // ticker symbol
PK tstamp : date
open : decimal(7, 3) // opening price
high : decimal(7, 3) // highest price
low : decimal(7, 3) // lowest price
close : decimal(7, 3) // closing price
volume : decimal // traded volume
tsquick_irtab // irregular time series of one ticker

FIGURE 9. Schema of the experimental time-series data.

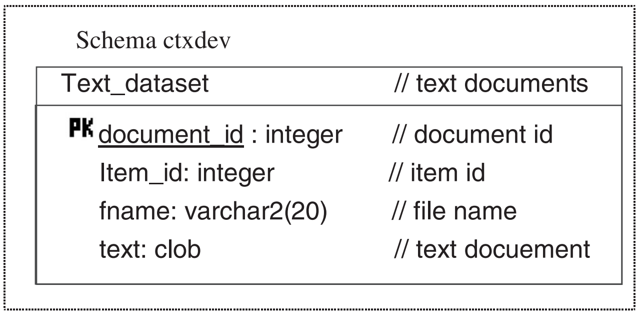

Schema ctxdev
Text_dataset // text documents
PK document_id : integer // document id
Item_id: integer // item id
fname: varchar2(20) // file name
text: clob // text docuement

FIGURE 10. Schema of the experimental text data.

Irregular time-series. Because irregular time-series have data at different time stamps for different ticker symbols, groupsize 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 groupsize (i.e. g).

MinGrpMavg:

[Case 2: Default] $\text{cost} = a_0 + a_1 d + a_2 w + a_3 d^2 + a_4 w^2 + a_5 dw$

[Case 3: User] $\text{cost} = a_0 + a_1 d + a_2 w + a_3 dw$

NthGrpMavg:

[Case 2: Default] $\text{cost} = a_0 + a_1 d + a_2 w + a_3 d^2 + a_4 w^2 + a_5 dw$

[Case 3: User] $\text{cost} = a_0 + a_1 (d+2)w + a_2 (d + w + 1) + a_3 (d+2) \log_2 (d+2)$

4.3.2. Text-search UDF cost models

Since the text-search UDFs are black boxes, we determine the cost variables based on the well-known text indexing and search mechanism [18]. It is straightforward to identify one cost variable predominant for all three UDFs: the number of documents containing the searched keyword phrase (abbreviated to numdocs). The value of this variable can be obtained from the index with a simple look-up of the postings lists and any necessary Boolean processing (e.g. AND, OR, NOT) depending on the text query expression.

Unfortunately, however, the internal format of the postings lists of the Oracle Text index 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 as explained below. 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

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

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

$$\begin{aligned}\text{freq}(K_1 \text{ AND } K_2) &= \text{freq}(K_1) \times \text{freq}(K_2)\\ \text{freq}(K_1 \text{ OR } K_2) &= \text{freq}(K_1) + \text{freq}(K_2)\\ &\quad - \text{freq}(K_1) \times \text{freq}(K_2)\end{aligned}$$

where K_1 and K_2 denote keyword phrases. Hence, for an arbitrary search keyword phrase K, e.g., $K = (K_1 \text{ AND } K_2) \text{ OR } (K_3 \text{ AND } K_4)$, freq($K$) is calculated from freq(K_1), freq(K_2), freq(K_3) and 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] $\text{cost} = a_0 + a_1 n + a_2 n^2$

ProximityTextSearch:
[Default] $\text{cost} = a_0 + a_1 n + a_2 m + a_3 n^2 + a_4 m^2 + a_5 nm$

ThresholdTextSearch:
[Default] $\text{cost} = a_0 + a_1 n + a_2 t + a_3 n^2 + a_4 t^2 + a_5 nt$

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 276 MHz CPUs, 1024 MB RAM and 55 GB hard disk. Oracle is configured to use 16 MB 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 4. 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 0–100 years, the window size of 1–100 days, and the group size of 5–12 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, 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.

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 the 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 a 'rainy day' scenario.

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).

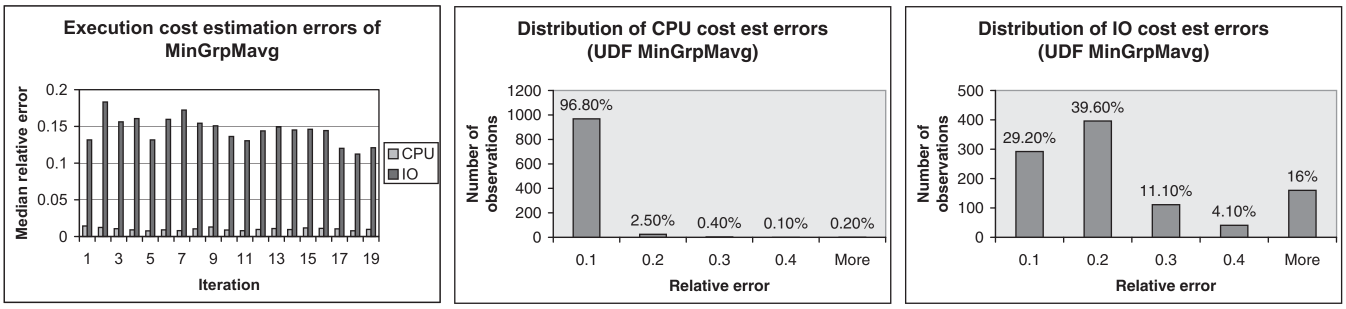
Negative estimate cost. When multiple regression is used, some of the predicted values may be negative. This is not desired in cost estimation. Hence, if a negative cost is ever estimated for a query, we simply use zero instead.

4.5. Experimental results

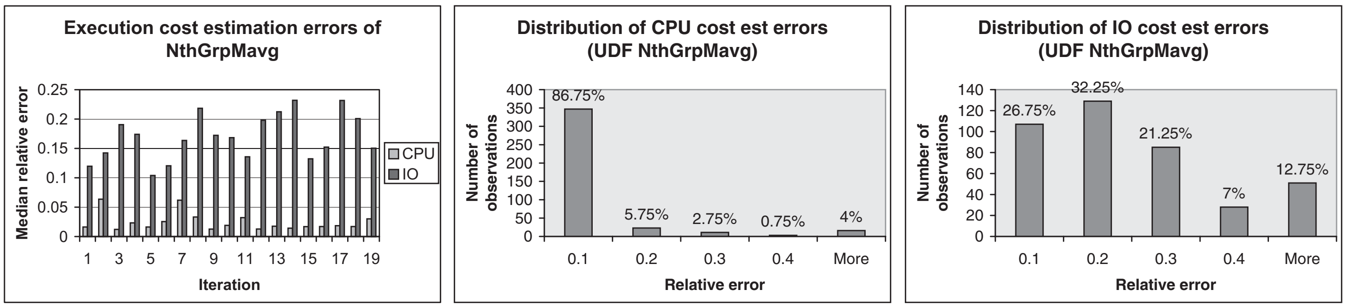
We conduct three types of experiments with a focus on the following aspects: (i) the accuracy of cost estimation, (ii) handling nominal cost variables and (iii) adapting to the changes of cost variable values. We use the default models for the time-series UDFs. The results are almost the same as those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

4.5.1. Experiment 1: cost estimation accuracy

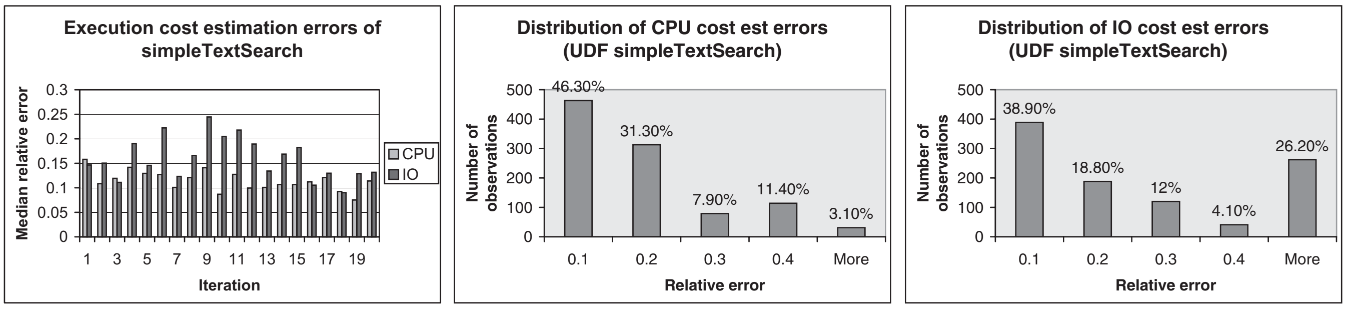
We use the two financial time series UDFs on regular time-series data and the three text search functions. For each UDF, Figure 11 shows the median relative errors of the UDF cost



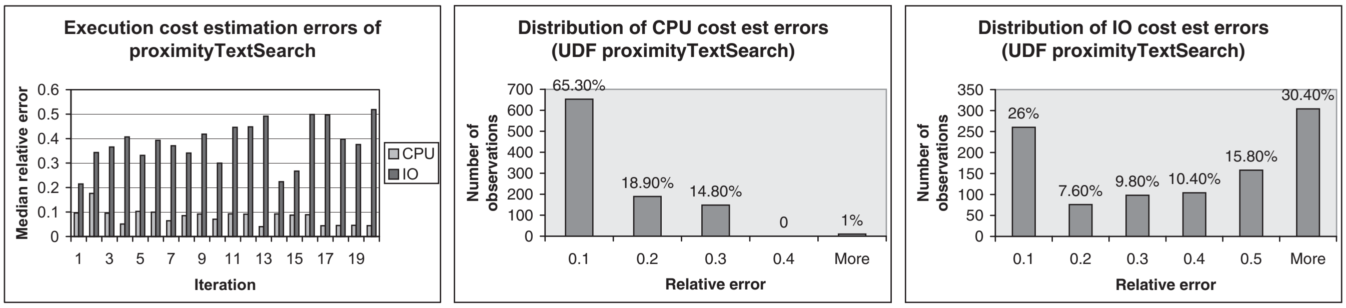
(a) MinGrpMavg on regular time-series data



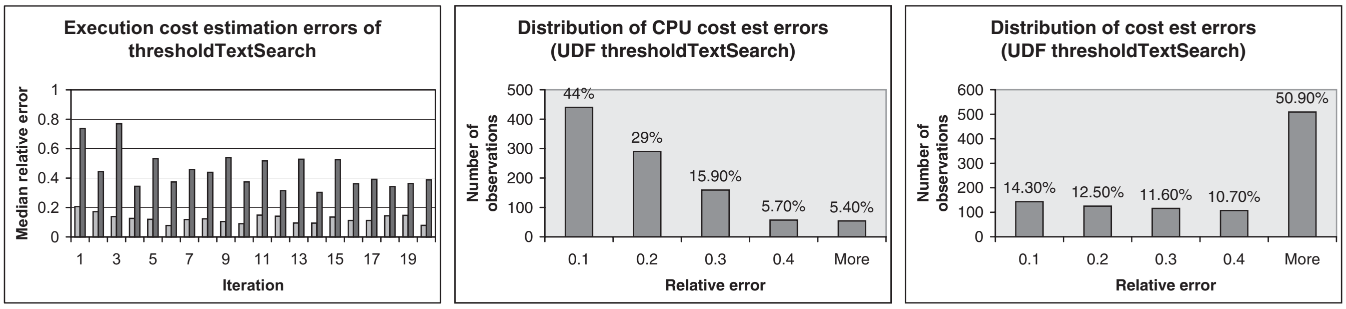
(b) NthGrpMavg on regular time-series data



(c) SimpleTextSearch



(d) ProximityTextSearch



(e) ThresholdTextSearch

FIGURE 11. Cost estimation errors for UDFs.

TABLE 1. Rate of cost estimations with relative errors <30%.

UDF	CPU cost (%)	Disk I/O cost (%)
MinGrpMavg	99.7	79.9
NthGrpMavg	95.25	80.25
simpleTextSearch	85.5	69.7
proximityTextSearch	99.0	43.4
thresholdTextSearch	88.9	38.4

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 11, 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 lower 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 1 also shows that the text search UDFs incur higher cost estimation errors more often than the financial time series UDFs. The reason 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 cannot 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 12 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 lower than 30%. The errors are quite comparable to those without nominal cost variables despite the irregularity of the data.

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 13, we see that the cost estimation errors are quite large during the first 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.

5. RELATED WORK

We find related work in the following three categories: UDF cost modeling, regression-based cost modeling and self-tuning modeling.

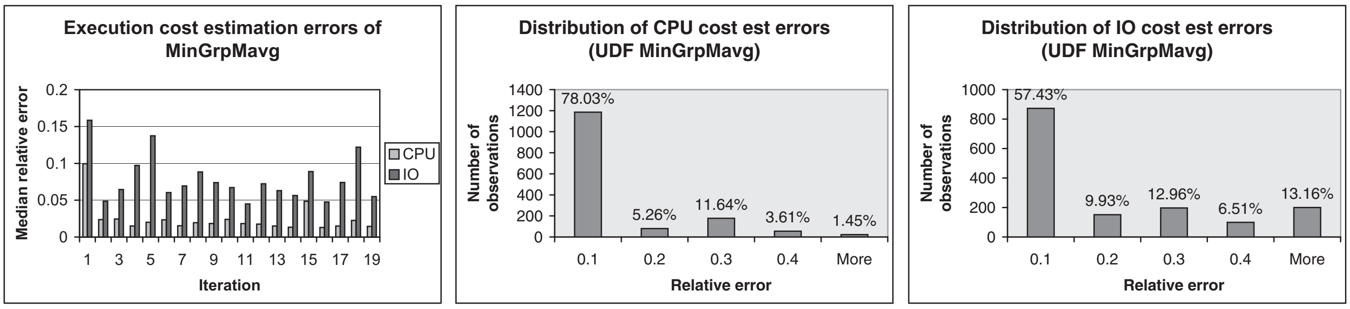
5.1. UDF cost modeling

Our work relates to those in [7] and [8] in terms of UDF cost modeling. In [7], 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 a 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 [8], Boulos *et al.* present a curve fitting technique based on neural networks. However, this approach provides a very complex solution that cannot be incorporated into an ORDBMS.

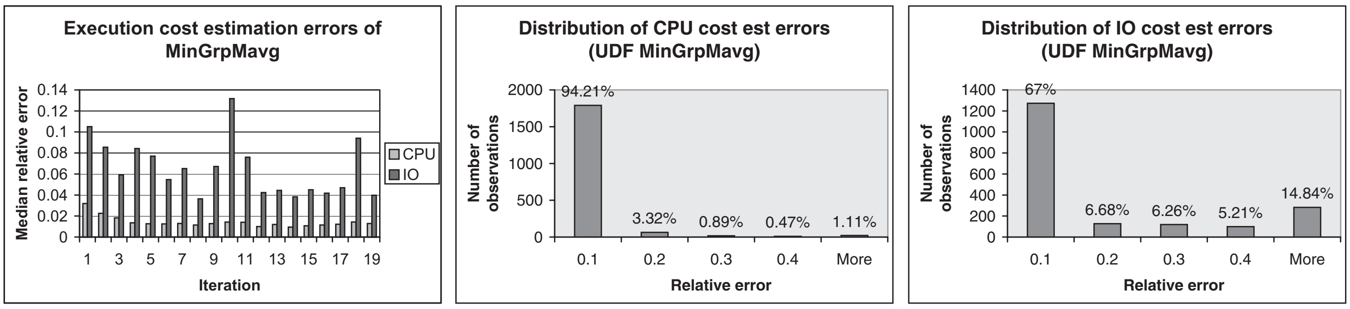
5.2. Regression-based cost modeling

Multiple regression has been used to generate a cost model in other works as well. Andres *et al.* [21] model DBMS performance as a regression equation and tune its coefficients by running a set of representative workload. Ebrahimi [22] uses a similar approach to tune the coefficients of software (not database) cost model.

There are two kinds of efforts made to derive local cost functions of query operations for use by a global query optimizer in a multidatabase environment: model calibrating [23, 24] and query sampling [25, 26, 27]. Du *et al.* [23] develop a cost function by combining the cost models of individual query operations (e.g. select, join) into a regression equation and calibrating the coefficients at each local DBMS by running synthetic operations on a synthetic database. Gardarin *et al.* [24] extend Du et al.'s work to an object-oriented query optimization.



(a) Case 2



(b) Case 3

FIGURE 12. Cost estimation errors for MinGrpMavg on irregular time-series data.

TABLE 2. Rate of cost estimations with relative errors <30%.

Case number	CPU cost (%)	Disk I/O cost (%)
Case 2	94.94	80.33
Case 3	98.42	79.95

In the query sampling method [25, 26, 27], Zhu *et al.* categorize all possible query operations into classes by the data access method used, and develop regression cost models associated with each class. Each class contains either unary (select) or binary (join) operations. Then, at each local DBMS, they generate a cost function for each class of query operations by fitting the cost model to a cost data set generated by executing query operations randomly selected from the class. Differently from the works in [23, 24], this method uses the entire real data actually used in a local DBMS.

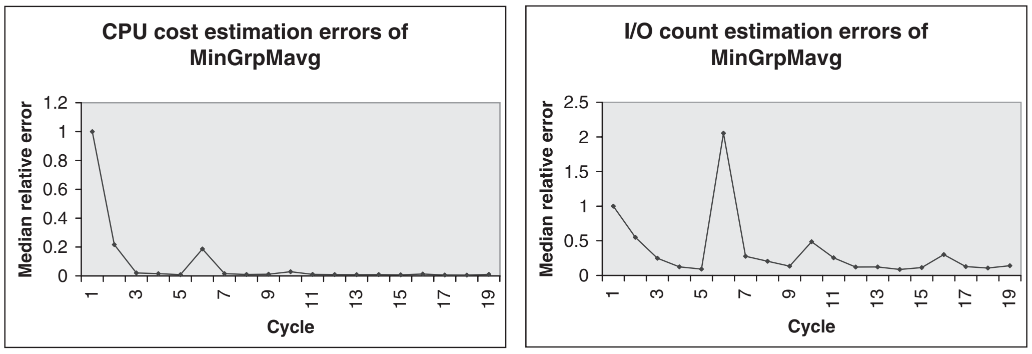
Both the model calibrating and query sampling methods aim at facilitating cost function generation in the data profile approach. However, they still require users to understand the concepts like index-based table scanning, and be capable of building cost models from the DBMS implementations of query operations like select and join. Furthermore, these methods assume users know the database objects (e.g. tables, indexes) accessed by a query, but this assumption is not necessarily true when dealing with a UDF.

5.3. Self-tuning modeling

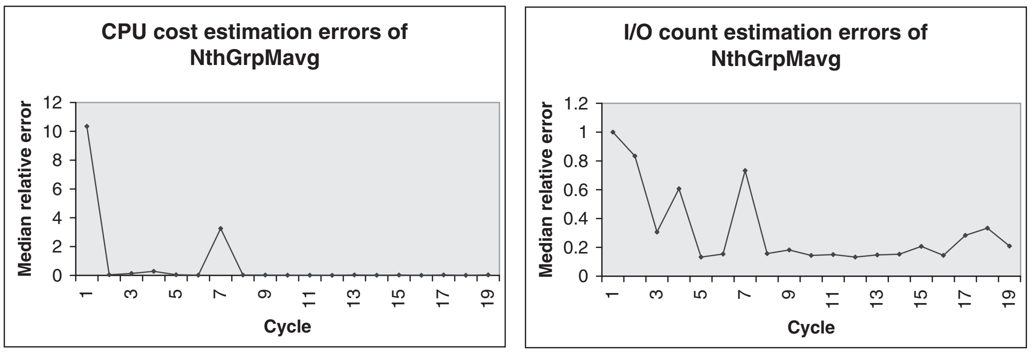
Recently there have been efforts for building a self-tuning DBMS [28], as exemplified by the automin project [29] at Microsoft Corporation. Self-tuning DBMSs are able to automatically tune themselves to application needs and hardware capabilities, thus reducing the administration overhead significantly. In this section, we provide a quick survey of existing works that are using self-tuning techniques and distinguish them from our work.

Chaudhuri *et al.* [29] 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.

There have been several papers presenting the self-tuning approach for estimating the selectivity of simple predicates (i.e. predicates on relational attributes) [30, 31, 32]. Chen and Roussopoulos [30] present a query feedback-based approach to estimating selectivity without accessing the actual tuples in the database. They use a cumulative data distribution function to estimate the selectivity for a range query by estimating the values at the two extreme points of the query range. The modeling technique used is polynomial regression, and an algorithm similar to ours is used for incremental updates of the regression model. Aboulnaga and Chaudhuri [31] and Bruno *et al.* [32] present a self-tuning approach to building and maintaining a histogram to estimate the selectivity. 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 with these works, but is distinct for estimating the UDF execution cost instead of selectivity. Note that their works address the selectivity of simple predicates, not UDF predicates.



(a) MinGrpMavg



(b) NthGrpMavg

FIGURE 13. Cost estimation errors on localized data sets.

In [33], Stillger *et al.* present 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. Unlike our approach which performs tuning at the level of a UDF which is executed as one step within a query, their approach performs tuning at the entire query level and, therefore, incurs higher overhead to collect the statistics needed for the tuning. Moreover, different types of query predicates need separate tuning processes.

In [34], Lee *et al.* present a self-tuning approach to data placement in a shared-nothing parallel database system. If a load imbalance happens, it determines the amount of data to be moved from the overloaded node and integrates the moved data into selected destination nodes. Although called ‘self-tuning’, this work is essentially about dynamic resource allocations and is closer to a trigger-action mechanism. Thus, their work is different from the continuous self-tuning mechanism supported in our work.

6. CONCLUSIONS

6.1. Summary

We have presented a self-tuning approach to building and maintaining the cost functions of 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.

6.2. Further work

The current framework may not work well for UDFs with non-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. One 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 two alternative techniques—multi-dimensional quadtree and non-parametric regression. Our eventual goal is to evolve the STM into a more generic framework that supports different kinds of modeling techniques.

We cannot always expect users to analyze the run-time of complex UDFs. They may not be capable of performing the task even if they have written the UDF codes. Automating run-time analysis would be useful in such a case. One approach is to incorporate code analysis techniques. The specific process is like this. First, the code of a UDF is analyzed and broken into fragments that configure the flow of the code. Second, a run-time model of each code fragment is derived based on its pattern, specifically, by choosing one among the run-time models associated with predetermined patterns. Third, the run-time model of the UDF is built as a combination of the run-time models of code fragments. We conjecture that the run-time model of each fragment can be a simple regression model like a quadratic model.

Another effort in the plan is to build the selectivity function of a UDF predicate, which is another important piece of statistics required by an ORDBMS query optimizer.

ACKNOWLEDGEMENTS

We thank the anonymous referees, whose comments were invaluable in improving the original manuscript to the current shape. We also 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. This research has been supported by the US Department of Energy through Grant No. DE-FG02-ER45962.

REFERENCES

[1] Seshadri, P. (1998) Predator: a resource for database research. *SIGMOD Record*, **27**(1), 16–20.

[2] Hellerstein, J. and Stonebraker, M. (1993) Predicate migration: optimizing queries with expensive predicates. In *Proc. ACM SIGMOD 93*, Washington DC, 26–28 May, pp. 267–276. ACM Press, New York.

[3] Hellerstein, J. (1994) Practical predicate placement. In *Proc. ACM SIGMOD 94*, Minneapolis, MN, 24–27 May, pp. 325–335. ACM Press, New York.

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

[5] Seshadri, P., Livny, M. and Ramakrishnan, R. (1997) The case for enhanced abstract data types. In *Proceedings of VLDB 97*, Athens, Greece, 26–29 August, pp. 66–75. Morgan Kaufmann, San Francisco.

[6] Seshadri, P. (1998) Enhanced abstract data types in object-relational databases. *VLDB J.*, **7**(3), 130–140.

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

[8] Boulos, J., Viemont, Y. and Ono, K. (1997) A neural network approach for query cost evaluation. *Trans. Inform. Process. Soc. Jpn*, **38**(12), 2566–2575.

[9] Lee, B., 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, USA. Downloadable from http://www.cs.uvm.edu/tr/CS-03-10.shtml.

[10] VanHorn, D., Lee, B., Buzas, J. and Thompson, P. (2003) *Metadata-based generation of statistical cost functions for text search*. Technical Report CS-03-14, Department of Computer Science, University of Vermont, Burlington, VT, USA. Downloadable from http://www.cs.uvm.edu/tr/CS-03-14.shtml.

[11] Barbara, D. *et al.* (1997) The New Jersey data reduction report. *IEEE Data Eng. Bull.*, **20**(4), 3–42.

[12] Raphaely, D. and Murray, C. (1991) *Oracle8i Time Series User's Guide (Release 8.1.5)*. Oracle Corporation, Redwood City, CA.

[13] McGregor, C. *et al.* (2001) *Oracle9i Text Application Developer's Guide (Release 9.0.1)*. Oracle Corporation, Redwood City, CA.

[14] Murray, C. *et al.* (2001) *Oracle9i Spatial User's Guide and Reference (Release 9.0.1)*. Oracle Corporation, Redwood City, CA.

[15] Graefe, G. (ed.) (1993) Query processing in commercial database systems. *IEEE Data Eng. Bull.*, **16**(4), 3–52.

[16] Gietz, B. *et al.* (2001) *Oracle9i Data Cartridge Developer's Guide, Release 1 (9.0.1)*. Oracle Corporation, Redwood City, CA.

[17] Elmasri, R. and Navathe, S. (2003) *Fundamentals of Database Systems* (4th edn). Addison-Wesley, Boston, MA.

[18] Frakes, W. and Baeza-Yates, R. (1992) *Information Retrieval—Data Structures & Algorithms*. Prentice Hall, Englewood Cliffs, NJ.

[19] David, K., Lawrence, K. and Keith, M. (1998) *Applied Regression Analysis and other Multivariable Methods*. Duxbury Press, CA.

[20] Neter, J., Kutner, M., Nachtsheim, C. and Wasserman, W. (1996) *Applied Linear Statistical Models*. McGraw-Hill Professional Publishing.

[21] Andres, F., Kwakkel, F. and Kersten, M. L. (1995) Calibration of a DBMS cost model with the software testpilot. In *Proc. CISMOD 95*, Bombay, India, 15–17 November, pp. 58–74. Springer-Verlag, Berlin.

[22] Ebrahimi, N. B. (1999) How to improve the calibration of cost models. *IEEE Trans. Softw. Eng.*, **25**(1), 136–140.

[23] Du, W., Krishnamurthy, R. and Shan, M.-C. (1992) Query optimization in heterogeneous DBMS. In *Proc. VLDB 92*, Vancouver, Canada, 23–27 August, pp. 277–291. Morgan Kaufmann.

[24] Gardarin, G., Sha, F. and Tang, Z.-H. (1996) Calibrating the query optimizer cost model of IRO-DB, an object-oriented

federated database system. In *Proc. VLDB 96*, Mumbai (Bombay), India, 3–6 September, pp. 378–389. Morgan Kaufmann.

[25] Zhu, Q. and Larson, P.-A. (1994) A query sampling method for estimating local cost parameters in a multidatabase system. In *Proc. ICDE 94*, Houston, TX, 14–18 February, pp. 144–153. IEEE Computer Society.

[26] Zhu, Q. and Larson, P.-A. (1996) Building regression cost models for multidatabase systems. In *Proc. PDIS 96*, Miami Beach, FL, 18–20 December, pp. 220–231. IEEE Computer Society.

[27] Zhu, Q., Sun, Y. and Motheramgari, S. (2000) Developing cost models with qualitative variables for dynamic multidatabase environments. In *Proc. ICDE 2000*, San Diego, CA, 28 February–3 March, pp. 413–424. IEEE Computer Society.

[28] Chaudhuri, S. (ed.) (1999) Self-tuning databases and application tuning. *IEEE Data Eng. Bull.*, **22**(2), 3–46.

[29] Chaudhuri, S., Christensen, E., Graefe, G., Narasayya, V. and Zwilling, M. (1999) Self-tuning technology in Microsoft SQL Server. *IEEE Data Eng. Bull.*, **22**(2), 20–26.

[30] Chen, C. M. and Roussopoulos, N. (1994) Adaptive selectivity estimation using query feedback. In *Proc. ACM SIGMOD 94*, Minneapolis, MN, 24–27 May, pp. 161–172. ACM Press, New York.

[31] Aboulnaga, A. and Chaudhuri, S. (1999) Self-tuning histograms: building histograms without looking at data. In *Proc. ACM SIGMOD 99*, Philadephia, PA, 31 May–3 June, pp. 181–192. ACM Press.

[32] Bruno, N., Chaudhuri, S. and Gravano, L. (2001) STHoles: a multidimensional workload-aware histogram. In *Proc. SIGMOD 2001*, Santa Barbara, CA, 21–24 May, pp. 211–222. ACM Press, New York.

[33] Stillger, M., Lohman, G., Markl, V. and Kandil, M. (2001) LEO—DB2's learning optimizer. In *Proc. VLDB 2001*, Roma, Italy, 11–14 September, pp. 19–28. Morgan Kaufmann.

[34] Lee, M., Kitsuregawa, M., Ooi, B., Tan, K. and Mondal, A. (1991) Towards self-tuning data placement in parallel database systems. In *Proc. ACM SIGMOD 2000*, Dallas, TX, 14–19 May, pp. 225–236. ACM Press.

APPENDIX A. FINANCIAL TIME-SERIES UDFS

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
    ti tsdev.ticker_index.ticker_index_id%TYPE;
    ts tsdev.tsquick_tab.tstamp%TYPE;
    tm tsdev.tsquick_tab.close%TYPE;
    tickdummy VARCHAR2(10);tickcount NUMBER:=0;
    reccount NUMBER := 0;
BEGIN
    //The subquery is used for reading the data of all ticker symbols
    //within the group and build a group average time series.
    //The outer query then calculates the group moving average.
    --Cost~(groupsize*(daterange+windowsize)+daterange*windowsize)
    OPEN CURSOR 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'))
         GROUP BY x.ticker_index_id,x.tstamp
         HAVING x.tstamp between to_date(sdate,'DD-MON-YY')
            AND to_date(edate,'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 NthGrpMvgAvg;
```

APPENDIX B. COST FUNCTION REGISTRATION THROUGH ODCI

```
--Create type minmavg_stat, which includes all the statistics functions.
CREATE OR REPLACE TYPE minmavg_stat AS OBJECT (
    curnum NUMBER,
    STATIC FUNCTION ODCIGetInterfaces(ifclist OUT sys.ODCIObjectList)
        RETURN NUMBER,
    STATIC FUNCTION ODCIStatsFunctionCost(func sys.ODCIFuncInfo,
        cost OUT sys.ODCICost, args sys.ODCIArgDescList,
        ticker char,sdate char, edate char, wsize NUMBER) RETURN NUMBER,
        PRAGMA restrict_references(ODCIStatsFunctionCost, WNDS, WNPS),
    STATIC FUNCTION ODCIStatsSelectivity(pred sys.ODCIPredInfo,
        sel OUT NUMBER, args sys.ODCIArgDescList, strt NUMBER, stop NUMBER,
        ticker char,sdate char, edate char, wsize NUMBER) RETURN NUMBER,
        PRAGMA restrict_references(ODCIStatsSelectivity, WNDS, WNPS)
)
/
-- Create type body minmavg_stat
--    a) Function ODCIGetInterfaces     : Oracle Internal function (mandatory).
--    b) Function ODCIStatsFunctionCost : Cost function (regression equation).
--    c) Function ODCIStatsSelectivity  : Selectivity function (default values).
CREATE OR REPLACE TYPE BODY minmavg_stat IS
    STATIC FUNCTION ODCIGetInterfaces(ifclist OUT sys.ODCIObjectList)
        RETURN NUMBER IS
    BEGIN
        ifclist := sys.ODCIObjectList(sys.ODCIObject('SYS','ODCISTATS1'));
        RETURN ODCIConst.Success;
    END ODCIGetInterfaces;
    STATIC FUNCTION ODCIStatsFunctionCost(func sys.ODCIFuncInfo,
        cost OUT sys.ODCICost, args sys.ODCIArgDescList,
        ticker char, sdate char, edate char, wsize NUMBER)
        RETURN NUMBER IS
        fname               VARCHAR2(30);
        tcount              NUMBER;
        range               NUMBER;
        size                NUMBER;
    BEGIN
        cost := sys.ODCICost(NULL, NULL, NULL);
        select (TO_DATE(edate,'DD-MON-YY') - TO_DATE(sdate,'DD-MON-YY')) date_range
                  into range
        from dual;
        range := range - (+16055.5);
        size  := wsize - (+49.44);
        cost.CPUCost :=  ROUND(
            ( 1 * (+11.8272450115377392) ) +
            ( range * (+0.0007452936798574) ) +
            ( size * (+0.0423070733974652) ) +
            ( (range*range) * (+0.0000000004402906) ) +
            ( (size*range) * (+0.0000025356459042) ) +
            ( (size*size) * (-0.0000188506209238)) ;
        cost.IOCost := ROUND(
            ( 1 * (+290.7927961404646453) ) +
            ( range * (+0.0194580835416424) ) +
            ( size * (-0.0427221861374802) ) +
            ( (range*range) * (+0.0000000224353175) ) +
            ( (size*range) * (+0.0000070627539797) ) +
            ( (size*size) * (+0.0010460367870459)) );
        RETURN ODCIConst.Success;
    END;
    STATIC FUNCTION ODCIStatsSelectivity(pred sys.ODCIPredInfo,
        sel OUT NUMBER, args sys.ODCIArgDescList, strt NUMBER, stop NUMBER,
        ticker char, sdate char, edate char, size NUMBER)
        RETURN NUMBER IS
    BEGIN
        sel := -1;
        RETURN ODCIConst.Success;
    END;
END;
/

ASSOCIATE STATISTICS WITH FUNCTIONS sys.minmavg USING minmavg_stat
/
```