A Statistical Cost-Modeling of Financial Time Series Functions for an Object-Relational DBMS Query Optimizer

Byung S. Lee[1,*], Vinod Kannoth[1], Jeff Buzas[2]

[1]Department of Computer Science, University of Vermont, Burlington, VT 05405

[2]Department of Mathematics and Statistics, University of Vermont, Burlington, VT 05405

E-mails: {bslee, buzas}@emba.uvm.edu, v2vinod@us.ibm.com

Abstract

Financial time series functions are in prevalent use in stock market analysis, and are so important in business applications as to be supported by every major commercial object-relational database management system (ORDBMS). These ORDBMSs require users to provide the cost functions of user-defined functions (UDFs) for their query optimizers. The traditional approach to developing a cost function is to build an *analytic* function of variables derived from data configurations and system configurations. This is an overwhelming task for users without advanced knowledge of the internal implementation of an ORDBMS.

In the case of financial time series functions, there is a *statistical* approach much easier for users. Users provide a set of variables influencing the costs (as we call the cost variables) based

*The contact author, e-mail: bslee@cs.uvm.edu, tel: (802)656-1919, fax: (802)656-0696

on their understanding of the algorithm and provide a specification for sampling the values of the variables. Then, the system collects data points by observing the execution costs at those sample values and builds a cost function by applying a regression model to the collected data set. The user can specify a model based on the run-time analysis, or the system uses a default quadratic model of the user-provided cost variables.

We experiment with aggregate financial time series functions as the UDFs. We compare a model based on run-time analysis and a model fully quadratic. The results from the considered UDFs show less than 5% median relative errors for both models. Interestingly, the quadratic model achieves slightly smaller errors than the run-time analysis model, thus suggesting that the former model may well substitute the latter model given the cost variations of the UDFs. In addition, we validate the resulting statistical cost functions by applying them to the extensible query optimizer of a commercial ORDBMS.

Keywords: financial time series function, cost function, extensible query optimizer, object-relational database management system, quadratic regression model, run-time analysis model

1 Introduction

Financial time series functions are used extensively in stock market analysis, and are so important in business applications that they are supported by almost every major object-relational database management system (ORDBMS) in the form of a module extending the server DBMS. Examples are Data Cartridges from Oracle, Extenders from IBM-DB2, DataBlades from IBM-Informix, and OLE-DB from Microsoft SQL Server.

These ORDBMSs require users to provide cost functions of user-defined functions (UDFs) for their extensible query optimizers. A traditional cost function is an analytic function defined with variables derived from a data profile describing data configurations (e.g., cardinality, selectivity, blocking factor) and system configurations (e.g., buffer size, disk page size). Typically, identifying and deriving these variables require advanced knowledge of DBMS internal implementations. This is an overwhelming task for most users. However, there is an easier way for financial time series functions.

In this paper we generate the cost functions of these UDFs as statistical functions, with only light requirements for users. Users are required to provide the variables that influence the cost (as we call the cost variables) and a specification for generating a data set of execution costs. Cost variables are identified or derived from the UDF based on users' understanding of the algorithm. Then, the system generates a data set according to the user-provided specification and builds a cost function by applying a regression model to the generated data set. The user can specify a model based on run-time analysis, or the system uses a default quadratic model of user-provided cost variables. The rationale behind this quadratic model is given in Section 3.1.

In our work the cost data set is generated using a grid-based "*parade of runs*" similar to the one used by Boulos and Ono in [8]. This method executes a UDF repeatedly for grid sample values of cost variables within the ranges provided by a user. This technique is simple, and guarantees to cover all regions of the data space.

The UDFs considered in our work are extended from the conventional moving average function[1]. The UDFs show cost variations in the linear-logarithmic scale while the basic moving average does

in the linear scale. In general, the costs of all financial time series functions are characterized by their smooth, continuous, and monotonous variations and, therefore, are favorable to a quadratic model.

We perform experiments using several financial time series functions including NthMavg and NthGrpMavg. The former calculates the n-th smallest moving average of a particular ticker data, and the latter calculates the n-th smallest moving average of a group of ticker data. We compare two alternative regression models. One is based on the run-time analysis of an algorithm, and the other is a full quadratic equation of cost variables. We use generic cost metrics, that is, the CPU time and the number of disk pages fetched from disk to main memory. In addition to the data set fit with the models, we generate another data set for testing the accuracy of cost estimation. This is done through another parade of runs based on random sample values of cost variables.

The experimental results show that the median relative errors of the estimated costs are smaller than 5% for both models. This remarkable precision demonstrates how effective the proposed cost modeling approach is for financial time series functions. Moreover, the difference in the cost estimation errors of the two models is negligible, which indicates that a quadratic model is as precise as a run-time analysis model for the UDFs considered.

In addition, we validate the generated statistical cost functions in a commercial ORDBMS. Specifically, we implement the cost function part of Oracle's Data Cartridge Interface (ODCI)[3] as the quadratic regression equations. The results show that the regression models do affect the ordering of predicates involving UDFs (called UDF predicates) as specified for general cost functions in [3].

This paper is organized as follows. After providing background information in Section 2, we

describe our approach to building cost models and the evaluations in Section 3. We validate the resulting cost functions using a commercial ORDBMS in Section 4. Related works are described in Section 5. We discuss ongoing efforts for generalizing the presented work in Section 6, and conclude the paper in Section 7.

2 Background

In this section we provide a quick overview of the financial time series functions and the extensible query optimizer of an ORDBMS.

2.1 Financial time series functions

A time series refers to a sequence of time-stamped data, and a *financial* time series in particular is a sequence of daily financial summary data. Time series in general may be regular or irregular depending on whether they have an associated calendar. A regular time series has an associated calendar, and data arrive predictably at intervals defined by the calendar. An example is a series of daily stock market summaries, such as the trade volumes and opening, high, low, and closing prices. Its calendar is defined by the frequency (e.g., day), pattern (e.g., '011110' for data during five weekdays and not during weekends), minimum and maximum timestamp dates, and exception dates (e.g., weekdays with no data or weekends with data). An irregular time series does not have an associated calendar, and data arrive unpredictably at unspecified points in time. An example is a series of account transactions (e.g., deposits, withdrawals) at a bank teller machine. An irregular time series may have long periods with no data or short periods with bursts of data.

There are several different kinds of financial time series functions, but of particular interest in this paper are aggregate functions. Each aggregate function is of the form[1]:

f(ts ORTDNumSeries, [startdate DATE, enddate DATE])

where the function f can be any of avg, count, min, minN, median, max, maxN, product, sum, stddev, and variance. These functions are applied to a numeric time series (ORDTNumSeries) in a date range bounded by startdate and enddate. These functions return single numbers as a result of processing the data and, therefore, can be used in a UDF predicate of the form "UDF(input_arguments) op constant" (where op denotes a comparison operator). Besides, they are computationally expensive enough to necessitate a cost function for a query optimizer.

The cost variations of all these aggregate functions are either linear or linear-logarithmic with the date range as the only cost variable. In this paper we create more complex aggregate UDFs that involve two to three cost variables. They will be described in Section 3.2. The UDFs considered are extensions of the moving average function illustrated in Figure 1. For a given stock ticker symbol, its moving average is defined as the average of daily stock closing prices computed within a gliding k-day window between start date and end date. As a result, given a time series of daily closing prices, moving average returns another time series of window averages.

2.2 Extensible query optimizer of an ORDBMS

Traditionally a cost estimate-based query optimizer generates a query execution plan based on the estimated costs of alternative plans. The cost of each plan is the accumulation of the costs of all

[1] Based on Oracle8i Time Series[1]

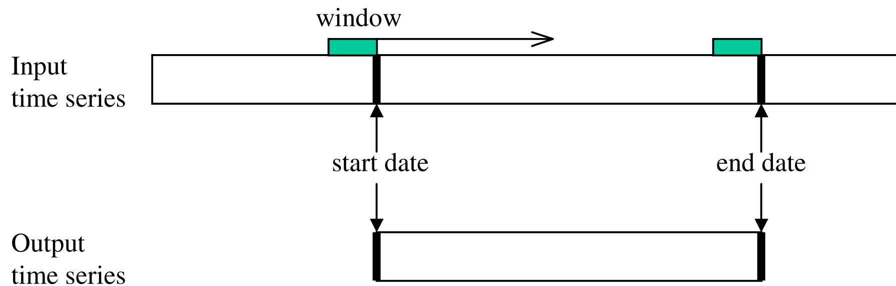


Figure 1: A moving average function.

operations in it, and each operation has an associated cost function that estimates the CPU usage and disk access. A cost function uses variables instantiated to estimate the cost. Some variables are specific to the system (e.g., disk block size, main memory buffer size), and some are specific to the database (e.g., cardinality and tuple size of a relation, cardinality of distinct column values). Collectively we call these kinds of variables the data profile variables. Typically DBMS developers who develop the routines executing individual operations provide those cost functions.

An ORDBMS's extensible framework [2, 3] provides an interface for incorporating UDF cost functions into its cost estimate-based query optimizer. Three cost functions are needed for each UDF, one for each of the CPU cost, disk I/O cost, and network I/O cost. The metrics of these cost components should be immune to the change of system environment as much as possible. Therefore, for example, Oracle's extensible query optimizer uses the following metrics [3].

- CPU cost: the number of machine instructions executed by the CPU, excluding the overhead of invoking the UDF.

- Disk I/O cost: the number of data pages transferred from disk to main memory buffer.

- Network I/O cost: the number of data blocks transmitted via the network.

Our approach uses the first two cost metrics. The third metric is not considered here because it is not actually used by any ORDBMS yet.

3 Generating Cost Functions

In this section we describe some particulars of our cost model building approach and the experiments performed to evaluate the approach.

3.1 Cost model building

In principle, the cost of executing a UDF can be determined exactly if one knows the cost variables and the functional form (cost function) relating the cost variables to the actual cost. In practice, however, the exact cost function and cost variables are typically unknown. Therefore, an approximation in the form of cost model building is required.

We consider two approaches to building a cost model. One is based on run-time analysis (RTA) of the UDF, which involves identifying cost terms from the algorithm, building a regression equation as a linear combination of the terms, and tuning the coefficients. The other approach is to use a full quadratic (Quad) model, which involves identifying cost variables, building a full quadratic model with them, and tuning the coefficients. The quadratic models we employ are equivalent to the second order Taylor approximation to the true but unknown cost function. Second order approximations are used extensively in a wide variety of applications [17] because the approximation is quite good provided the true unknown function is reasonably smooth.

A cost variable is either ordinal or nominal. A nominal one cannot be used in a cost function

because of its random effect on the execution cost, while it is possible to build separate models for individual values of a nominal variable if the cardinality is manageable. In our experiment below, we consider only ordinal case by using regular time series data.

In general the available data buffer size influences the disk I/O cost and, therefore, may well be a cost variable. However, we do not consider it for two reasons. First, the data buffer size is changed very infrequently. Second, typical financial time series functions perform a linear scan of data and, thus, access data pages at most once regardless of the buffer size. For other functions accessing data pages repeatedly (e.g., for sorting), we need to regenerate the cost functions if the buffer size changes significantly.

3.2 Evaluations

In this section we evaluate the cost modeling performed using the two time series UDFs NthMavg and NthGrpMavg.

Experimental UDFs: The two UDFs have the following signatures.

- NthMavg(tickersymbol, startdate, enddate, windowsize, n)
- NthGrpMavg(groupsymbol, startdate, enddate, windowsize, n)

Given a ticker symbol, NthMavg returns the n-th minimum of moving averages calculated within a specified date interval. NthGrpMavg extends NthMavg by considering a *group* of ticker symbols instead of a single one. That is, given a group symbol (e.g., NASDAQ), it returns the n-th minimum moving average of the group average time series. Selecting the n-th minimum involves sorting, for

which we use mergesort. These two UDFs are complex enough to be selected as the experimental UDFs considering that typical ones such as moving sequence functions (e.g., moving average/sum) and cumulative sequence functions (e.g., cumulative average/sum/min/max) involve only linear scan of data without sorting. Both functions are implemented in Oracle PL/SQL.

There are two other UDFs used in the experiment: MinMavg and MinGrpMavg, which calculate the minimum values with linear scans only and no sorting. However, we do not discuss them in this paper because their costs vary linearly with cost variables so that the cost functions are trivial compared with those of NthMavg and NthGrpMavg. Besides, their experimental results do not address anything different from those of NthMavg and NthGrpMavg.

Appendix A shows the algorithm of NthGrpMavg in a simplified PL/SQL syntax and the cost formula associated with each major step of the algorithm. First, in Lines 7∼14, it looks for data records pertinent to all ticker symbols belonging to the group named groupsymbol, scans the records from windowsize ahead of startdate to enddate, groups the scanned records by their `tstamp` (i.e., date), and calculates the average of all `close`'s (i.e., ticker closing prices) in each group (i.e., day). Second, in Lines 15∼19, it fetches the daily group averages one by one and saves them into a temporary array `temp` (or, precisely, an indexed one-column table). Third, in Lines 20∼26, it calculates the moving average of the data in the array `temp` and saves the resulting moving average time series data into another temporary array `tempavg`. Last, in Lines 27 and 28, it sorts `tempavg` using mergesort and returns the n-th element. Since mergesort takes $O(N \log_2 N)$ for a file with N elements, we assume the last step takes time proportional to (daterange+2)$\log_2$(daterange+2) where daterange+2 is the size of `tempavg`. We do not show the algorithm of NthMavg here. It

is similar to the algorithm of NthGrpMavg except the step of generating a group moving average time series.

Experimental time series data: Figure 2 shows the schema of financial ticker time series data used in the experiment. The schema `tsdev` contains two tables. The table `ticker_index` is an index to ticker symbols that are members of a group, and the table `tsquick_tab` is a table that contains the time series data of a particular ticker symbol. Data records of `tsquick_tab` are sorted by their primary key fields.

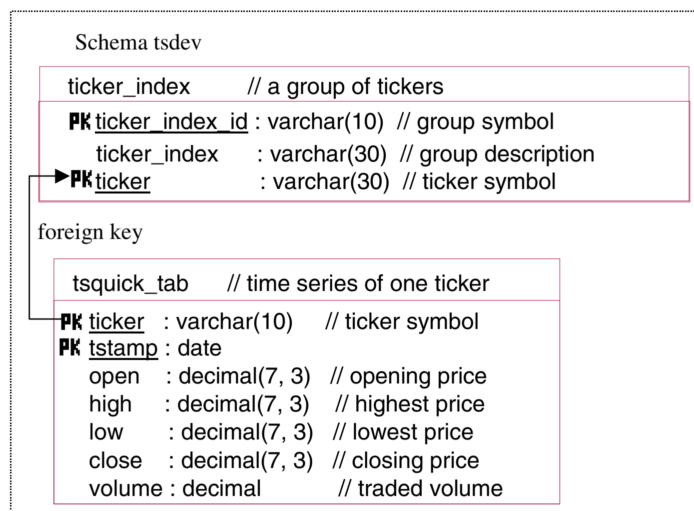


Figure 2: Schema of the experimental time series data.

The volume of time series data processed by the UDFs is easily increased by replicating the records stored in `tsquick_tab` to different ranges of the value of column `tstamp`. The data thus expanded amounts to a date range of 80 years, or 29220 days.

Calculating a group average in NthGrpMavg requires reading records across ticker symbols at each `tstamp`. For this reason, the step of generating a group time series in NthGrpMavg incurs a

significant overhead.

Parade of runs: For each UDF, we compare its RTA model and Quad model by fitting to a training data set and evaluating the accuracy of the model using a test data set. Both data sets are generated through parade of runs.

Prior to the parade of runs, we set the following finite range of values for each ordinal cost variable: the date range of 0 to 80 years, the window size of 1 to 60 days, and the group size of 1 to 20 ticker symbols. Within these ranges, we take grid sample points to generate a training data set so that the samples cover the entire data space evenly. In contrast, we take random sample points to generate a test data set because they mimic the values of cost variables used in actual UDF executions.

For the training data sets, we take 12 date ranges and 4 window sizes for NthMavg so that there are $4 \times 12 = 48$ data points, and 8 date ranges, 4 window sizes, and 4 group sizes for NthGrpMavg so that there are $8 \times 4 \times 4 = 128$ data points. The test data sets have the same number of data points as the training data sets. The number is higher for NthGrpMavg because it has one more cost variable than NthMavg. These numbers are large enough to render the data set space sufficiently dense, and in fact fewer data points are typically used to fit second order models.

During the parade of runs, we clear the cache by filling up the data buffer with dummy data between each run. This removes the unpredictable effect of data caching on repeated executions of a UDF. (Most DBMS query optimizers either disregard the caching effect or oversimplify it [4].) Besides, we increase the odds of clearing the same pages from the operating system buffer by having the dummy data much larger than the data buffer size. (The OS caching effect could

be eliminated entirely if we used direct I/O, which is often used in a DBMS to ensure forced disk writes of log records.) Occasionally, buffer space shortage due to excessive workload may cause extra disk I/O as some pages are flushed and refetched. We disregard this workload effect because we have observed that their effects are trivial compared with the buffer caching effect. Workload effects are easily trivialized through tuning the system configuration (e.g., buffer size, page size).

Among the three generic cost metrics mentioned in Section 2.2, we use CPU cost and disk I/O cost, but do not consider network I/O cost because the ORDBMS we use does not handle it yet. Particularly for the CPU cost, we use CPU time as the metric instead of the number of machine instructions. CPU time can be converted to the number of machine instructions as CPU time $\times$ CPU speed / the average cycles per instruction. (We use an arbitrary number 1.5 as the average cycles per instruction. The actual number is immaterial to the results of our experiment.) As for the computing platform, we use Sun Ultra Enterprise 450 with four 276MHz CPUs, 1024 Mbyte RAM, and 55 Gbyte hard disk.

Cost models Since we use *regular* time series data, the costs of NthMavg and NthGrpMavg are independent of the ticker symbol and start date. From the algorithm of NthGrpMavg shown in Appendix A, we identify three cost variables: daterange as enddate $-$ startdate, windowsize as is, and groupsize as the number of ticker symbols in the named group. Then, using these variables, we obtain the RTA model and the Quad model as follows. In the equations below, $c_0, c_1, c_2, \ldots, c_9$ denote coefficients (or, parameters) and D, W, G denote daterange, windowsize, and groupsize, repectively.

- RTA model for NthGrpMavg: By analyzing the algorithm, we produce the following run-

time formula consisting of four cost terms, each of which corresponding to each step of the NthGrpMavg algorithm in Appendix A.

$$Cost = c_0 + c_1 G(D+W+1) + c_2(D+W+1) + c_3(D+2)W + c_4(D+2)\log_2(D+2) \quad (1)$$

- Quad model for NthGrpMavg: Given the three cost variables D, W, and G, the model is:

$$Cost = c_0 + c_1 D + c_2 W + c_3 G + c_4 D^2 + c_5 W^2 + c_6 G^2 + c_7 DW + c_8 WG + c_9 GD \quad (2)$$

Similarly, the RTA model and the Quad model of NthMavg are as follows. NthMavg derives only two cost variables datarange and windowsize.

- Run-time analysis of NthMavg returns the following model, of which the terms are slightly different from the corresponding terms in the RTA model of NthGrpMavg.

$$Cost = c_0 + c_1 D + c_2(D - W + 1)W + c_3(D - W + 1)\log_2(D - W + 1) \quad (3)$$

- Quad model of NthMavg given the two cost variables is

$$Cost = c_0 + c_1 D + c_2 W + c_3 D^2 + c_4 W^2 + c_5 DW \quad (4)$$

Cost estimation Table 1 shows the cost estimation errors for the two UDFs, specifically the mean absolute error (mae) and the mean (mre) and median (dre) of relative errors (i.e., the ratio of the residual error and the observed value). The cost estimation is done only within the range of cost variables used in the parade of runs because extrapolation outside the range may not accurately reflect the actual costs. In the table, median relative errors are more credible than mean relative errors. For instance, the mean relative error 73.3% of the NthMavg CPU cost for RTA is misleading

because the high error is attributed to two data points with very small CPU costs. (Estimated costs 26.5 and 31.1 seconds for observed costs 2.12 and 1.76 seconds.)

	CPU cost						Disk I/O cost					
	RTA model			Quad model			RTA model			Quad model		
UDF	mae	mre	dre	mae	mre	dre	mae	mre	dre	mae	mre	dre
NthMavg	7.98	73.3	3.83	1.14	1.47	0.55	12.88	17.5	2.49	16.09	13.6	4.18
NthGrpMavg	11.23	10.0	2.41	2.71	1.41	0.76	323.68	1.44	1.29	226.72	0.98	0.71

mae = mean absolute error (seconds for CPU, pages for disk I/O), mre = mean relative error (%), dre = median relative error (%)

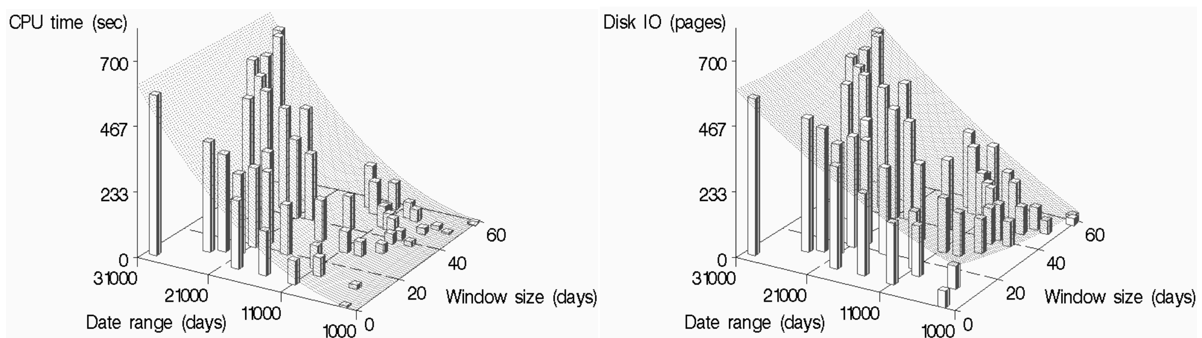
Table 1: Cost estimation errors of NthMavg and NthGrpMavg.

As we can see in Table 1, the modeling errors of the Quad approach are comparable to those of the RTA approach and even slightly lower. Apparently the run-time analysis only approximates the true cost function, whereas the quadratic model contains more terms, which allows for additional flexibility in estimating the costs.

Based on the run-time analysis, the true cost functions of both NthMavg and NthGrpMavg are nonlinear with the cost variables. For example, the RTA model for NthGrpMavg contains the term $(D + 2)\log_2(D + 2)$. It is evident from Table 1 that the Quad model provides an excellent approximation to the nonlinearity. This is encouraging because it suggests that the simple quadratic model may be adequate for generating the cost functions of other UDFs that show nonlinear but reasonable smooth and monotonous cost variations.

Figure 3 shows plots of the CPU and disk I/O costs of NthMavg, and Figure 4 shows those of

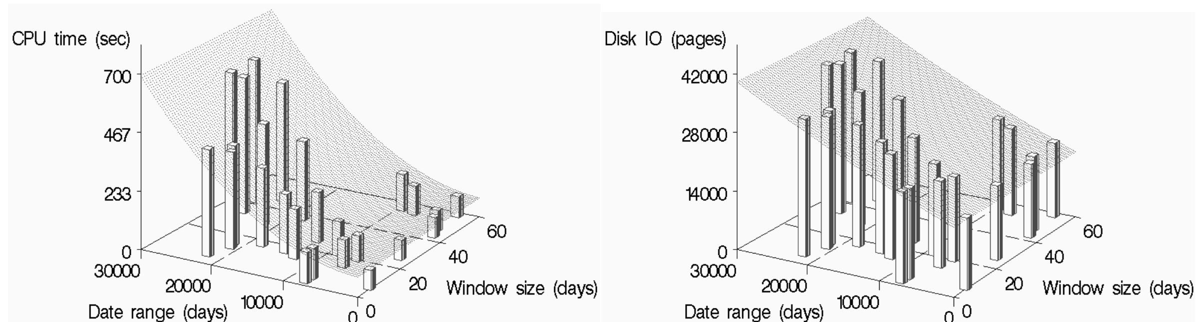
NthGrpMavg for group size 20. The observed costs (i.e., test data points) are shown as vertical bars, and the estimated costs are shown as a dotted surface. The plots look identical for both models. We can confirm visually that the estimated costs match the observed costs quite accurately. There is a caveat, however. While such a close match is consistently true for the CPU cost, it is not for the disk I/O cost. The observed cost randomly deviates from the estimated cost due to the unpredictable effects of data caching. What we see in Figure 4a and Figure 4b is after suppressing the caching effect at every run.



(a) CPU cost.

(b) Disk I/O cost.

Figure 3: Observed and estimated costs of NthMavg.



(a) CPU cost.

(b) Disk I/O cost.

Figure 4: Observed and estimated costs of NthGrpMavg for group size 20.

The overhead of a parade of runs, specifically the computation time, can be significant because it increases in proportion to the total number of sample points. However, compared with the histogram approach used in [8], the regression technique allows us to reduce the data set to a manageable size at a far smaller expense of estimation error. In the case of the two UDFs considered here, 48 and 128 data points proves far more than necessary to fit the RTA and Quad cost models, because reducing he number of data points to the minimum, that is, 3 points for each cost variable, does not increase the cost estimation error noticeably.

4 Validation of the Generated Cost Functions

In this section we demonstrate the practicality of our work by applying the generated regression models as cost functions to a commercial ORDBMS and verifying their influences on the order of evaluating query predicates. We first introduce Oracle's extensible query optimizer briefly and then present the performed experiments and the results.

4.1 Extensible query optimizer of Oracle

Oracle Data Cartridge [3] allows for an extension of the query optimizer to handle UDF[2] predicates (e.g., udf('01/01/1985', 50000, 'Sales') $>$ 60). This query optimizer expects users to provide two functions associated with each UDF predicate – one for estimating the execution cost of the UDF and the other for estimating the selectivity of the predicate – and uses them in the same manner as the cost and selectivity of a plain relational predicate (e.g., Employee.salary $>$ 50000). In case no

[2]A UDF in Oracle ORDBMS may be an object method, a stand-alone function, or a package function.

function is provided, Oracle uses a system default value.

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 and selectivity functions as the components of a "statistics object." Figure 5 sketches registering the CPU and disk I/O cost functions of NthMavg as the quadratic regression equations shown in Equation 4. ODCIStatsFunctionCost is an ODCI function.

```
Input:
·     startdate, enddate, windowsize: input arguments of NthMavg
Output:
·     Cost: an object of type ODCICost, consisting of CPU, disk I/O, and network I/O costs
·     Success:  a flag indicating a successful completion
Static Function ODCIStatsFunctionCost:
{
   Initialize Cost to null;
   // Cost variables are daterange and windowsize.
   daterange := select (enddate – startdate) from dual;
   // Center the cost variables.
   D := daterange – 16055.5;
   W := windowsize – 49.44
   // Cost function expressed as a regression equation.
   Cost.CPUCost := 11.8 + 0.000745*D + 0.0423*W +
                        0.000000000440*D*D + 0.00000253*D*W – 0.0000188*W*W;
   Cost.IOCost     := 290. + 0.0194*D - 0.0427*W +
                        0.0000000224*D*D + 0.00000706*D*W + 0.00104*W*W;
   return(Success, Cost);
}
```

Figure 5: Registering the cost functions of NthMavg through ODCIStatsFunctionCost.

4.2 Experiments

For our experimental purpose, we introduce another UDF Foo whose cost function returns a default value different from the system default. In addition, we exclude the *selectivity* of a UDF predicate from consideration, thus always using the system default. There are two reasons for this exclusion. First, generating a selectivity function is amenable to histogram techniques [18, 19, 20] but not to regression, and thus is beyond the scope of this paper. Second, the selectivity of a UDF predicate is independent of the cost of the UDF and, therefore, leaving it out does not invalidate the experimental results based on the cost only.

Specifically, the experiment aims at demonstrating that the statistical cost function affect the order of evaluating two UDF predicates that respectively involve NthMavg and Foo. For this purpose, we create a test case that involves three tables – Employee, Company, and TSQuick – and one SQL query like the one shown in Figure 6.

```
Tables:
   Company (ticker, name, type, hq_location, category, turnover, profit_loss, date_estd)
   Employee (employee_id, employee_name, birth_date, salary, company, title,
                  employee_background, email, date_of_hire)
    TSQuick (ticker, open, high, low, close, volume)
Query:
    select em.employee_name, em.company
   from tsdev.employee em, tsdev.company co, tsdev.tsquick ts
  where trim(co.ticker)=trim(ts.ticker)
    and sys.Foo()
    and trim(em.title) = 'CEO‘ and em.salary > 1950000
    and em.company= co.ticker and trim(co.type) = 'C corp‘
    and tsdev.NthMAvg(ts.ticker,'01-MAR-00','01-DEC-70',10,200) > 60;
```

Figure 6: A UDF validation test case.

In [3] it is stated that Oracle query optimizer evaluates the predicates specified in a “where”

clause in the following order.

1. Non-UDF predicates, in the order specified in the clause
2. UDF predicates with associated cost functions, in an increasing order of the costs
3. UDF predicates without associated cost functions, in the order specified in the clause

Based on this ordering rule, we consider three cases of providing the cost functions of NthMavg and Foo differently for the test query in Figure 6. We use the Oracle TKPROF trace utility [21] to observe the order of predicate evaluations because its trace output lists the statements in the order of their first execution.

Table 2 summarizes the results. The results in all three cases are consistent with the rules mentioned above, thereby validating the cost functions. In Case 1, neither UDF has an associated cost function registered, thus using system default costs. In this case, their predicates are evaluated in the order they appear in the "where" clause. (We tried both orders and verified its consistency.) In Case 2, only NthMavg has an associated cost function. Specifically, we use a constant cost function in Case 2A and the quadratic model shown in Equation 4 in Case 2B. The actual constant costs used in Case 2A are immaterial.

In Case 3, both UDFs have associated cost functions. Foo's cost function returns constant costs – 100,000,000 machine instructions for the CPU cost and 300 disk pages for the disk I/O cost. (These constant costs are roughly the median of the CPU costs and disk I/O costs in the NthMavg data sets. We assume one million machine instructions per second for converting the CPU time to the number of machine instructions.) In comparison, NthMavg's cost function is the quadratic

Test Cases	Test Results
Case 1: • Foo: no cost function • NthMavg: no cost function	The two UDF predicates are evaluated in the same order as listed in the query.
Case 2A: • Foo: no cost function • NthMavg: constant cost function	The NthMavg predicate is evaluated before the Foo predicate.
Case 2B: • Foo: no cost function • NthMavg: quadratic cost function.	The NthMavg predicate is evaluated before the Foo predicate.
Case 3A: • Foo: constant cost function (CPU: 100000000 instructions, IO: 300 pages) • NthMavg: quadratic cost function with startdate 01-MAR-00 and enddate 01-DEC-10.	The NthMavg predicate is evaluated before the Foo predicate. (Cost of NthMavg < cost of Foo)
Case 3B: • Foo: constant cost function (CPU: 100000000 instructions, IO: 300 pages) • NthMavg: quadratic cost function with startdate 01-MAR-00 and enddate 01-DEC-72.	The Foo predicate is evaluated before the NthMavg predicate. (Cost of NthMavg > cost of Foo)

Table 2: Summary of results from the cost function validation tests.

model in Equation 4, with two different cases of the cost: lower than Foo's cost due to the shorter date range in Case 3A and higher than Foo's cost due to the longer date range in Case 3B.

5 Related Works

We find related works in the following two aspects: generating the cost functions of UDFs and using regression techniques for doing it.

5.1 UDF cost function generation

There have been noticeable research efforts on optimizing queries involving UDF predicates [5, 6, 7]. However, all these efforts assume the cost functions of UDFs are already available and do not deal

with the issue of generating them. Our survey finds only one research work on generating cost functions, conducted by Boulos and Kinji in [8] and [9]. In [8], the authors present a technique based on multi-dimensional histograms. The approach is to execute a UDF repeatedly for different values of its input arguments and construct a multi-dimensional histogram from the observed execution costs. In this approach, however, the data set becomes too voluminous to be stored and processed with reasonable performance. The authors mitigate this problem by sampling the generated data sets but, as a result, incur high errors (55% – 79% relative error) in the cost estimation. In [9], the authors 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 Using regression techniques

Andres et al. [10] model DBMS performance as a regression equation and tune its coefficients by running a set of representative workload, and Ebrahimi [11] uses a similar approach to tune the coefficients of software (not database) cost model. In addition, 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 [12, 13] and query sampling [14, 15, 16]. Du et al. in [12] 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. in [13] extend Du et al.'s work to an object-oriented query optimization.

In the query sampling method [14, 15, 16], 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. Unlike [12, 13], 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.

6 Generalization Efforts

There are a few efforts being made to generalize the work beyond its current scope. We discuss them here.

We have used financial aggregate time series functions as the representatives of UDFs that show smooth and monotonous cost variations. The same cost modeling approach can be used for any other kinds of UDFs with the same characteristics of cost variations. Currently we are extending our approach to text search functions with a particular attention to identifying dominant cost variables in the midst of the "disturbing factors" such as the nominal characteristics of search keywords and the unpredictable effects of a theme in a theme query, keywords and the maximum spanning distance

in a proximity query, stemmed words in a stem query, and the threshold of occurrence frequencies in a threshold query. (Readers are referred to [22] for details of these functions.)

There is another effort, which aims at resolving the difficulty of handling nominal cost variables and reducing the overhead of parades of runs. We are currently working on an alternate approach in which an ORDBMS query optimizer updates a cost function incrementally and continually using a data set collected from the most recent executions of a UDF. This approach fits in the recent trend of a self-tuning DBMS [23].

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. This brings home the need for automating run-time analysis. One viable 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.

7 Conclusion

We have presented a statistical regression approach to generating the cost functions of financial time series functions for an ORDBMS query optimizer. In this approach, users are asked to identify cost variables and specify how to sample the values of the variables. Users may provide a model

based on run-time analysis, or the system uses a fully quadratic model of the cost variables. Then, the system generates a cost function by fitting the model to a data set generated by executing the UDF repeatedly for the sample values of cost variables.

We have considered two UDFs implemented in Oracle PL/SQL: NthMavg (for calculating the n-th minimum moving average) and NthGrpMavg (for calculating the n-th minimum group moving average). Both functions are aggregate functions based on financial time series data and have been extended from the commonly used moving average function. They have nominal input arguments but allow them to be converted to ordinal cost variables.

We have considered and compared two kinds of cost models: run-time analysis model and quadratic model. The results show smaller than 5% median relative errors for both models. The results also show that the quadratic model is slightly better than run-time analysis model despite its simplicity. This suggests that users may do without the burden of run-time analysis.

In addition, we have used the generated regression models as cost functions in a commercial ORDBMS and verified their influence on ordering query predicates involving the UDF, thus demonstrating the practicality of our approach. Finally, we have discussed our ongoing efforts that generalize the work presented in this paper.

Acknowledgments

This research has been supported by the US Department of Energy through Grant No. DE-FG02-ER45962. We thank Dr. Ghaleb Abdulla at the Lawrence Livermore National Laboratory for his comments on the initial draft of the paper.

References

[1] Oracle8i Time Series User's Guide – Release 2 (8.1.5), Oracle Documentation Library, 2000, Chapter 5.

[2] UDB Administration Guide: Performance, DB2 Universal Database and DB2 Connect for Windows,OS/2 and UNIX Version 6 Publications, DB2 Product and Service Technical Library, 2000, Chapter 20.

[3] Oracle 8i Data Cartridge Developer's Guide - Release 2 (8.1.6), Oracle Documentation Library, 2000, Chapter 8.

[4] Peter Gassner, Guy M. Lohman, K. Bernhard Schiefer, and Yun Wang, "Query Optimization in the IBM DB2 Family," Data Engineering Bulletin, Vol. 16, No. 4, 1993, pp. 4-18.

[5] S. Chaudhuri and K. Shim, "Optimization of Queries with User-defined Predicates," Proceeding of the International Conference on Very Large Data Bases, Mumbai, India 1997.

[6] J. Hellerstein, "Practical Predicate Placement," Proceeding of the ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, 1994.

[7] J. Hellerstein and M. Stonebraker, "Predicate Migration: Optimizing Queries with Expensive Predicates," Proceeding of the ACM SIGMOD International Conference on Management of Data, Washington, D.C. 1993.

[8] Jihad Boulos and Kinji Ono, "Cost Estimation of User-Defined Methods in Object-Relational Database Systems," SIGMOD Record, Vol. 28, No. 3, 1997, pp. 22-28.

[9] J. Boulos, Y. Viemont, and K. Ono: "A Neural Network approach for query cost evaluation," Transactions of the Information Processing Society of Japan, Vol. 38, No. 12, 1997, pp 2566 - 2575.

[10] Frederic Andres, Fred Kwakkel, and Martin L. Kersten, "Calibration of a DBMS Cost Model with the Software Testpilot," Proceedings of the International Conference on Information Systems and Management of Data, 1995, pp. 58-74.

[11] Nader B. Ebrahimi, "How to Improve the Calibration of Cost Models," IEEE Transactions on Software Engineering, Vol. 25, No. 1, 1999, pp. 136-140.

[12] Weiming Du, Ravi Krishnamurthy, and Ming-Chien Shan, "Query Optimization in Heterogeneous DBMS," Proceedings of the 18th International Conference on Very Large Data Bases, August 1992, pp. 277-291.

[13] Georges Gardarin, Fei Sha, and Zhao-Hui Tang, "Calibrating the Query Optimizer Cost Model of IRO-DB, an Object-Oriented Federated Database System," Proceedings of the VLDB Conference on Very Large Data Bases, 1996, pp. 378-389.

[14] Qiang Zhu and Per-Ake Larson, "A Query Sampling Method for Estimating Local Cost Parameters in a Multidatabase System," Proceedings of the 10th International Conference on Data Engineering, February 1994, pp. 144-153.

[15] Qiang Zhu and Per-Ake Larson, "Building Regression Cost Models for Multidatabase Systems," Proceedings of 4th IEEE International Conference on Parallel and Distributed Information Systems, December 1996, pp 220-231.

[16] Qiang Zhu, Yu Sun, and S. Motheramgari, "Developing Cost Models with Qualitative Variables for Dynamic Multidatabase Environments," Proceedings of the 16th International Conference on Data Engineering, March 2000, pp. 413-424.

[17] A.I. Khuri and J.A. Cornell, Response Surfaces: Designs and Analyses, Marcel Dekker, New York, 1996.

[18] Ju-Hong Lee, Deok-Hwan Kim, and Chin-Wan Chung, "Multi-Dimensional Selectivity Estimation Using Compressed Histogram Information," Proceedings of the ACM SIGMOD International Conference on Management of Data, 1999, pp. 205-214.

[19] M. Muralikrishna and David W. DeWitt, "Equi-Depth Histogram for Estimating Selectivity Factors for Multi-Dimensional Queries," Proceedings of the ACM SIGMOD International Conference on Management of Data, 1988, pp. 28-35.

[20] Viswanath Poosala, Yannis E. Ioannidis, Peter J. Haas, and Eugene J. Shekita, "Improved Histograms for Selectivity Estimation of Range Predicates," Proceedings of the ACM SIGMOD International Conference on Management of Data, 1996, pp. 294-305.

[21] Oracle8i Designing and Tuning for Performance - Release 2 (8.1.6), Oracle Documentation Library, 2000, Chapters 4, 6.

[22] Oracle8i interMedia Text Reference - Release 8.1.5, Oracle Documentation Library, February 1999, Chapter 4.

[23] Surajit Chaudhuri (ed.), Data Engineering Bulletin: Self-tuning Databases and Application Tuning, Vol. 22, No. 2, 1999.

Appendix

A Algorithm of NthGrpMavg

```
   // Simplified from the original PL/SQL code.
1  FUNCTION NthGrpMavg(groupsymbol: CHAR, startdate: DATE, enddate: DATE,
2                      windowsize: NUMBER, n: NUMBER)
3     temp: TABLE OF tsdev.tsquick_tab.close%type;
4     tempavg: TABLE OF tsdev.tsquick_tab.close%type;
5     j, k, tot: NUMBER;
6  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)
7     OPEN CURSOR c1 FOR
8        (SELECT b.tstamp, sum(b.close)/count(b.close) AS group_close
9           FROM tsdev.ticker_index a, tsdev.tsquick_tab b
10        WHERE a.ticker_index_id = groupsymbol
11          AND a.ticker = b.ticker
12          AND b.tstamp >= startdate - windowsize
13          AND b.tstamp <= enddate
14       GROUP BY b.tstamp);
   -- Cost ~ daterange+windowsize+1
15    LOOP UNTIL c1%NOTFOUND
16       fetch c1 into c1_rec;
17       temp(i) := c1_rec.group_close;
18       i := i + 1;
19    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
20    FOR j = 1 TO (temp.count - windowsize + 1)
21    BEGIN
22       tot := 0 ;
23       FOR k = j TO (j + windowsize - 1)
```

```
24          tot := tot + temp(k);
25       tempavg(j) := tot/windowsize;
26    END;

   // Mergesort tempavg and return the n-th element of the sorted tempavg.
   -- Cost ~ (daterange+2)log(daterange+2)
   -- Note: tempavg.count = daterange+2
27    Mergesort(tempavg, 1, tempavg.count);
28    RETURN(tempavg(n));
29 END NthGrpMvgAvg;
```