STATISTICAL COST-MODELING OF FINANCIAL TIME SERIES FUNCTIONS

Vinod Kannoth*, Byung Suk Lee**, Jeff Buzas***
*Federal Reserve Information Technology, 701 East Byrd Street, Richmond, VA 23219, USA
**Department of Computer Science, University of Vermont, Burlington, VT 05405, USA
***Department of Mathematics and Statistics, University of Vermont, Burlington, VT 05405, USA
E-mails: vinod.kannoth@frit.frb.org, byung.lee@uvm.edu, jeff.buzas@uvm.edu

Abstract
We present a *statistical regression* approach to building a cost model of an aggregate financial time series function. The cost model is needed by an object-relational DBMS query optimizer. This approach is much easier than the traditional analytical approach and yet achieves a highly precise model. Users only need to provide a set of variables influencing the costs. This requires only high-level understanding of how the function works. Experiments show that the cost models thus built are highly precise and that quadratic models are adequate.

Key Words
Regression, financial time series, query optimization

1 Introduction

Financial time series (FTS) functions, used extensively in stock market analyses, are supported by all major object-relational database management systems (ORDBMSs). There are different types of FTS functions, such as extraction, shift, aggregate, arithmetic, cumulative sequence, moving sequence, and conversion.

Among these, only *aggregate* FTSs, which return single values, are usable in the 'where' clause of an SQL query statement. (See an example in Fig. 1.) We thus focus on aggregate FTSs in this paper. An aggregate FTS function is used to perform an aggregation over a time series and return the aggregate value. Those typically supported by an ORDBMS are Avg, Count, Sum, Product, Variance, Min, MinN (mininum N elements), Max, MaxN (maximum N elements), and Median. If a date range (i.e., start date and end date) is given, then the aggregation is done only on the elements in the date range.

```
select c.name, c.stock_ticker
from Company c, Time_Series ts
where c.country = "USA"
and c.stock_ticker = ts.ticker
and MIN(ts.ticker, 01/01/2003, 12/31/2003) > 50
and MAX(ts.ticker, 01/01/2003, 12/31/2003) < 100
and MED(ts.ticker, 01/01/2003, 12/31/2003) > 75
and MED(ts.ticker, 01/01/2003, 12/31/2003) < 85
```

Figure 1. a query with aggregate FTS functions.

Extensible query optimizer of an ORDBMS needs the execution cost function of a user-defined function (UDF) specified in the 'where' clause of an SQL statement [1, 2]. (The extensible query optimizer will be described in Section 2.3.) A UDF is a database function that is *not* built in an ORDBMS. Thus, there is no cost function provided by the ORDBMS manufacturers. FTS functions, including aggregate FTS functions, are in this category.

Without a built-in cost function, it is left up to the application developers (who are users of the ORDBMS) to provide the cost functions of aggregate FTS functions used in their query statements. Traditionally, a database cost function is built as an analytical function of "data variables." These variables are derived from a data profile describing data configurations (e.g., cardinality, selectivity, blocking factor) and system configu-

rations (e.g., buffer size, disk page size). Typically, identifying and deriving these data variables require advanced knowledge of DBMS internal implementations [3, 4]. This is an overwhelming task for most users, and the resulting cost model suffers from inaccuracy. This point will be revisited in Section 2.2. In this paper, we propose an easier way – a *statistical regression* modelling approach. In this approach, users only need to provide the variables influencing the cost (termed *cost variables*) and a specification for generating a training data set from the observed execution costs. Cost variables can be determined in a straightforward manner based on users' understanding of the algorithm at a high level. The specification can be in the form of a grid plan based on the sampling range and sampling interval of each cost variable. Then, the system generates a training data set according to the user-provided specification and builds a cost function expressed as a regression model fitting the training data set.

We have observed that, for all the existing aggregate FTS functions, their execution costs are characterized by smooth, continuous, and monotonous variations. This makes sense because these functions perform computations on the ticker price while scanning time series elements in the data. Although the computation result may fluctuate as much as the ticker price does, the computation *time* (i.e., execution cost) is determined solely by the number of elements processed. Therefore, the cost is determined by the time interval over which the computation is performed.

Given cost variations with these characteristics, we show that a *quadratic* function can model cost precisely. The rationale behind this quadratic model is given further in Section 3 where details of the cost model building procedure are discussed. Experimental results show that our approach achieves quite accurate cost estimations.

Two main contributions are made through this paper. First, we propose an approach which enables users to build the cost functions of aggregate FTS functions without knowing the internals of an ORDBMS. Second, we demonstrate the merit of our approach through experiments, specifically, the accuracy of generated cost functions.

```
CREATE TYPE ORDSYS.ORDTCalendar AS OBJECT
(
      caltype INTEGER,
      name VARCHAR2(256),
      frequency INTEGER,
      pattern ORDSYS.ORDTPattern,
      minDate DATE,
      maxDate DATE,
      offExceptions ORDSYS.ORDTExceptions,
      onExceptions ORDSYS.ORDTExceptions
);
```

This 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).

Figure 2. calendar of financial time series data.

The rest of this paper is organized as follows. Section 2 provides some background information. Section 3 describes our approach to building cost models. Section 4 evaluates our approach. Section 5 discusses related work. Section 6 concludes.

2 Background

This section contains a brief overview of financial time series, cost estimate-based query optimization (with a focus on the analytical approach) and extensible query optimizer of an ORDBMS.

2.1 Financial time series

A *financial* time series (FTS) refers to a sequence of financial summary data – typically daily summary. A time series in general may be *regular* or *irregular* depending on whether it has an associated calendar. A regular time series data arrive predictably at intervals defined by a calendar. An example is a series of daily stock market summaries, such as trade volumes and the opening, high, low, and closing prices. Fig. 2 shows a calendar used in Oracle^{TM}ORDBMS.

An irregular time series has no associated

calendar, and data arrive unpredictably at unspecified time points. 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.

2.2 Query optimization using analytical cost modelling

A cost estimate-based query optimizer generates an optimal query execution plan (QEP) for a given query, based on the cost estimates of alternative QEPs [5, 6]. In the analytical approach, the cost estimate of a QEP is the sum of the costs of executing all the operators in the QEP. For this estimation purpose, each operator has an associated cost function.

The cost function is expressed as a formula of variables (or parameters) that must be instantiated to calculate the cost. There are typically three kinds of variables: hardware variables, data profile variables, and derived variables. Hardware variables are constant per database instance. Examples are the disk block size and the main memory buffer size. Data profile variables describe the statistics of the data stored. Examples are the number of rows in a table, the size of a row in a table, the number of distinct values of a column (or columns) of a table, and the height of an index on a table. Derived variables are calculated from hardware variables and data profile variables. Examples are the number of blocks in a disk-resident file, the number of records per disk block, the selectivity of a predicate on a column, and the selectivity of a join predicate.

As mentioned in Introduction, we collectively call these kinds of variables *data variables*. Typically, ORDBMS developers, who develop the routines executing individual operations, are responsible to determine the data variables and build the cost functions of these variables.

Evidently, this analytical process of generating a cost function is very complicated. Moreover, the cost functions thus generated are not always effective, and it becomes worse as the query complexity increases. Due to the insurmountable complexity involved in the task of developing cost functions analytically, developers often simplify the cost functions drastically. As a result, the generated cost functions suffer from inaccuracy.

The QEP selected by such a query optimizer is never optimal. It is even said by Krishnamurthy et al. [7] that for a traditional query optimizer "It is more important to avoid the worst execution than to obtain the best execution." The problem becomes worse when it comes to *UDFs*, which may include multiple queries to be optimized as an ordered set. Furthermore, even if the analytical approach were effective, deferring the overwhelming task to users is the last thing a market-savvy DBMS manufacturers should do. This is the problem our proposed approach is aiming to resolve.

2.3 Extensible query optimizer

An ORDBMS's extensible framework 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 changes of the system environment. Therefore, for instance, Oracle's extensible query optimizer uses the following metrics: for CPU cost, the number of machine instructions executed by the CPU, for disk I/O cost, the number of data pages transferred from disk to main memory buffer, and for 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.

An extensible query optimizer can handle UDF predicates (e.g., udf('01/01/1985', 50000, 'Sales') $>$ 60) in the same manner as it handles plain relational predicates (e.g., Employee.salary $>$ 50000). This extensible query optimizer needs users to provide the cost functions associated with each UDF predicate.[1] Then, provided with

[1] It also needs a selectivity function of a UDF predicate,

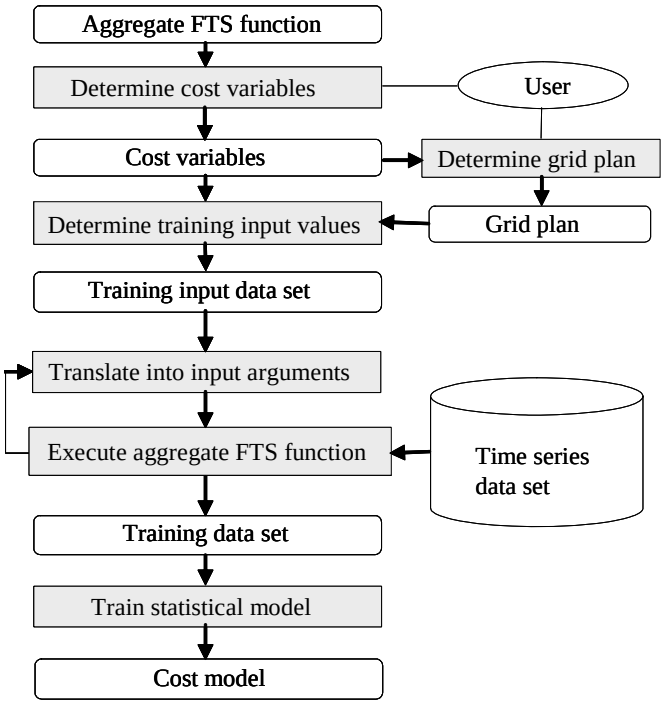


Figure 3. cost model building procedure.

cost functions, it orders the UDF predicates based on their estimated execution costs.

3 Model Building Steps

Fig. 3 shows the cost model building steps. In the first step, users determine *cost variables* and a *grid plan*. As will be demonstrated in Section 4.2, determining cost variables is a straightforward task in the case of aggregate FTS functions. A grid plan is defined as the sampling range and sampling interval for each cost variable. This grid sampling covers the model space (i.e., the Cartesian space defined by cost variables) evenly.

In the second step, a *training input data set* is generated by taking samples in the model space according to a grid plan. In the third step, a *training data set* is generated by executing the aggregate FTS function at each grid point. For this, first the aggregate FTS function's input argument values are generated from the training input data set. Each input argument is either a cost variable itself or transformed from one or more cost variables. For example, two cost variables startdate and enddate may be transformed into an input argument daterange as daterange = enddate - startdate + 1. The generated training data set is formed by combining the cost variables and the measured CPU and disk IO costs, that is, $\{\langle v_1, v_2, \cdots, v_d, CPU\_cost, disk\_IO\_cost\rangle\}$ where $v_1, v_2, \cdots, v_d$ are cost variables.

In the fourth step, the training data set is used to build a *cost model* through statistical training. Specifically, the system builds a full quadratic regression model by using the cost variables as the regression variables and fitting the model to the training data set. Quadratic regression models have shown to be precise enough for aggregate FTS functions in our evaluations (Section 4). The quadratic models we employ are equivalent to the second order Taylor approximation [8] to the true but unknown cost functions[2]. Second order approximations are used extensively in a wide variety of applications [9] because the approximation is quite good provided that 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, although it is possible to build separate models for individual values of a nominal variable if the cardinality is manageable. In our evaluations below, we consider only the ordinal case, by using *regular* time series data.

4 Evaluations

Evaluations are done for the cost estimation accuracies achieved using statistical cost models. In this section we present the experiment setup, regression model building, training and test data set generation, and then regression analyses.

4.1 Setup

4.1.1 Aggregate FTS functions

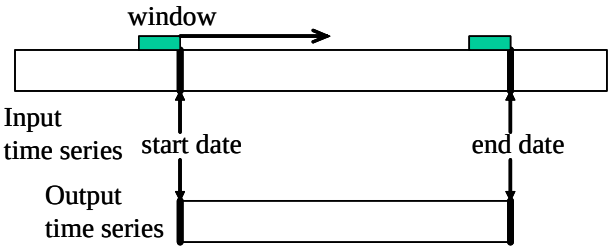
Two groups of aggregate FTS functions are used. The first group (Group 1) consists of Count,

but this is out of the scope of this paper.

[2] Considering a *single* cost variable x, the second order Taylor approximation of $f(x)$ is $f(0) + f'(0)\,x + f''(0)\,\frac{x^2}{2}$.

```
CREATE OR REPLACE FUNCTION FTSCount(
   tickersymbol: VARCHAR(10),
   startdate: DATE, enddate: DATE )
   numcnt NUMBER;
BEGIN
   SELECT COUNT(tstamp) into numcnt
   FROM tsdev.tsquick_tab
   WHERE ticker = tickersymbol
     AND tstamp >= startdate
     AND tstamp <= enddate;
   RETURN numcnt;
END FTSCount;
```

Figure 4. source code of FTSCount.



For a given stock ticker symbol, this function computes the average daily stock closing price within a sliding window between start date and end date (inclusive), and generates a time series of window average as a result.

Figure 5. moving average function.

Avg, and Min. These are basic ones commonly used in financial time series analysis. It is sufficient to study the three functions for Group 1, as cost curves for the other functions, such as sum, variance, product, and max are extremely similar. The second group (Group 2) consists of MinMavg, NthMavg, MinGrpMavg, and NthGrpMavg. These functions are our own extensions, and are studied here because they have more complex cost functions. All these FTS functions are implemented in Oracle PL/SQL.

Every aggregate FTSf in the group 1 has the following signature:

FTSf(tickersymbol, startdate, enddate)

where f is "Count", "Avg", or "Min". Fig. 4 shows the code of FTSCount. The codes of FTSAvg and FTSMin are the same except for the aggregate function in the SELECT clause. The codes are very simple, and so is the variation of the execution costs.

Group 2 functions are based on the moving average (Mavg) function described in Fig. 5. Their signatures are as follows.

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

Given a ticker symbol, NthMavg returns the n-th minimum of moving averages calculated within a specified date range, and MinMavg returns the first minimum. 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 time series of group average. MinGrpMavg returns the first minimum instead of the n-th minimum. Selecting the n-th minimum involves sorting, for which we use mergesort, whereas selecting the first minimum involves only linear scan and no sorting. Figure 7 in Section 4.2 shows the source code of NthGrpMavg.

We believe that Group 2 functions, which are based on the moving average function, are adequate enough to represent "complex" FTS aggregate functions. There may be other functions we may use, such as moving sum and cumulative average/sum/min/max. Moving sum is not different from moving average in terms of cost variables and the cost varations. Cumulative average, sum, min, and max are even simpler than moving average. They give only one the cost varible (i.e., date range) and their execution costs vary only linearly with the cost variable. In contrast, as will be seen in Section 4.2, Group 2 functions give two or three cost variables and their execution costs vary linearly or linear-logarithmically (i.e., $n \log n$).

4.1.2 Financial time series data

Fig. 6 shows the schema of financial ticker time series data used in the experiment. The schema `tsdev` contains two tables. The ta-

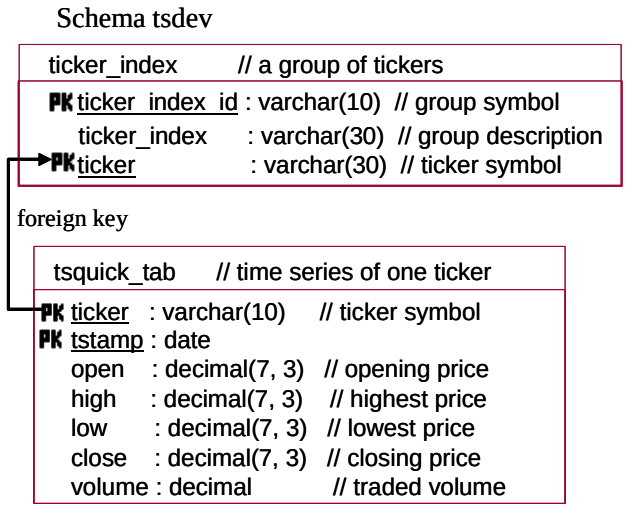


Figure 6. financial time series data schema.

ble `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 the primary key fields.

The date range of time series is 29220 days (or 80 years) for each ticker. There are four groups in the data, with the group size of 5, 10, 15, and 20.

4.2 Regression model building

In order to demonstrate the adequacy of quadratic models, we compare the precision of quadratic models (Quad) with the precision of models obtained through run-time analysis (RTA) of aggregate FTS functions. Table 1 summarizes the cost variables and the regression models of the aggregate FTS functions used in the experiments. Since we consider *regular* time series data, ticker symbol and start date have no effect on the costs and, therefore, are not cost variables.

From the semantics of the group 1 functions (see Fig. 4), it is obvious that their cost variations are linear with the date range as the only cost variable. Hence, the RTA model is a linear regression as shown in Table 1(a). In contrast, we need two to three cost variables for the group 2 functions, as shown in Tables 1(b) and (c). The cost variables and the RTA models can be determined straightforwardly through simple analysis of the codes. Fig. 7 shows the source code of NthGrpMavg. We omit the other source codes due to space limit.

Table 1. Regression Models.

Cost variables: daterange (D), windowsize (W), groupsize (G)

Quad	$c_0 + c_1 D + c_2 D^2$
RTA: Count, Avg, Min	$c_0 + c_1 D$

(a) Group 1: Count, Avg, Min.

Quad	$c_0 + c_1 D + c_2 W + c_3 D^2 + c_4 W^2 + c_5 DW$
RTA: Min	$c_0 + c_1 D + c_2 DW$
RTA: Nth	$c_0 + c_1 D + c_2 (D - W + 1) +$ $c_3 (D - W + 1) \log_2 (D - W + 1)$

(b) Group 2: MinMavg, NthMavg.

Quad	$c_0 + c_1 D + c_2 W + c_3 G + c_4 D^2 + c_5 W^2 +$ $c_6 G^2 + c_7 DW + c_8 WG + c_9 GD$
RTA: Min	$c_0 + c_1 D + c_2 DW + c_3 WG + c_4 GD$
RTA: Nth	$c_0 + c_1 G(D + W + 1)D + c_2 (D + W + 1) +$ $c_3 (D + 2)W + c_4 (D + 2) \log_2 (D + 2)$

(c) Group 2: MinGrpMavg, NthGrpMavg.

Let us discuss the regression model for NthGrpMavg as an example. Fig. 7 shows the fragments of the code and the cost formula for each fragment. In the first fragment (Lines 10∼17), it looks for data records of all ticker symbols belonging to a group named `groupsymbol`. Then, while scanning the records from `startdate - windowsize` to `enddate`, it groups the scanned records by their date (i.e., `tstamp`) and calculates the average of all ticker closing prices (i.e., `close`) in each group. In the second fragment (Lines 18∼22), it fetches the daily group averages one by one and saves them into a temporary array `temp`. In the third fragment (Lines 23∼29), 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`. In the fourth fragment (Lines 30∼31), it sorts `tempavg` using mergesort and returns the `n`-th element. Since mergesort takes $O(N \log_2 N)$ for a file of 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`. Thus, the RTA model of NthGrpAvg is expressed a summa-

```
1  CREATE OR REPLACE FUNCTION NthGrpMavg
2    (groupsymbol: VARCHAR(10), startdate: DATE,
3     enddate: DATE, windowsize: NUMBER, n: NUMBER)
4  RETURN NUMBER
5  IS
6    temp: TABLE OF tsdev.tsquick_tab.close%type;
7    tempavg: TABLE OF tsdev.tsquick_tab.close%type;
8    j, k, tot: NUMBER;
9  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)
10   OPEN CURSOR c1 FOR
11     (SELECT b.tstamp, SUM(b.close)/COUNT(b.close)
                         AS group_close
12     FROM tsdev.ticker_index a,tsdev.tsquick_tab b
13     WHERE a.ticker_index_id = groupsymbol
14       AND a.ticker = b.ticker
15       AND b.tstamp >= startdate - windowsize
16       AND b.tstamp <= enddate
17     GROUP BY b.tstamp);
   -- Store the tuples fetched using cursor c1 into
   -- temp.
   -- Cost ~ daterange + windowsize + 1
18   LOOP UNTIL c1%NOTFOUND
19     fetch c1 into c1_rec;
20     temp(i) := c1_rec.group_close;
21     i := i + 1;
22   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
23   FOR j = 1 TO (temp.count - windowsize + 1)
24   BEGIN
25     tot := 0 ;
26     FOR k = j TO (j + windowsize - 1)
27        tot := tot + temp(k);
28     tempavg(j) := tot/windowsize;
29   END;

   -- Mergesort tempavg.
   -- Cost ~ (daterange + 2) * log(daterange + 2)
   -- Note: tempavg.count = daterange + 2
30   MergeSort(tempavg, 1, tempavg.count);

   -- Return the n-th element of the sorted tempavg.
31    RETURN(tempavg(n));
32 END NthGrpMvgAvg;
```

Figure 7. source code of NthGrpMavg.

tion of the four cost formulas, one for each code fragment.

4.3 Training and test data sets

Table 2 shows the grid plans used in the experiments. A training data set is generated by executing an aggregate FTS function at grid sample points. Each execution is repeated four times and averaged to suppress noise.

A test data set is generated by taking *random* sample points. (Thus, the sampling intervals are irrelevant for a test data set.) This mimics the cost variable values as they appear in actual UDF execution patterns.

During the data set generations, we clear the database buffer between each consecutive runs to eliminate the unpredictable caching effect. (Most DBMS query optimizers disregard the caching effect or use an overly simple model [10].) This is done by executing an I/O-intensive dummy routine.

We use *CPU time* as a generic CPU cost metric instead of the number of machine instructions (see Section 2.3). CPU time can be converted to the number of machine instructions as CPU time $\times$ CPU speed / the average cycles per instruction (CPI). We use an arbitrary number 1.5 as the average CPI; the actual number is immaterial in our experiments.

All the data sets are generated on Sun Ultra Enterprise 450 with four 276 MHz CPUs, 1024 Mbyte RAM, and 55 Gbyte hard disk.

4.4 Regression analyses

Table 3 summarizes the cost estimation errors resulting from the regression analyses done for each aggregate FTS function. In a majority of cases, the median relative errors (DRE) are lower than 5% and the mean relative errors (MRE) are lower than 10%. DREs are more credible than MREs in the table. For instance, the MRE 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, respectively.

Table 2. Grid Plans.

D	Aggregate FTS functions	Cost variables	Sampling range	Sampling interval†	Data set size‡
1	Group 1: Count, Avg, Min	daterange	0 ∼ 80 years	10 years	36 points
2	Group 2: MinMavg, NthMavg	daterange	1 ∼ 80 years	10 years	
		windowsize	1 ∼ 60 days	15 days	180 points
3	Group 2: MinGrpMavg, NthGrpMavg	daterange	1 ∼ 80 years	10 years	
		windowsize	1 ∼ 60 days	15 days	
		groupsize	5 ∼ 20 tickers	5 tickers	720 points

(D: dimensionality. †: not applicable to test data set generation. ‡: = 4 × the number of sample points.)

Table 3. Cost Estimation Error Statistics of Experimental Aggregate FTS Functions.

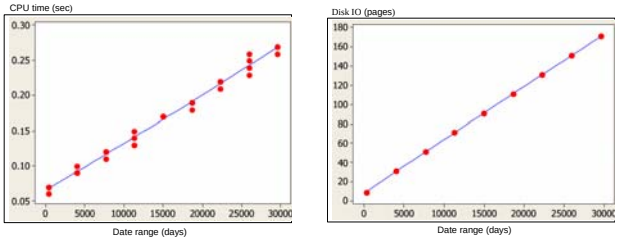
Aggregate FTS function	CPU cost (seconds)						Disk I/O cost (pages)					
	Quad model			RTA model			Quad model			RTA model		
	MAE	MRE	DRE	MAE	MRE	DRE	MAE	MRE	DRE	MAE	MRE	DRE
Count	0.02	13.85	11.37	0.02	14.14	12.23	0.52	0.62	0.35	0.52	0.62	0.35
Avg	0.01	6.28	5.63	0.01	6.17	5.36	0.52	0.62	0.35	0.52	0.62	0.35
Min	0.02	8.44	7.18	0.02	8.85	7.56	0.52	0.62	0.35	0.52	0.62	0.35
MinMavg	0.47	4.95	1.69	0.45	3.59	1.85	30.38	67.2	21.9	27.4	53.3	22.8
MinGrpMavg	4.18	2.64	1.56	4.19	3.28	1.66	192.98	1.90	1.51	194.88	2.16	1.49
NthMavg	1.14	1.47	0.55	7.98	73.3	3.83	16.09	13.6	4.18	12.88	17.5	2.49
NthGrpMavg	2.71	1.41	0.76	11.23	10.0	2.41	226.72	0.98	0.71	323.68	1.44	1.29

MAE = mean absolute error, MRE = mean relative error (%), DRE = median relative error (%)

As we can see in Table 3, the cost estimation errors of Quad models are very close to those of RTA models, and even lower in many cases. The reason is that both the RTA and Quad models approximate the true cost function, but the Quad model is more flexible. Indeed we see that Quad models are precise enough to substitute all the RTA models shown in Table 1.

Note that the disk I/O costs of Count, Avg, and Min are exactly the same. The reason is that the three FTS functions access exactly the same set of disk pages through an index scan on the column `tstamp` of the table `tsquic_tab`. The CPU costs are slightly different because different aggregationt functions involve different computations.

Fig. 8 shows plots of the CPU and disk I/O costs of Avg, specifically the observed costs (shown as dots) and the fitted lines of estimated costs. Fig. 9 shows plots of the CPU and disk I/O costs of NthGrpMavg, specifically the observed costs (shown in vertical bars) and the fitted surfaces of estimated costs (shown in dotted membranes). The fittings have been done using



(a) CPU cost. (b) Disk I/O cost.

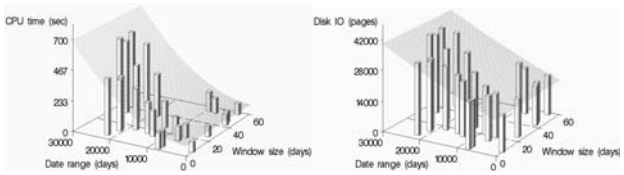
Figure 8. observed costs and fitted lines of Avg.

the quadratic models. We can visually confirm that the estimated costs match the observed costs quite accurately.

5 Related Work

We find related work in terms of using regression for generating a cost function. Andres et al. [11] model DBMS performance as a regression equation and tune its coefficients by running a set of representative workload, and Ebrahimi [12] 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



(a) CPU cost. (b) Disk I/O cost.

Figure 9. observed costs and fitted surfaces of NthGrpMavg (group size = 20).

operations for use by a global query optimizer in a multidatabase environment: model calibration [13, 14] and query sampling [15, 16]. In the model calibration method, Du et al. in [13] 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 [14] extend Du et al.'s work to an object-oriented query optimization.

In the query sampling method [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 that are randomly selected from the class. Unlike model calibration [13, 14], this method uses the entire real data actually used in a local DBMS.

Both model calibration 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 Conclusion

In this paper, we have presented a novel approach to generating the cost functions of aggregate financial time series (FTS) functions. Unlike the traditional analytical approach, the proposed approach uses statistical regression. Users only need to provide cost variables and a grid plan for sampling the cost variables. Then, the system generates a training data set through executing the aggregate FTS function according to the grid plan and builds a cost function by fitting a quadratic regression model (of the cost variables) to the training data set.

We have demonstrated the accuracy of cost estimations through experiments. The results obtained from using two groups of aggregate FTS functions show that our approach accomplishes accurate cost estimations – lower than 5% median relative errors and 10% mean relative errors in a majority of cases – when measured against regular time series data. This accuracy is superior to what is achievable from using the traditional analytical approach.

Acknowledgments

We thank the anonymous referees for their comments on the original manuscript. This research has been supported through US DOE Grant DE-FG02-ER45962 and NSF Grant IIS-0415023.

References

[1] S. Chaudhuri & K. Shim, Optimization of queries with user-defined predicates, *ACM Trans. on Database Systems*, 24(2), 1999, 177-228.

[2] J. Hellerstein, Optimization techniques for queries with expensive methods, *ACM Trans. on Database Systems*, 23(2), 1998, 113-157.

[3] S. Chaudhuri, An overview of query optimization in relational systems, Proc. the ACM Conf. on Principles of Database Systems, Seattle, WA. 1998, 34-43.

[4] Y.E. Ioannidis, Query optimization, in A. Tucker (Ed.), *The Computer Science and*

Engineering Handbook, (CRC Press, 1997) 1038-1057.

[5] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, & T. Price, Access path selection in a relational database management system, Proc. ACM SIGMOD Conf. on Management of Data, Boston, MA, 1979, 23-34.

[6] M. Roth, F. Ozcan, & L. Haas, Cost models do matter: providing cost information for diverse data sources in a federated system, Proc. Conf. on Very Large Data Bases, Edinburgh, Scotland, 1999, 599-610.

[7] R. Krushnamurthy, H. Boral, & C. Zaniolo, Optimization of non-recursive queries, Proc. Conf. on Very Larege Data Bases, Bombay, India, 1986, 128-137.

[8] S. Lang, *Calculus of several variables* (Addison-Wesley Publications, 1979).

[9] A.I. Khuri & J.A. Cornell, *Response surfaces: designs and analyses* (Marcel Dekker, NY, 1996).

[10] P. Gassner, G. Lohman, K. Schiefer, & Y. Wang, Query optimization in the IBM DB2 family, *Data Engineering Bulletin*, 16(4), 1993, 4-18.

[11] F. Andres, F. Kwakkel, & M. Kersten, Calibration of a DBMS Cost Model with the Software Testpilot, *Proc. Conf. on Information Systems and Management of Data*, Bombay, India, 1995, 58-74.

[12] N. Ebrahimi, How to Improve the Calibration of Cost Models, *IEEE Trans. on Software Engineering*, 25(1), 1999, 136-140.

[13] W. Du, R. Krishnamurthy, & M.-C. Shan, Query Optimization in Heterogeneous DBMS, *Proc. Conf. on Very Large Data Bases*, Vancouver, Canada, 1992, 277-291.

[14] G. Gardarin, F. Sha, & Z.-H. Tang, Calibrating the Query Optimizer Cost Model of IRO-DB, an Object-Oriented Federated Database System, *Proc. Conf. on Very Large Data Bases*, Bombay, India, 1996, 378-389.

[15] Q. Zhu & P.-A. Larson, Building Regression Cost Models for Multidatabase Systems, *Proc. IEEE Conf. on Parallel and Distributed Information Systems*, Miami Beach, FL, 1996, 220-231.

[16] Q. Zhu, Y. Sun, & S. Motheramgari, Developing Cost Models with Qualitative Variables for Dynamic Multidatabase Environments, *Proc. IEEE Conf. on Data Engineering*, San Diego, CA, 2000, 413-424.

Vinod Kannoth is Database Analyst at Federal Reserve Information Technology. His areas of interest include database systems, data warehousing, and content management. He holds an MS degree in Computer Science from the University of Vermont.

Byung Suk Lee is Associate Professor of Computer Science at the University of Vermont. His areas of interest include databases, data streams, and sensor networks. He holds a Ph.D. degree in Electrical Engineering (Database Systems) from Stanford University.

Jeff Buzas is Associate Professor of Statistics at the University of Vermont. His areas of interest include measurement error models, bioassay, and instrumental variables. He holds a Ph.D. degree in Statistics from North Carolina State University.