Tuning Time Series Queries in Finance: case studies and recommendations

Dennis Shasha
Courant Institute of Mathematical Sciences
Department of Computer Science
New York University
shasha@cs.nyu.edu
http://cs.nyu.edu/cs/faculty/shasha/index.html

Abstract

Financial databases are different from regular ones: they involve more money. Furthermore,

- *Database speed and reliability can make the difference between prosperity and ruin.*
- *Money for information systems (and for salaries) is no object.*
- *Data must be accessible from many points on the globe with subsecond response.*
- *Many calculations require computations on vectors, usually time series.*

In a previous tutorial article [10], I presented several case studies having to do with reliability, interoperability, data coherency, and tuning. In this complementary paper, I discuss examples of challenges that follow from the time series nature of financial data. I also suggest a way to attack this problem.

1 Cultural Context

As a frequent consultant to Wall Street, I am exposed to a technical atmosphere that is unrecognizable to most researchers and system programmers: time is short, tempers are shorter, and bonuses influence truth.

- New financial products are introduced within days of their conception.
- Traders routinely yell obscenities into their phones. Some break those same phones by banging the handset against its base in fury.[1]
- Technical people often bear the brunt of this wrath and usually must do so with good humor since their yearly bonuses depend largely on what the traders say about them.

Copyright 1999 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.
Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[1] This behavior goes home with some. The director of technology of a major investment bank was once handcuffed to a fence by the police so he would stop harassing the umpire during his son's little league game.

In such an environment, planning horizons rarely exceed six months and technical purchase decisions often depend on what some trader used at his or (rarely) her last job. The result is a babel of systems that are held together by the sweat of highly paid programmers and administrators.

Like the traders, these technical workers change jobs frequently (once every two years or so) and so feel a need to tell a good story about the latest technology in the press. One group leader told me in 1994 that he used C++ in his group, partly because he knew it would attract people who wanted to put knowledge of that language on their resume *for their next job*. Replace C++ by Java Beans and you will find a similar story as of 1999.

Fortunately for the industry, the technical people are quite smart. So, while they follow fads, the fads usually answer a real need. Sybase enjoyed a huge success on Wall Street in the late 1980s, for example, because the system had light weight threads and triggers. I still use the word 'fad' however because technology selection is not done in a vacuum on Wall Street. Everyone knows what everybody else is doing. The more popular a technology becomes, the more people want to use it because it will give them job mobility. Conversely, vastly superior technology may have a hard time capturing interest until, well, it catches on. I will discuss a needed superior technology towards the end of this article.

2 Four Ways of Working with Time

Financial applications treat time in many ways, but the following four queries illustrate the main variants.

- Compute the value of my portfolio of bonds based on future payments and projected interest rates.
- Find correlations or other statistics among different stock prices over time or between option premiums and stock prices in order to help make better trading decisions.
- Find all events that are likely to precede a rise in the price of a security, regardless of precise timing constraints.
- Determine the data available at time X to trader T when he made the decision to buy Y for his own account.

Let us discuss each variant and its generalizations in turn.

3 Present Value

Contrary to what one reads in the popular press, bonds are far more important than stocks for most investment banks. Companies, municipalities, and nations issue debt and the markets buy them. Each debt note has an associated payment stream occurring at regular intervals, e.g. each half-year. These debt notes are then swapped for cash (I'll borrow money and give some bonds I'm holding as collateral), tranched (sliced into payment streams of different risks), and otherwise manipulated. The basic computational issue is to value these payment streams.

But how should one value ten yearly one million dollar payments? The basic principle is to normalize all payments to their *present value*. The present value of a payment of x dollars in i years assuming an interest rate of r is $y = (x/((1+r)^i))$ dollars. (This definition is justified by the fact that putting y dollars in the bank at interest r would yield the same amount of money at the maturity of the bond as taking each of the payments and putting them in the bank at interest r.)

So, the basic problem is to compute expressions of the form $(1+r)^i$ for many different values of i. Since the value of r is never known precisely, many different r's are tried in order to calculate the statistics of the value of a bond.

Notice that each such calculation involves computing the log of $1+r$, then multiplying by i and then taking the inverse log of the result. Since many expressions have the same value of r but differ on i, we'd like to avoid doing the log more than once for a given r.

Relational systems on Wall Street typically store each payment in the form: (amount, date, interest). The present value of each row is then computed independently of all others either through a trigger or an object-relational extension. This strategy misses the significant log optimization described above and the resulting performance is punishingly slow. For this reason, many applications take the data out of the relational database and perform this calculation in some other way.

Avoiding this exodous from the database requires the object-relational system to share computational work even among external functions. That is an undecidable task in general since the external functions are written in a Turing-complete language. Another possibility is to call external functions on bigger entities (e.g. arrays of payments and dates) so the function itself can do the necessary computational sharing.[2] We will explore this second possibility later.

3.1 Regular Time Series and Statistics

Trends are a popular abstraction for many average stock investors. Knowing that "the market is going up" is a powerful motivation to buy and conversely. Unfortunately, history and countless investment books suggest that this does not yield good results for two reasons. First, it is not clear when a trend has started. Second, trend trading ignores the different properties of different companies.

A far better strategy is to make use of correlations between different companies having similar properties. Take two similar banks B1 and B2. In the absence of differentiating external news (e.g. a default of a creditee of B1 alone), the prices of the two banks should track one another closely. So, if B1 starts going up while B2 does not, a "pairs trader" will sell B1 and buy B2.

In the FinTime benchmark[3] Kaippallimalil J. Jacob of Morgan Stanley and I have tried to capture the typical queries correlation and regression analysis requires.

The model consists of a few relational tables that typically contain infrequently changing information about stocks and a number of time-series tables about prices and stock splits.

Here are three typical queries:

1. Get the closing price of a set of 10 stocks for a 10-year period and group into weekly, monthly and yearly aggregates. For each aggregate period determine the low, high and average closing price value.

2. Find the 21-day and 5-day moving average price for a specified list of 1000 stocks during a 6-month period. (Use split adjusted prices).

3. Find the pair-wise coefficients of correlation in a set of 10 securities for a 2 year period. Sort the securities by the coefficient of correlation, indicating the pair of securities corresponding to that row.

Here are some features that relational systems require to answer such queries:

1. the special treatment of the semantics of time (e.g. the discovery of weekly boundaries and the ability to aggregate over such boundaries)

2. the ability to treat events in a time-ordered sequence (e.g. to perform moving averages)

3. the ability to perform time-based statistical operations on multiple sequences.

[2] It should go without saying that the overhead of each call should be low. Unfortunately, null external function calls appear to require thousands of instructions in some large well-known database management systems.

4 Irregular Time Series and Frequency Counting

Hurricanes, oil spills, earthquakes, and other natural and manmade disasters can influence the stock market in many ways. These events are not regular (one hopes) and so moving averages and complex signal processing techniques are inapplicable. On the other hand, the fusion of disaster events with price data may make the following queries meaningful and useful:

- (Specific) Which disaster event type affects the price of insurance company X?
- (General) Which event sequences affect which companies within 5 days?

We have used the example of disasters, but similar queries apply to large purchases prior to merger announcements and other suspected insider activity.[3] One way to conceptualize such problems is to discover patterns in timed event sequences[7]. Relational systems would model this by attaching a time attribute to events, but the discovery of sequence of events would require one or more joins. Most real-life systems would take some data from the database and do the real work outside.

5 Bitemporality

It is often necessary to know what someone knew at a certain date. For example, did trader Jeff believe that a merger was to take place between companies X and Y as of time Z when he bought stock in X? This might be grounds for legal concern. What was the best public guess as to the third quarter profits of company X at time Z? This might explain why trader Carol took such a large position in the company at that time.

Such queries are called bitemporal[4, 11, 12] because they involve two times: the time of the event in question (the merger or the third quarter profits in our examples) and the time of the query (time Z in both cases). The point is that our knowledge of the event in question may be better now, but what we are interested in is what was known at some time in the past based on the transactions that had committed up to that point.

Whereas most queries having to do with compliance or personnel review occur rarely enough to be irrelevant for performance tuning, the same does not hold for other queries involving time. For example, consider a portfolio accounting application with queries like this:

- As of last June, what was the anticipated revenue stream of this portfolio over the next ten years.

Since a portfolio consists of many securities, this involves combining the streams from many different securities after determining which securities were in the database last June.

As Snodgrass and Jensen show in their book[12], such queries can be implemented in relational databases, but may require pages of code and may in fact be very slow. Indeed, the bank officer who described the portfolio accounting application put the problem this way: "The functionality excites the users but makes the [standard relational] server glow a dull cherry red..."

At least two solutions are possible to speed up such actions:

- Implement the bitemporal model inside the database system. The risk is that this may create overly specialized implementations.
- Treat the data as ordered and see where that leads. Please read on.

[3] Of course, such event sequence queries appear elsewhere, e.g. in web data mining, A typical problem there is to infer the probability that a person will buy an item given that he has visited some sequence of pages. Sequence plays a role but not necessarily wall clock time.

6 Recommendations

My recommendations follow from a simple observation: *Sequences are more powerful than sets.*

Membership, intersection, union, and difference can be easily applied to sequences. On the other hand, order, nth best, moving averages, and other statistics require a tortured and inefficient encoding on sets, but are linear time on sequences.

Now let us consider two ways in which sequences might extend the relational model:

- Tables might be maintained in a specific order, as **arrables** (array tables). Selections, joins, and projections can all carry their multiset-theoretic meaning to arrables (SQL is a multiset language when all is said and done), but new operations can be defined that exploit order. These new operations may include nth best operations, moving averages on certain columns, correlations between columns and so on.

  Ordering tables into arrables has certain disadvantages however. For example, an arrable cannot be stored using hashing. Ordering does not however preclude a secondary hash index. The extra overhead for ordering is therefore one more access in most cases[9]. This extra access may be significant in disk-resident data.

- Another approach to order is to break a relational commandment. In a normal SQL statement group by statement, e.g.

  ```
  select id, avg(price)
  from stock
  group by id
  ```

  each group corresponds to one id and there are one or more scalar aggregates associated with that group (here, the average price in the group). Suppose instead that that the following were possible:

  ```
  select id, price, date
  from stock
  group by id
  ```

  The net result here would be to cause a *vector* of prices and dates to be associated with each stock. Now that we have vectors, we can apply vector style operations that may be offered by a system or that an application program may write. For example, we may sort the prices in each group by date and then construct moving averages. Regrettably, allowing vector elements in rows violates first normal form. Maybe this is ok.

These two ways of incorporating order are separable. If introduced together, then the stock arrable might be ordered by date and then the query becomes:

```
select id, price
from trade
group by id
```

Prices will then be ordered by date within each date, provided the grouping preserves the sort order of the trade arrable. Even better, we can construct the 5 day moving average in one shot, given a function that computes the moving average of a sequence, say "movavg" and takes a time parameter:

```
select id, movavg[5,price]
from trade
group by id
```

Several systems treat sequences as first class objects including Fame[1], SAS[8], S-Plus[6], and KDB[2]. KDB supports a language called KSQL which is a semantic extension to SQL supporting order and time[4] in a few ways:[5]

- Tables are ordered, often but not necessarily by time.
- Special functions are offered to exploit this order, e.g. first, last, nth, and moving aggregate statistics. For example, if the trade table is ordered in descending order by price, then the following produces the 17th highest price of each stock

```
select last 17 first price
 from trade
 group by stock
```

- Other functions can be added that apply to scalars, vectors or entire tables by the user, just as in conventional object-relational systems. For example, suppose we want to compute the dot product of prices per stock offset by 10 days. The dot product function is first defined (in this case, in the language K) and then used

```
dotprod:{[x;y] +/(x*y)}
select dotprod[(10 drop price), (-10 drop price)]
 from trade
 group by stock
```

- Date is a data type and the system is aware of operations on that type. In this example, the trade table is grouped by date and month.

```
select volume, price
 from trade
 group by date.month
```

Looking back on the time series queries we face in finance, let us see what order does for us:

- Computing the present value and similar operations are natural on sequences. If the external functions to compute these functions are vector-aware, then optimizations, such as performing the logarithm once instead of many times, can easily be incorporated.
- Correlations, moving averages, grouping by time periods such as weeks and months, and special purpose interpolations require an awareness of date as a special data type and the ability to add arbitrary functions in an object-relational style. Event data mining requires a subset of this functionality.
- Bitemporality remains a challenge, but less of one. Because bitemporal queries involve unchanging historical data, one can afford to store redundant data, e.g. the entire set of portfolio entries each time a portfolio changes. So, the query that makes a current server glow a cherry red, can perhaps become red hot.

7 Conclusion

Time as used in finance presents a performance challenge to conventional databases. After reviewing the main query variants, this paper recommends a simple but radical change: embrace order.

[4] KSQL also shortens some SQL statements, e.g. "group by" becomes "by" and some joins are implicit. I have rendered the examples without those shortcuts here.

[5] SRQL shares many of these ideas and adds a few other twists[5]

8 Acknowledgments

Many thanks to Marc Donner for introducing me to financial databases and to Arthur Whitney for profound scientific insight.

References

[1] Information about Fame can be found at www.fame.com.

[2] KDB and KSQL can be downloaded on a trial basis from www.kx.com.

[3] K. Jacob and D. Shasha "FinTime, a financial time series benchmark" http://cs.nyu.edu/cs/faculty/shasha/fintime.html

[4] Christian S. Jensen and Richard T. Snodgrass "Semantics of Time-Varying Information *Information Systems, 21(4)* pp. 311-352 (1996)

[5] Raghu Ramakrishnan, Donko Donjerkovic, Arvind Ranganathan, Keven S. Beyer, and Muralidhar Krishnaprasad, "SRQL: sorted relational query language" *SSDBM 98*

[6] B.D. Ripley and W.N. Venables, *Modern Applied Statistics with S-Plus* Springer Verlag (1994) http://www.stats.ox.ac.uk/ ripley/ has a lot of useful functions.

[7] John F. Roddick and Myra Spiliopoulou. "A bibliography of temporal, spatial and spatio-temporal data mining research." SIGKDD Newsletter, 1999.

[8] Information about SAS (the statistical package) can be found at www.sas.com

[9] Dennis Shasha *Database Tuning — a principled approach*, Prentice-Hall, Englewood Cliffs, NJ, 1992.

[10] Dennis Shasha "Lessons from Wall Street: case studies in database tuning, configuration, and replication" *Proc. 1997 ACM Sigmod Conf.*, pp. 498-501. Slides are available from the research link of Shasha's web page.

[11] Richard T. Snodgrass (editor), Ilsoo Ahn, Gad Ariav, Don Batory, James Clifford, Curtis E. Dyreson, Ramez Elmasri, Fabio Grandi, Christian S. Jensen, Wolfgang Kifer, Nick Kline, Krishna Kulkarni, T. Y. Cliff Leung, Nikos Lorentzos, John F. Roddick, Arie Segev, Michael D. Soo and Suryanarayana M. Sripada, *The TSQL2 Temporal Query Language*, Kluwer Academic Publishers, 1995.

[12] Richard T. Snodgrass and Christian S. Jensen *Developing Time-Oriented Database Applications* Morgan-Kaufman, 1999.