Fast Algorithms for Time Series with applications to Finance, Physics, Music, Biology, and other Suspects [*]

Alberto Lerner Dennis Shasha Zhihua Wang Xiaojian Zhao Yunyue Zhu
Courant Institute of Mathematical Sciences
New York University
New York, NY 10012
{lerner,shasha,zhihua,xiaojian,yunyue}@cs.nyu.edu

ABSTRACT

Financial time series streams are watched closely by millions of traders. What exactly do they look for and how can we help them do it faster? Physicists study the time series emerging from their sensors. The same question holds for them. Musicians produce time series. Consumers may want to compare them. This tutorial presents techniques and case studies for four problems:

1. Finding sliding window correlations in financial, physical, and other applications.
2. Discovering bursts in large sensor data of gamma rays.
3. Matching hums to recorded music, even when people don't hum well.
4. Maintaining and manipulating time-ordered data in a database setting.

This tutorial draws mostly from the book *High Performance Discovery in Time Series: techniques and case studies*, Springer-Verlag 2004. You can find the power point slides for this tutorial at
http://cs.nyu.edu/cs/faculty/shasha/papers/sigmod04.ppt.

The tutorial is aimed at researchers in streams, data mining, and scientific computing. Its applications should interest anyone who works with scientists or financial "quants." The emphasis will be on recent results and open problems. This is a ripe area for further advance.

1. SLIDING CORRELATION DISCOVERY

1.1 Motivation from Finance

Millions of people watch the price time series of the stock market every day. Some believe in trends and momentum: if the market is going up, it will keep going up. History has not been kind to such people – every burst bubble leaves many of them bankrupt. The basic problem is that the price movement of day d is a poor predictor of the price movement on day $d + 1$.

One problem the trenders face is that stocks may go up or down in synchrony with external factors such as oil price rises, elections, interest rate changes and so on. So many traders look for a way to make money by "hedging." For example, if $s1$ and $s2$ are closely related stocks and you buy $s1$ and sell the $s2$ for the same Euro amount, external factors will have little influence left. The question then is which to buy and which to sell and when.

Correlation traders use the following reasoning. The *day return* of a stock s from day d to $d{+}1$ is $\big(s(d{+}1)-s(d)\big)/(s(d))$. This corresponds to the percentage gain or loss on s from day d to $d + 1$. Now, if the returns of $s1$ and $s2$ have been correlated very closely, but suddenly the return of $s1$ is much lower than the return of $s2$, then a correlation trader will buy $s1$ and sell $s2$ in the belief that the two stocks will return to correlation. As opposed to the momentum strategy, this is a convergence strategy. Traders made a large profits based on this idea in the 1980s. But like all stock market ideas, once several people jump on it, profits disappear. So, now the frontier is intra-day correlations, where both the quantity of data and the response requirements are greatly accelerated.

1.2 Algorithmic Techniques for Sliding Correlation

A line is the shortest route between two points but the easiest path wanders – Kentucky Pete, rock climber.

Computationally therefore the problem is to find pairs of stocks whose returns are highly correlated for a certain window of time and then flag them when they fall out of correlation. The first part is the algorithmic challenge. When there are thousands of time series (as in finance) or millions (as in satellite data), finding all pairwise correlations takes $O(w\ n^2)$ time where n is the number of time series and w is the length of the windows. Worse still, the sliding window requirement implies that one must compute these correlations at every time point. This becomes prohibitively very quickly.

Fortunately, a synergy of three techniques accelerates the solution greatly:

1. data reduction techniques to create low dimensional synopses via Fourier transforms, Wavelet transforms,

[*]Work supported in part by U.S. NSF grants IIS-9988636 and N2010-0115586.

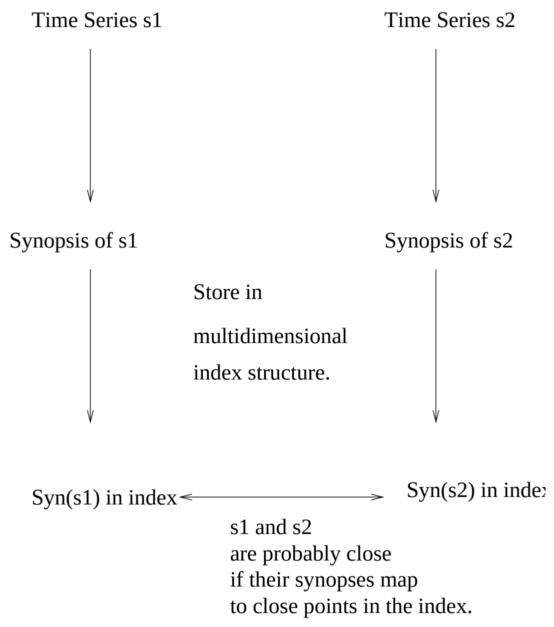


Figure 1: GEMINI framework: map each time series to a lower dimension and then find similar ones by looking them up in a multidimensional index structure

singular value decomposition, and random projection

2. multidimensional indexing techniques on the low dimensional synopses to find highly correlated pairs
3. fast updates to the synopses for moving correlation

Professor Christos Faloutsos will discuss the first two in depth in his tutorial, which is entirely just since he helped invent many of those techniques in the GEMINI framework[1]. But a figure 1 will help explain those first two steps. What is remarkable is that these two steps make the distance computation on entire time series much faster. Here is why: if we can reduce the dimensionality of a window computation from w to k, then each individual correlation requires only $O(k)$ time and the entire caculation then requires only $O(k\ n^2)$ time. But the benefits don't end there. If k is small enough or, as we show in the tutorial, it can be split into smaller groups, then we can use a multidimensional index. This could in principal reduce the time to find the highest correlated pairs to $O(kn \log n)$.

The easiest path wanders...

Different data structures give different tradeoffs among speed, type of time series and accuracy. For example, price time series are well handled by Fourier transform-based reductions whereas return time series are best handled using sketches[3].

The next problem is to perform these computations at every timepoint, or nearly so. To achieve this, we distinguish among three time periods:

- timepoint – the smallest unit of time over which the system collects data, e.g. second.
- basic window – a consecutive subsequence of timepoints over which we maintain a synopsis incrementally.
- sliding window – a user-defined consecutive subsequence of basic windows over which the user wants statistics, e.g. an hour. The user might ask, "which pairs of stocks were correlated with a value of over 0.9 for the last hour?"

Our basic strategy then is to maintain a synopsis (e.g. Fourier coefficients, sketches) for each basic window of each time series and then when a basic window completes, incrementally update the synopsis for the sliding window ending at that time. This is the basis for our system StatStream (figure 2).

For many applications, lagged correlations are useful, i.e. the correlation between a window of from time $t1$ to $t1 + w$ in stream $s1$ with a window from time $t2$ to $t2+w$ for stream $s2$. These can be done with roughly the same efficiency[3].

Many problems in this area are open. Here is just one example: find the maximum window size w for which the above (lagged or unlagged) correlations are maximized. Others will be mentioned later.

2. ELASTIC BURST DETECTION

Consider the following burst detection application. The scientists use the astronomical telescope to constantly observe high-energy photons from the universe. When many photons observed, they assert the existence of a Gamma Ray Burst. The scientists hope to discover primordial black holes or completely new phenomena by the detection of Gamma Ray Bursts. The occurrences of Gamma Ray Bursts are highly variable, flaring on timescale of minutes to days. Once such a burst happens, it should be reported immediately. Other telescopes could then point to that portion of sky to confirm the new astrophysical event. The data rate of the observation is extremely high.

Burst detection is the activity of finding abnormal aggregates in data streams. Such aggregates are based on sliding windows over data streams. In some applications, we want to monitor many sliding window sizes simultaneously and to report those windows with aggregates significantly different from other periods. If the problem were to detect bursts over a fixed sized window, then there is a simple linear time algorithm: keep a running sum of the aggregates over that window size and raises an alarm every time the threshold is exceeded.

However, if we want to to detect bursts over many window sizes each with its own threshold (where those thresholds are monotonically increasing with window size), the problem becomes much more difficult.

Fortunately, we can get a time reduction if the thresholds are large enough so it is uncommon for them to be exceeded. That case is common in applications involving people, because people ignore alarms if they occur too frequently – "The Boy Who Cried Wolf" phenomenon – so alarms are designed to occur very infrequently.

The basic idea is to use a data structure called a *Shifted Binary Tree*. This data structure (figure 3) helps to detect bursts for many window sizes at once all by looking at just a logarithmic number of window sizes. The half-overlapping nature of the structure gives us the following lemma:

LEMMA 1. *Given a time series of length n and its shifted binary tree, any subsequence of length $w, w \leq 1 + 2^i$ is included in one of the windows at level $i + 1$ of the shifted binary tree.*

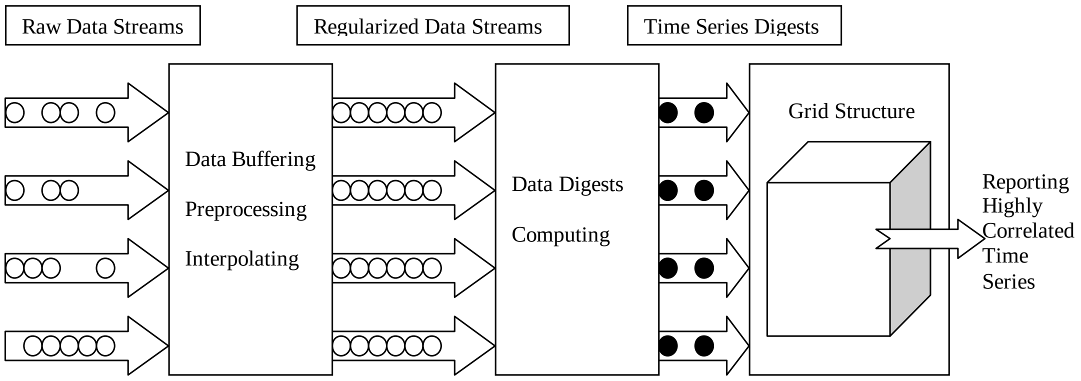


Figure 2: The software architecture of StatStream

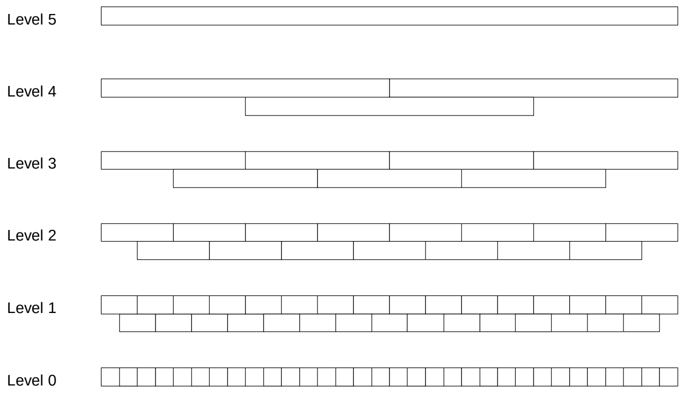
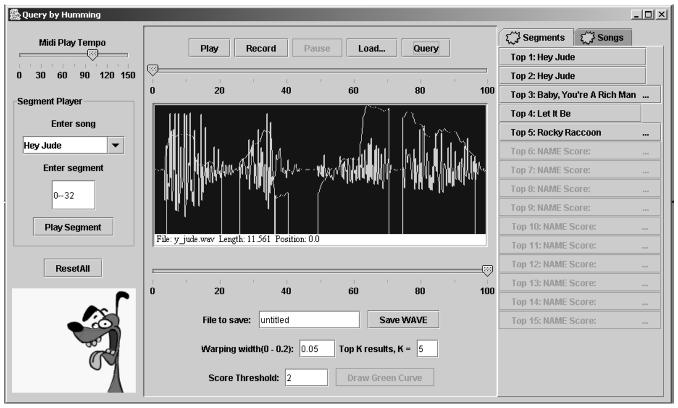


Figure 3: Shifted Binary Tree



Enter song: Hey Jude

Enter segment: 0--32

File: y_jude.wav Length: 11.561 Position: 0.0

File to save: untitled

Warping width(0 - 0.2): 0.05 Top K results, K = 5

Score Threshold: 2

Top 1: Hey Jude
Top 2: Hey Jude
Top 3: Baby, You're A Rich Man ...
Top 4: Let It Be
Top 5: Rocky Raccoon ...
Top 6: NAME Score: ...
Top 7: NAME Score: ...
Top 8: NAME Score: ...
Top 9: NAME Score: ...
Top 10: NAME Score: ...
Top 11: NAME Score: ...
Top 12: NAME Score: ...
Top 13: NAME Score: ...
Top 14: NAME Score: ...
Top 15: NAME Score: ...

Figure 4: The Graphical User Interface of HumFinder

The assumption about hard-to-exceed thresholds is important because a window of size w may be handled by a window of size nearly $2w$. If the threshold for w is hard to exceed then bursts that exceed the threshold in a window of size $2w$ will be rare as well. (We can reduce 2 to a smaller factor as well.)

Finding bursts over k windows on a stream of size n requires $O(kn)$ time if done naively. When thresholds are high, the Shifted Binary Tree approach can do this in $O(n)$ time. In a real high energy physics application whose goal it is to detect bursts of gamma rays on 14 windows, our technique was able to improve the time by a factor of 7. (This is less than the theoretical improvement because of overhead common to both the new and old implementations.)

Some open problems in this area concern finding bursts for many different kinds of events as well as finding correlations among bursts in different data streams.

3. QUERY BY HUMMING

What would you think if I sang out of tune, Would you stand up and walk out on me. Lend me your ears and I'll sing you a song, And I'll try not to sing out of key. – The Beatles.

The goal of a Query by Humming system is to allow an untrained user to find a song by humming part of the tune. This is difficult because most people have poor relative pitch and hum at inconsistent tempos.

There are two basic techniques used so far for this unsolved problem:

1. Segment the hum query into notes and query using string matching
2. Use dynamic time warping on the time series of the frequency. Dynamic time warping is tolerant to variations in tempo [2].

In our work we combine both approaches, trying to improve better speed and accuracy. We build on the work of several other research groups, notably Keogh [2]. Figure 4 shows our system.

Preliminary testing of the system on real people, including many researchers attending SIGMOD 2003, gave good performance and high satisfaction. We are still working on expanding our melody database and adapting the system to different hummers. How to scale up the system with high retrieval precision is still an open problem.

4. AQUERY: QUERY LANGUAGE FOR ORDERED DATA

Time series data is ordered so it is most convenient to manipulate it using a query language that handles order.

An *order-dependent query* is one whose result (interpreted as a multi-set) changes if the order of the input records is changed. In a stock-quotes database, for instance, retrieving all quotes concerning a given stock for a given day does not depend on order, because the collection of quotes does not depend on order. By contrast, finding the five price moving average in a trade table gives a result that depends on the order of the table. Query languages based on the relational data model can handle order-dependent queries only through add-ons. SQL:1999 [?], for example, permits the use of a data ordering mechanism called a "window" in limited parts of a query. As a result, order-dependent queries become difficult to write in those languages and optimization techniques for these features, applied as pre- or post-enumerating phases, are generally crude. The goal of AQuery is to offer users a natural extension to SQL that can be well optimized. Inthis it is inspired by other similar efforts [?, ?, ?].

To give you a flavor for the language, consider the schema Trades(ID, date, timeofday, volume, price), where ID is the ticker symbol of a traded security.

Consider the following query: for a given stock (MSFT) and a given date, find the best profit one could obtain by buying it and then selling it later that day (short selling – in which an item is sold before it is bought – is disallowed). Algorithmically, the solution is straightforward: compute the profit resulting from selling at each trade t by subtracting the price at t by the minimum price seen up until t. The answer to the query is the maximum of these profits. We can render this dirctly in AQuery:

```
SELECT    max(price - mins(price))
FROM      Trades
          ASSUMING ORDER timeofday
WHERE     ID = 'MSFT' AND date = '06/06/04'
```

This notion of ASSUMING says that the cross-product of the tables in the FROM clause are to be ordered semantically in ascending order by time for the purposes of the rest of the query. The mins function takes the running minimum of the price column (AQuery is column-oriented like its inspiration KDB). The expression price - mins(price) performs a vector to vector subtraction and then max takes the maximum of that.

When we say "ordered semantically" we mean that the implementation guarantees that the outcome is *as if* the ASSUMING clause had reordered the tables in the FROM clause. The implementation avoids this when possible. In this example, the reordering occurs only on the price column and only after the selection of trades for MSFT has taken place.

The tutorial discusses the design and implementation of AQuery as well as preliminary uses. The main open problem is how to handle streaming data.

5. CONCLUSION

Data arriving in time order (a data stream) arises in fields ranging from physics to finance to medicine to music, just to name a few. Often the data comes from sensors (in physics and medicine for example) whose data rates continue to improve dramatically as sensor technology improves. Further, the number of sensors is increasing, so correlating data between sensors becomes ever more critical in order to distill knowlege from the data. On-line response is desirable in many applications (e.g., to aim a telescope at a burst of activity in a galaxy or to perform magnetic resonance-based real-time surgery). These factors – data size, bursts, correlation, and fast response – present a host of challenges. We have only scratched the surface.

6. REFERENCES

[1] Christos Faloutsos, M. Ranganathan, and Yannis Manolopoulos. Fast subsequence matching in time-series databases. In *Proc. ACM SIGMOD International Conf. on Management of Data*, pages 419–429, 1994.

[2] Eamonn Keogh. Exact indexing of dynamic time warping. In *VLDB 2002,Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*, pages 406–417, 2002.

[3] Dennis Shasha and Yunyue Zhu. *High Performance Discovery in Time Series: Techniques and Case Studies*. Springer-Verlag, 2004.