# Incremental Methods for Simple Problems in Time Series: algorithms and experiments *

Xiaojian Zhao Xin Zhang Tyler Neylon Dennis Shasha

Courant Institute of Mathematical Sciences
New York University, NY 10012

{xiaojian,xinzhang,shasha}@cs.nyu.edu, neylon@cims.nyu.edu

**ABSTRACT**

*A time series (or equivalently a data stream) consists of data arriving in time order. Single or multiple data streams arise in fields including physics, finance, medicine, and music, 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 and as the number of sensors increases. So fast algorithms become ever more critical in order to distill knowledge from the data. This paper presents our recent work regarding the incremental computation of various primitives: windowed correlation, matching pursuit, sparse null space discovery and elastic burst detection. The incremental idea reflects the fact that recent data is more important than older data. Our StatStream system contains an implementation of these algorithms, permitting us to do empirical studies on both simulated and real data.*

## 1 MOTIVATION

Many applications generate multiple data streams. For example,

- The earth observing system data project consists of 64,000 time series covering the entire earth [1] though the satellite covers the earth in swaths, so the time series have gaps.
- There are about 50,000 securities traded in the United States, and every second up to 100,000 quotes and trades (ticks) are generated.

Such applications share the following characteristics:

- Updates come in the form of insertions of new elements rather than modifications of existing data.
- Data arrives continuously.
- One pass algorithms to filter the data are essential because the data is vast. Provided the filter does its job properly, there should be few enough candidates that even expensive detailed analysis per candidate will have only a modest impact on the overall running time. "Few enough" does not imply extremely high precision. In our experiments a precision of even 1% can still reduce computation times compared to a naive method by factors of 100 or more.

## 2 INCREMENTAL PRIMITIVES FOR DATA FUSION

Data fusion is the set of problems having to do with finding interesting relationships among multiple data streams. In this section we describe algorithms for some basic problems in data fusion: the incremental discovery of pairs of time series windows having high Pearson correlations (we call this problem *windowed correlation*), the incremental selection of representing time series among an evolving time series pool (*incremental Matching Pursuit*) and the detection of a sparse null space in a collection of time series. These representative problems offer a toolkit of techniques for many data fusion problems.

### 2.1 Preliminaries

A data stream, for our purposes, is a potentially unending sequence of data in time order. For specificity, we consider data streams that produce one data item each time unit.

Correlation over windows from the same or different streams has many variants. This paper focuses on synchronous and asynchronous (i.e., lagged) variations, defined as follows.

- (Synchronous) Given $N_s$ streams, a start time $t_{start}$, and a window size $w$, find, for each time window $W$ of size $w$, all pairs of streams $S_1$ and $S_2$ such that $S_1$ during time window $W$ is highly correlated (over 0.95 typically) with $S_2$ during the same time window. (Possible time windows are $[t_{start} \cdots t_{start+w-1}]$, $[t_{start+1} \cdots t_{start+w}]$, $\cdots$ where $t_{start}$ is some start time.)
- (Asynchronous correlation) Allow shifts in time. That is, given $N_s$ streams and a window size $w$, find all time windows $W_1$ and $W_2$ where $|W_1| = |W_2| = w$ and all pairs of streams $S_1$ and $S_2$ such that $S_1$ during $W_1$ is highly correlated with $S_2$ during $W_2$.

*This work has been partly supported by the U.S. National Science Foundation under grants: NSF IIS-9988345,N2010-0115586, and MCB-0209754.

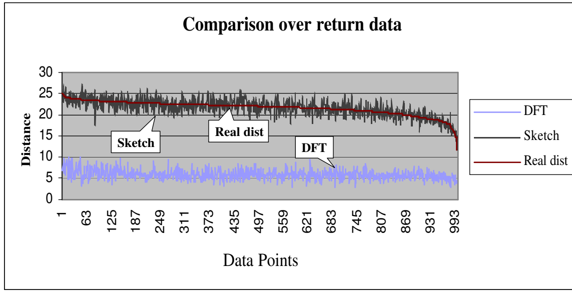



Figure 1: Discrete Fourier Transform and Sketch estimates for return data.

## 2.2 Related work

In some applications, time series exhibit a fundamental degree of regularity, allowing them to be represented by the first few few coefficients of a Fast Fourier or Wavelet Transform [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]. with little loss of information. Random walk data such as price time series is of this *cooperative* character.

By contrast, the time series in some applications resemble white noise, so their energy is not concentrated in only a few frequency components. For these *uncooperative* applications we adopt a sketch-based approach, building on work by Johnson et al. [13], Kushikvitz et al. [14], Indyk et al. [15], and Achlioptas [16].

## 2.3 The challenge and our contributions

In our earlier work [17, 18] we showed how to solve the windowed correlation problem addressed in this paper. That algorithm works quite well in the cooperative setting using high quality digests obtained based on Fourier transforms. Unfortunately, many applications generate uncooperative time series. Stock market returns (i.e., the change in price from one time period (e.g., day, hour, or second) to the next divided by the price in the first time period, symbolically $(p_{t+1} - p_t)/p_t$) for example are "white noise-like." That is, there is almost no correlation from one time point to the next.

Empirical studies confirm that sketches work much better than Fourier methods for uncooperative data. Figure 1 compares the distances of the Fourier and sketch approximations for 1,000 pairs of 256 timepoint windows having a basic window size of length 32. As you can see, the sketch distances are close to the real distances. On the other hand, the Fourier approximation is essentially never correct.

For collections of *uncooperative* time series, we proceed as follows:

1. We adopt a sketch-based approach on which a "structured random vectors" scheme is employed to reduce the time complexity.

2. A sketch vector partition strategy is used to overcome the "curse of dimensionality."

3. Combinatorial design and bootstrapping is combined to optimize and validate the parameter appropriate to any specific application.

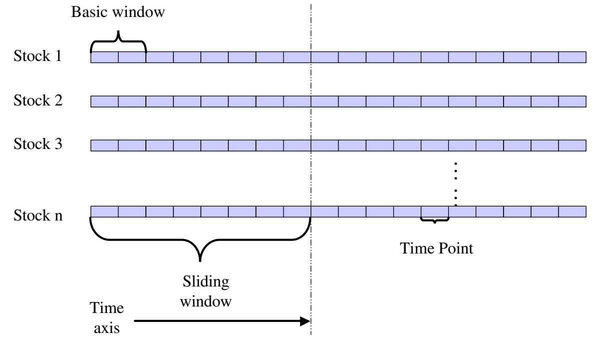



Figure 2: Sliding windows and basic windows.

The end result is a system architecture that, given the initial portions of a collection of time series streams, will determine (i) whether the time series are cooperative or not; (ii) if so, it will use Fourier or Wavelet methods; and (iii) if not, it will discover the proper parameter settings and apply them to compute sketches of the evolving data streams.

Thus, our contributions are of two kinds: (1) a greatly improved and more general solution to the on-line correlation problem; and (2) a synthesis of techniques – sketches, structured random vectors, combinatorial design with neighborhood search, and bootstrapping.

## 2.4 Algorithmic Ideas

Following [18, 17], our approach begins by distinguishing among three time periods from smallest to largest.

- timepoint – the smallest unit of time over which the system collects data, e.g., a second.

- basic window – a consecutive subsequence of timepoints over which the system maintains a *digest* (i.e., a compressed representation) e.g., two minutes.

- 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 streams were correlated with a value of over 0.9 for the last hour?"

Figure 2 shows the relationship between sliding windows and basic windows.

The use of the intermediate time interval called the basic window yields several advantages [18, 17], mainly on the near online response rates and free choice of window size.

## 2.5 The Sketch Approach

For each random vector $\mathbf{r}$ of length equal to the sliding window length $sw = nb \times bw$, we compute the dot product with each successive length $sw$ portion of the stream (successive portions being one timepoint apart and $bw$ being the length of a basic window). As noted by Indyk [19], convolutions (computed via Fast Fourier Transforms) can perform this efficiently off-line. The difficulty is how to do this efficiently online.

Our approach is to use a "structured" random vector. The apparently oxymoronic idea is to form each structured random vector $\mathbf{r}$ from the concatenation of $nb$ random vectors: $\mathbf{r} = \mathbf{s_1}, ..., \mathbf{s_{nb}}$ where each $\mathbf{s_i}$ has length $bw$. Further each $\mathbf{s_i}$ is either $\mathbf{u}$ or $-\mathbf{u}$, and $\mathbf{u}$ is a random vector in $\{1, -1\}^{bw}$. This choice is determined by a random binary $k$-vector $\mathbf{b}$: if $b_i$=1, $\mathbf{s_i}$=$\mathbf{u}$ and if $b_i$=0, $\mathbf{s_i}$=$-\mathbf{u}$. The structured approach leads to an asymptotic performance of $O(nb)$ integer additions and $O(\log bw)$ floating point operations per datum and per random vector. In our applications, we see 30 to 40 factor improvements over the naive method.

In order to compute the dot products with structured random vectors, we first compute dot products with the random vector $\mathbf{u}$. We perform this computation by making the inner products once every $bw$ timesteps. Then each dot product with $\mathbf{r}$ is simply a sum of $nb$ already computed dot products. The use of structured random vectors reduces the randomness, but experiments show that this does not appreciably diminish the accuracy of the sketch approximation. (For those who are interested in the algorithm, please refer to [20]).

Though structured random vectors enjoy good performance, a clever use of unstructured (that is, standard) random vectors together with convolutions can lead to an asymptotic cost of $O(\log \; sw \log(sw/bw))$ floating point multiplications per datum. Structured random vector approaches use $O(\log bw)$ multiplications and $O(\log(sw/bw))$ additions per datum. For the problem sizes we consider in this paper, the structured random vector approach is faster, though in principle it must be weighed against the small loss in accuracy.

## 2.6 Overcoming High Dimensionality by Partitioning

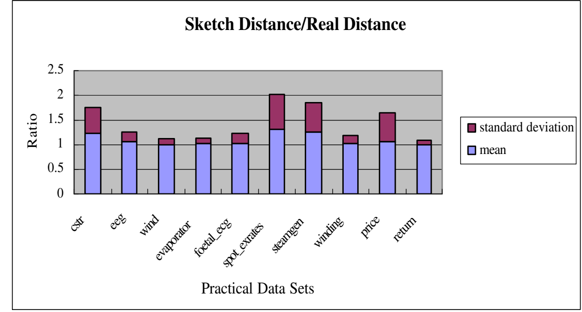
In many applications, sketch vectors are of length up to 60. (In such a case, there are 60 random vectors to which each window is compared and the sketch vector is the vector of the dot products to those random vectors). Multi-dimensional search structures don't work well for more than 4 dimensions in practice [18]. Comparing each sketch vector with every other one destroys scalability though because it introduces a term proportional to the square of the number of windows under consideration.

For this reason, we adopt an algorithmic framework that partitions each sketch vector into subvectors and builds data structures for the subvectors. For example, if each sketch vector is of length 40, we might partition each one into ten groups of size four. This would yield ten data structures. We then combine the closeness results of pairs from each data structure to determine an overall set of candidates of correlated windows. Sketch vector partitioning introduces a search space in four parameters, which we explore using combinatorial design.

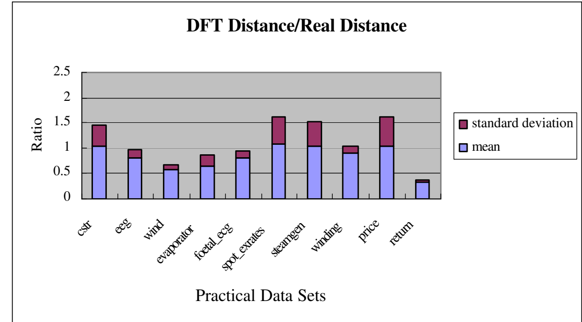
## 2.7 Empirical Study

Our empirical tests are based on financial data and on test sets from the UC Riverside repository [21]. [1]

[1]The stock data in the experiments are end-of-day prices from 7,861 stocks from the Center for Research in Security Prices (CRSP) at Wharton Research Data Services (WRDS) of the University of Pennsylvania [22]. All the other empirical data sets came from the UCR Time Series Data Mining Archive [21] maintained by Eamonn Keogh.




(a) Sketch




(b) DFT

Figure 3: DFT distance versus sketch distance over empirical data

The Hardware is a 1.6G, 512M RAM PC running RedHat 8.0. The language is K (www.kx.com).

### 2.7.1 Experiment: cooperativeness is a property of an application

In this experiment, we took a window size of 256 ($sw = 256$ and $bw = 32$) across 10 data sets and tested the accuracy of the Fourier coefficients as an approximation to distance compared with structured random vector-based sketches. Figure 3 shows that the Discrete Fourier Transform-based distance performs badly in some data types while our sketch based distance works stably across all the data sets.

### 2.7.2 Performance Tests

The previous subsection shows that the sketch framework gives a sufficiently high recall and precision. The next question is what is the performance gain of using (i) our sketch framework as a filter followed by verification on the raw data from individual windows compared with (ii) simply comparing all window pairs. Because the different applications have different numbers of windows, we take a sample from each application, yielding the same number of windows.

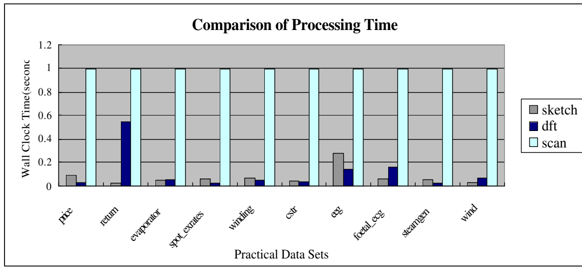
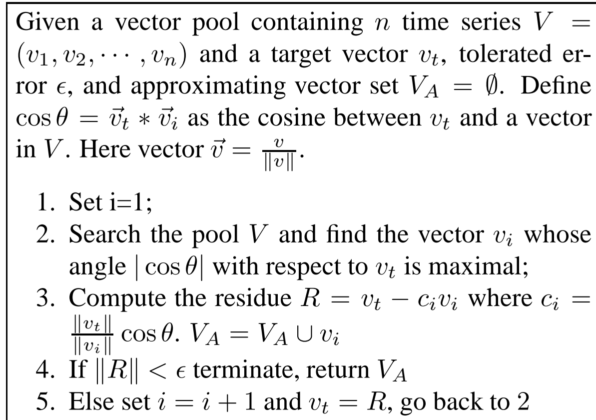


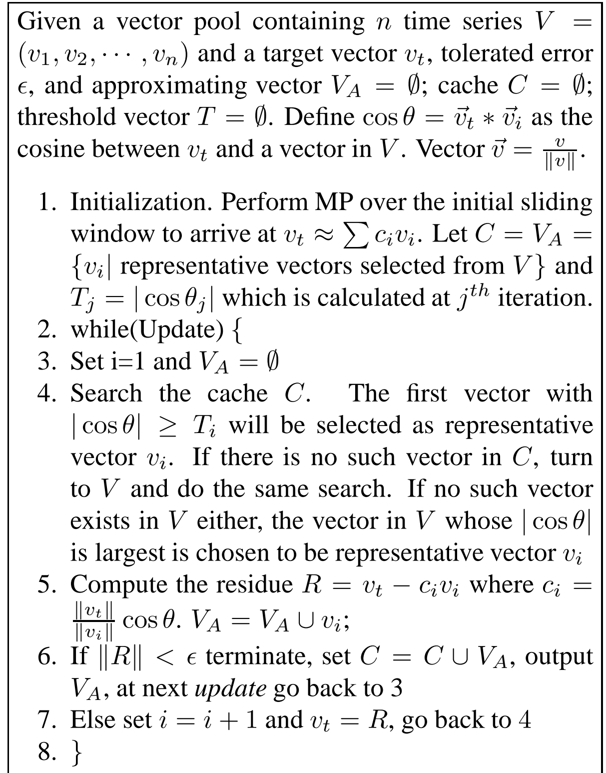

Figure 4: System performance over a variety of datasets.



Given a vector pool containing $n$ time series $V = (v_1, v_2, \cdots, v_n)$ and a target vector $v_t$, tolerated error $\epsilon$, and approximating vector set $V_A = \emptyset$. Define $\cos\theta = \vec{v}_t * \vec{v}_i$ as the cosine between $v_t$ and a vector in $V$. Here vector $\vec{v} = \frac{v}{\|v\|}$.

1. Set i=1;
2. Search the pool $V$ and find the vector $v_i$ whose angle $|\cos\theta|$ with respect to $v_t$ is maximal;
3. Compute the residue $R = v_t - c_i v_i$ where $c_i = \frac{\|v_t\|}{\|v_i\|}\cos\theta$. $V_A = V_A \cup v_i$
4. If $\|R\| < \epsilon$ terminate, return $V_A$
5. Else set $i = i + 1$ and $v_t = R$, go back to 2

Figure 5: Matching Pursuit Algorithm



Given a vector pool containing $n$ time series $V = (v_1, v_2, \cdots, v_n)$ and a target vector $v_t$, tolerated error $\epsilon$, and approximating vector $V_A = \emptyset$; cache $C = \emptyset$; threshold vector $T = \emptyset$. Define $\cos\theta = \vec{v}_t * \vec{v}_i$ as the cosine between $v_t$ and a vector in $V$. Vector $\vec{v} = \frac{v}{\|v\|}$.

1. Initialization. Perform MP over the initial sliding window to arrive at $v_t \approx \sum c_i v_i$. Let $C = V_A = \{v_i|$ representative vectors selected from $V\}$ and $T_j = |\cos\theta_j|$ which is calculated at $j^{th}$ iteration.
2. while(Update) {
3. Set i=1 and $V_A = \emptyset$
4. Search the cache $C$. The first vector with $|\cos\theta| \geq T_i$ will be selected as representative vector $v_i$. If there is no such vector in $C$, turn to $V$ and do the same search. If no such vector exists in $V$ either, the vector in $V$ whose $|\cos\theta|$ is largest is chosen to be representative vector $v_i$
5. Compute the residue $R = v_t - c_i v_i$ where $c_i = \frac{\|v_t\|}{\|v_i\|}\cos\theta$. $V_A = V_A \cup v_i$;
6. If $\|R\| < \epsilon$ terminate, set $C = C \cup V_A$, output $V_A$, at next *update* go back to 3
7. Else set $i = i + 1$ and $v_t = R$, go back to 4
8. }

Figure 6: Incremental Matching Pursuit Algorithm

Figure 4 compares the results from our system, a Fourier-based approach, and a linear scan over several data sets. To perform the comparison we normalize the results of the linear scan to 1. The figure shows that both the sketch-based approach described here and the Fourier-based approach are much faster than the linear scan. Neither is consistently faster than the other. However as already noted, the sketch-based approach produces consistently accurate results unlike the Fourier-based one.

## 3 Matching Pursuit

Matching Pursuit(MP) is an algorithm, introduced by Stephan Mallat and Zhifeng Zhang [23], for approximating a target vector by greedily selecting a linear combination of vectors from a dictionary. Figure 5 holds pseudocode for the MP algorithm. MP takes as input a target vector $v_t$ and a set of vectors $V$ from which it quickly extracts as output a smaller subset $V_A$ along with weights $c_i$ so that $v_t \approx \sum_{v_i \in V_A} c_i v_i$.

MP has been applied to many applications such as image processing [24], physics [25, 26], medicine [27, 28] etc. Several researchers have proposed fast variants of the algorithm [29, 30, 31]. However, to the best of our knowledge none has discussed how to apply MP incrementally to a group of time series. In this section, we will give a brief description of our incremental matching pursuit technique.

### 3.1 Incremental MP

Imagine a scenario where a group of representative stocks will be chosen to form an index e.g. for the Standard and Poor's (S&P) 500. This situation can be considered as a version of MP where the candidate pool consists of all the stock price vectors in the market and the target vector is the summation of all the vectors weighted by their volumes traded. The incremental problem is to adjust the set $V_A$ and the corresponding weights for each update to the underlying vectors periodically. Formally, given a target vector and a vector pool of size $n$, whenever an update takes place over both the target vector and the vectors in the pool, MP is performed. Here an operation "update" on a vector is defined such that a new basic window (bw) of data are inserted to the head and data of length bw are dropped off as outdated from the tail.

A naive method for the problem is straightforward. Whenever an update happens MP is run. However, recomputing MP entirely for each new sliding window is inefficient. A better idea is to reuse the previously computed linearly combination of vectors. We may expect a slight change of the approximating vector set $V_A$ and perhaps a larger change in the weights, when the basic window is small (Reminder: A basic window is a sequences of time points as defined in Figure 2). Whether this holds for an application is an empirical question. In our experiments on stock prices, this holds for very small basic windows only. With a relatively large basic window size (e.g. 30 time points), only the most significantly weighted approximating vectors from $V_A$ remain important. Moreover, any perturbation may direct the approximation to a different path and results in a different set $V_A$.

Our solution therefore lies in the angle space — the information given by the angle vector $(\cos\theta_1, \cos\theta_2, \ldots)$ where $\cos\theta_i = \vec{v}_t * \vec{v}_i$ for each $v_i \in V_A$.

### 3.2 Opportunities in Angle Space

The angle vector $(\cos\theta_1, \cos\theta_2, \ldots)$ appears in experiments to change only slightly over incremental vector updates. This gives us a clue to a promising, though heuristic algorithm. The basic idea is that although the approximating *vectors* $v_i$ may vary a lot between two consecutive sliding windows, every *angle* $\cos\theta_i$ of the corresponding rounds remains relatively consistent.

Therefore instead of searching through all of $V$ for the vector best approximating the residue or new target vector at each iteration, if a vector in the pool is found having $|\cos\theta|$ with respect to the current target vector that is larger than a certain threshold, then it is chosen. If such a vector doesn't exist, the vector with largest $|\cos\theta|$ is chosen as usual. This vector is selected as the representative vector and its residue with the target vector will be the new target vector for the next round of search. Here the difference from the standard algorithm resides in the search strategy: whenever a "good" vector is found, the current round of iteration is stopped, as opposed to the exhaustive search in Figure 5. The gain from this new algorithm is obvious: no need to compute the $\cos\theta$ between target vector and *all* the vectors in the pool $V$ in each iteration round.

One major concern with this method is the approximation power. Since the resultant vector of each search step is not optimal — in order words, not the one with largest $|\cos\theta|$ — the overall approximation power may be compromised. The empirical study shows that this is not a problem. We may carefully choose a vector of thresholds to yield results comparable to those calculated by regular MP.

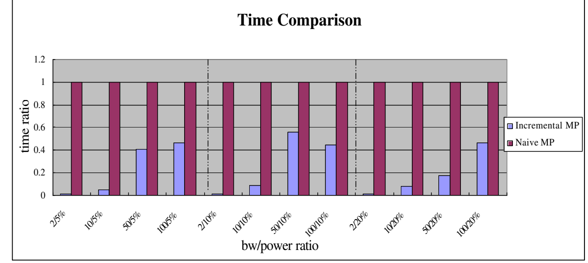
Here is an example:

Given a threshold vector, say, $T = \{0.9, 0.8, 0.7, 0.6, 0.5, \ldots\}$, a target vector $v_t$ and a candidate vector pool $V$, the first iteration is conducted by computing $\cos\theta$ between $v_t$ and $v$ in $V$ one by one. The first vector found with $|\cos\theta| \geq 0.9$ will be selected as the representative vector in this iteration, naming it $v_1$. Otherwise, if there is no such vector, the vector in $V$ with largest $|\cos\theta|$ is chosen to be the representative vector $v_1$. Then update the target vector by $v_t = v_t - P(v_t, v_1)$, where $P(v_t, v_1)$ is the projection of $v_t$ onto $v_1$. Test the termination criterion; if it is not met, start the next iteration. The second iteration is similar to the first one — the only difference being that the threshold for comparison is 0.8. Continue the algorithm with 0.7 in the third iteration, 0.6 in the fourth iteration, etc. until the termination condition is satisfied.

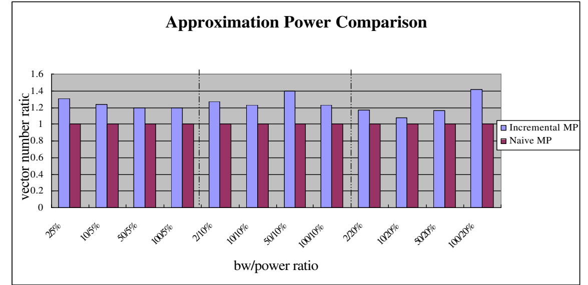
Figure 6 gives the full pseudocode.

In practical applications, we apply the regular nonincremental MP to the initial sliding window. Its $|\cos\theta|$ at each iteration will be used to initialize the threshold vector $T$. When the approximation power of using this angle vector $T$ gets unacceptably bad due to new data characteristics, the threshold vector is reinitialized to reflect the changes.

One bonus of this algorithm comes from the cache technique. Just as described above, the approximating vectors $V_A$ in the present sliding window may appear with high probability in the search launched for the following sliding window, we may take advantage of this property by keeping track of a cache $C$ for the pool $V$. The representative vector search is therefore performed first from the cache. If no "good" vector can be found in the cache, the rest of the vector pool is searched.




(a) Time Comparison




(b) Approximation Power Comparison

Figure 7: Time and Approximation Power Comparison

### 3.3 Empirical Study

The experimental data comes from the same sources as in the last section. Additional synthesized random walk time series are also used to illustrate gains in other applications. Similar results also hold for data distributions such as white noise (not shown here).

Figure 7 compares the results from incremental MP and naive MP. The sliding window size is fixed at $sw = 200$ time points. Whenever an update event happens, both incremental and regular MP are triggered. The power ratio in the figure is defined as $\frac{\|residue\|}{\|Original\ target\ vector\|}$. So a small power ratio entails more iterations. Performance is measured in terms of average time costs and returned approximating vector number (i.e. the average size of $V_A$ in each sliding window). To better demonstrate the comparison, we normalize the results of regular MP to 1.

One apparent observation in Figure 7(a) is the significant speed improvement when $bw$ is small compared to the sliding window size. This substantial speedup derives largely from the vector cache. Figure 7(b) shows that the number of vectors required by incremental MP is no more than 1.4 times the number required by naive MP to achieve the same representation fidelity.

The experimental results suggest the potential application of incremental MP in a real-time setting where rapid

response is as important as discovering a small approximating set $V_A$.

## 4 Finding Small Linear Identities

### 4.1 Problem & Motivation

Given data in the form of time series, it may be that some small subsets of the series are related to each other by simple linear equations. For example, if each time series represents the latest 100 prices of a certain stock, then a relation of the type MSFT = HPQ + IBM tells us that, for the last 100 trades of these stocks, Microsoft's share price has been exactly the sum of Hewlett-Packard's and IBM's.

An investor with such extra knowledge could take advantage of it in several ways. If we know the price of HPQ and IBM, then we may infer that of MSFT. If the investor wishes to have a portfolio split evenly between HPQ and IBM, then some overhead can be saved by simply investing in MSFT. On a larger scale, we could effectively invest in an equivalent of the entire S&P 500 (a weighted linear combination of 500 different stocks) by actually investing in a much smaller number. Of course these techniques can be extended to any set of time series with a propensity for linear dependencies — not just stocks.

One principle of the following work is the idea that smaller linear identities are less likely to be mere coincidences. If the equality involves fewer unknowns, then we have more power to use less information to make useful deductions.

From a mathematical perspective, we can consider the time series $\{x_1, \ldots, x_n\}$ as the columns of a matrix $X$. There is a 1-1 correspondence between sparse (mostly zero) column vectors $v$ with $Xv = 0$ and small linear equations between the vectors $x_i$. For example, if $v_2 = 1, v_5 = -1, v_{11} = 3$ and all other coordinates are zero, then equation $Xv = 0$ corresponds with $x_2 + 3x_{11} = x_5$. Hence we are trying to solve the

**Sparse Null Space Problem** Given an $m \times n$ matrix $X$ with corank $c$, find a full rank, optimally sparse $n \times c$ matrix $N$ with $XN = 0$.

An $n \times c$ matrix $N$ having full rank means that rank$(N) = \min(n, c)$. In our case, $c \leq n$, so all the columns of $N$ must be linearly independent (this excludes, for example, the trivial solution $N = 0$). By "optimally sparse," we mean that $N$ contains as many zero entries as possible. It turns out that maximizing the global number of zeros in $N$ also maximizes the column (local) sparsity, which is our real motivation. It can also be shown that, for each time series $x_i$, globally solving Sparse Null Space also locally finds a minimal linear equality involving $x_i$ (if one exists).

In addition, we would like to find our Sparse Null Space in an incremental, time-efficient manner to take advantage of the fact that our data consists of constantly evolving time series.

### 4.2 Related Work

It is easy to see that finding a Sparse Null Space is really just a (polynomially-equivalent) version of Matrix Sparsification: given matrix $B$, find a column-equivalent matrix $A$ with as many zero entries as possible. In the 1980's, L.J. Stockmeyer showed that this problem (and hence ours as well) is NP-hard (see [32] for the proof). All efforts thereafter have focused on using heuristics to achieve approximate sparsification in polynomial time.

In 1979, Topcu [33] introduced the Turnback algorithm, which gives a banded null matrix for a banded input matrix of the Sparse Null Space problem. However, this algorithm does not guarantee any more sparsity than using mere Gaussian elimination on the columns of an arbitrary null matrix.

In 1984, Hoffman and McCormick [32] found a class of matrices on which Matrix Sparsification was solvable in polynomial time. They also introduced the idea of using combinatorics and matchings within bipartite graphs to help sparsify matrices. A few years later, Coleman & Pothen [34], and Gilbert & Heath [35] formulated modifications to both the turnback techniques and the graph matching methods.

### 4.3 Our Approach

Instead of attempting to find a sparse null matrix from scratch, we assume that we have an already sparsified null matrix which must be updated as new rows are introduced — that is, as the time series evolve. This already sparsified null matrix is provided by the previous iterations of our own algorithm, which is capable of both building and maintaining this information.

Here we will sketch two techniques for sparsely tracking the null space of a set of time series. First we'll notice a trick for following the null space of a set of full history time series — that is, a matrix that is continuously gaining new rows. Then we'll see a generalization of the first method which can be applied to a partial history sliding window — a matrix that continues to gain new rows and simultaneously lose old ones.

#### 4.3.1 Tracking Full History Time Series

Our $n$ time series form the columns of matrix $A_t$, which has $t$ rows representing the $t$ time ticks passed thus far. Let $a_i$ represent the $i^{\text{th}}$ row of $A_t$ — the data from the $i^{\text{th}}$ time tick. Now we may recursively define

$$d_i := \mathsf{nmat}(a_i N_{i-1}) \qquad N_i := N_{i-1} d_i \qquad (1)$$

where $N_0$ is the identity matrix, and nmat returns a sparse null matrix of a row matrix (which is easily computed). It can be shown that $N_t$ is then a null matrix of $A_t$ of maximum rank. It can also be shown that each of the $d_i$, if nmat is computed properly, are optimally sparse. Thus this algorithm takes advantage of the fact that the product of sparse matrices is still likely to be somewhat sparse.

If $\ell$ is the maximum number of nonzeros in any column of $N_{i-1}$, then the time complexity of a single step (adding a single row) for this algirthm is $O(\ell n)$.

In order to help keep the null matrices extremely sparse, we may optionally introduce $\tilde{N}_i := \mathsf{smat}(N_i)$,

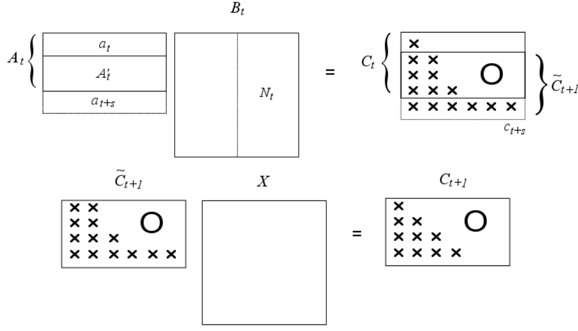



Figure 8: One iteration of tracking the null space

where smat is an arbitrary incremental column sparsification technique. Replacing each "$N_{i-1}$" with "$\tilde{N}_{i-1}$" in (1) above does not affect the correctness of the technique — specifically, each new $N_t$ is still guaranteed to be a full null matrix for the full history $A_t$.

However, in section §4.4 below, our experimental results reveal that this technique works relatively well without this optional (smat) step.

#### 4.3.2 Tracking a Partial History Sliding Window

We will need to do a little more work when rows are also being lost. Suppose our sliding window has constant width $s$. Then we can consider the $s \times n$ matrix $A_t$ as consisting of rows $a_t$ through $a_{t+s-1}$ (as above, $n$ is still the number of time series). It will be convenient to let $A'_t$ consist of rows $a_{t+1}$ through $a_{t+s-1}$, so that, writing informally, $A'_t = A_t - a_t$.

Along with each matrix $A_t$, we will track two corresponding matrices, $B_t$ which is $n \times n$ and invertible, and $C_t$ which is in column echelon form (lower triangular) with sparser rows near the top. We will maintain the matrix equality $A_t B_t = C_t$. By doing so, the last $c$ columns of $B_t$ will form a full null matrix of $A_t$, where $c = \text{corank}(A_t)$.

Our algorithm consists entirely of finding $B_{t+1}$ and $C_{t+1}$ based on $B_t, C_t$ and the new row $a_{t+s}$. Notice that $A_t = \binom{a_t}{A'_t}$. If $A_t B_t = C_t$ and $c_{t+s} = a_{t+s} B_t$ then, using $C'_t = C_t - c_t$, we have $\binom{A'_t}{a_{t+s}} B_t = \binom{C'_t}{c_{t+s}}$. Call this last matrix $\tilde{C}_{t+1}$ and notice that it is *almost* in column echelon form, and so is easy to reduce. The entire algorithm can be summarized as:

$$
\begin{aligned}
c_{t+s} &:= a_{t+s} B_t \\
B_{t+1} &:= B_t \cdot \text{col_reduce}(\tilde{C}_{t+1}) \\
C_{t+1} &:= \tilde{C}_{t+1} \cdot \text{col_reduce}(\tilde{C}_{t+1})
\end{aligned}
$$

(This pseudocode is illustrated in Figure 8.) Here, the function col_reduce$(X)$ returns a matrix $Y$ so that $XY$ is in column echelon form. The actual implementation of this pseudocode should of course efficiently perform the column reduction by acting directly on $B$ and $C$ at once.

Each iteration can take advantage of the almost-column-reduced nature of $C$ for improved time efficiency. In addition, any sparsity in the null space within $B$ is minimally perturbed in a single incremental iteration.

Each iteration of this algorithm takes time $O(n^2)$.

As above, we may wish to augment each iteration by applying an smat algorithm to the null space basis within $B$. Doing so at each step does not alter the invariant condition $AB = C$ or otherwise affect the correctness of the algorithm.

In our experimental results (§4.4), we attempt extra sparsification (a version of smat) only when the corank (dimension of the null space) increases. When this happens, we attempt to sparsify the new column by greedily performing any column-pairwise sparsifications offered by any of the old columns. This process runs in time $O(n^2)$. In our experiments, the corank rarely increased.

Notice that if we are only adding (not subtracting) columns to $A_t$, then the column reduction of $\tilde{C}_{t+1}$ amounts to multiplying the null space portions of $B_t$ and $\tilde{C}_{t+1}$ by nmat$(a_{t+s} B_t)$, which corresponds exactly with our technique from §4.3.1 above.

### 4.4 Experimental Results

We used both real (stock prices) and simulated data to test each of these two algorithms (full or partial history sliding window). Our real data is from the NYSETAQ stock database, and contains trade prices of a randomly selected subset of 500 stocks. The data was normalized so that 100 time ticks correspond to 1 full day (9am–4pm) of trading. The simulated data consists of time series which were mostly random linear combinations of the several previous values, and occasionaly a completely new random set of values.

Our algorithm is compared with (nonincremental) back substitution. The back substitution is performed at each time tick on the full sliding window. Recall that the turnback algorithm, aside from giving a banded structure, does not guarantee any increase in sparsity over back substitution; hence these sparsity comparisons should be similar to those from a turnback experiment as well.

Although time comparisons between these two algorithms are somewhat unfair, it is useful to see how well the sparsity of our much faster incremental algorithms fares against that of the slower nonincremental algorithm. (The authors are unaware of any other incremental sparse null space tracking algorithms to compare against.)

Initially, both algorithms (for full or parital history sliding windows) achieve sparsity nearly as efficiently as back substitution. However, the stability and sparsity of the partial history sliding window algorithm deteriorates over time. To deal with this problem, we periodically restarted the tracked decomposition $AB = C$. Luckily, when the need to periodically recalibrate is anticipated, we can do so in an incremental fashion (using essentially the full history technique from §4.3.1), without any awkward pauses in computation.

- Figure 9 shows how the density (percentage of the null matrix which is nonzero) evolves over 60 iterates of our algorithm on the simulated data. Notice the significant improvement we gain by periodically

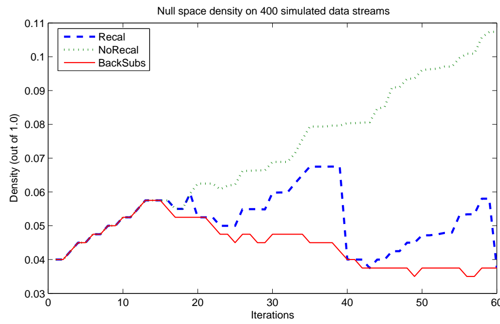



Figure 9: Density of null space comparison on simulated data. Back substitution remains between 3–6% while our periodically recalibrated algorithm achieves between 3–7%.

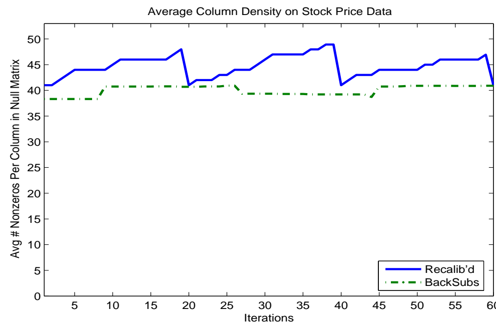



Figure 10: Average number of nonzeros per column in stock price data. A full column could have contained up to 500 nonzeros. Back substitution alternated between 38–41 while our algorithm accomplished between 41–49.

recalibrating our decomposition (which occurs every 20 iterations in the figures).

- Figure 10 compares the average number of nonzero entries per column between our algorithm and back substitution run on the stock price data. Although back substitution consistently does at least as well, the factor of improvement is small – in this experiment, the average ratio is 1.11 and the maximum is 1.25. In practice, a factor of 1.11 would mean that the linear identities we are finding contain 11% more terms than they would with the (much slower) back substitution method. For example, in the stock market application of tracking an index, the larger dependent set would mean that 11% more stocks would be needed to track the index.

- Figure 11 illustrates how the time complexity of our algorithm grows with respect to the width of the partial history sliding window. Each data point is the ratio between one iteration of our algorithm in proportion to back substitution on 1000 columns of simulated data. (In this experiment our algorithm always ran at least 49 times faster than the alternative.)

In summary, these incremental algorithms are time efficient and maintain sparsity well in comparison with back substitution. In addition, they are easily extensible

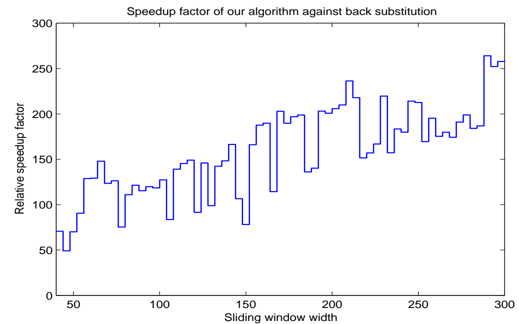



Figure 11: Relative speedup factor between our algorithm and back substitution over partial history sliding windows of increasing width. Tests were run on 1000 streams of simulated data. The factor represents the time efficiency gained over one iteration.

with augmented null matrix manipulation via an optional smat function.

# 5 Elastic Burst Detection

## 5.1 Problem Statement

A burst is a large number of events occurring within a certain period of time. It's a noteworthy phenomenon in many natural and social processe, for example, a burst of trading volume in some stock might indicate insider trading.

As an interesting and important phenomenon, burst discovery has attracted more and more interest under different settings. For example, [37] models the bursty behaviors in self- similar time series, such as disk I/O activity, network traffic; [38] studies the bursty and hierarchical structure in temporal text stream, such as emails, news articles, etc; [39] mines the bursty behavior in the query logs of the MSN search engine; [40, 41, 42] study the problem of detecting significant spatial clusters in multidimensional space. [43, 44] use multiresolution synopsis to estimate the number of 1's in the last $N$ elements in a 0-1 stream and the sum of bounded integers in an integer stream. Our interest here is in one dimensional data stream.

If the length $w$ of the time window when a burst occurs is known in advance, the detection can easily be done in linear time by keeping a running count of the number of events in the last $w$ time units. However, in many situations, the window length is unknown a priori. For example, interesting gamma ray bursts could last several seconds, several minutes or even several days. One has to monitor bursts across multiple window sizes. Furthermore, in many data applications, looking at different window scales at the same time gives different insight into the data.

The elastic burst detection problem [18] is to simultaneously detect bursts across multiple window sizes. Formally:

**Problem 1** *Given a non-negative time series $x_1, x_2, ...$, a set of window sizes $W = w_1, w_2..., w_m$, a monotonic, associative aggregation function $A$ (such as "sum" or "maximum") that maps a consecutive sequence of*

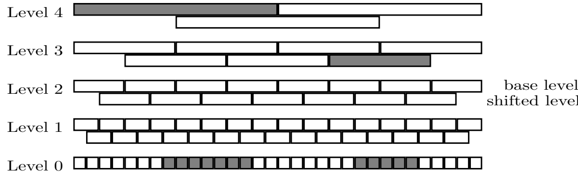



Figure 12: Shifted Binary Tree (SBT) and the shadow property. The shadowed subsequences of size 7 and 5 are included in the shadowed windows at level 4 and 3 respectively.

*data elements to a number (it is monotonic in the sense that* $A[x_t \cdots x_{t+w-1}] \leq A[x_t \cdots x_{t+w}]$*, for all* $w$*), and thresholds associated with each window size,* $f(w_j)$*, for* $j = 1, 2, ..., m$*, the elastic burst detection is the problem of finding all pairs* $(t, w)$ *such that* $t$ *is a time point and* $w$ *is a window size in* $W$ *and* $A[x_t \cdots x_{t+w-1}] \geq f(w)$*.*

A naive algorithm is to check each window size of interest one at a time. To detect bursts over the $m$ window sizes in a sequence of length $N$ would then require $O(mN)$ time. This is unacceptable in a high-speed data stream environment.

### 5.2 Shifted Binary Tree (SBT)

In [18], we presented a simple data structure called the *Shifted Binary Tree (SBT)* that could be the basis of a filter that would detect all bursts, yet perform in time independent of the number of windows when the probability of bursts is very low.

A Shifted Binary Tree is a hierarchical data structure inspired by the Haar Wavelet Tree. The leaf nodes of this tree (denoted level 0) have a one-to-one correspondence to the time points of the incoming data; a node at level 1 aggregates two adjacent nodes at level 0. In general, a node at level $i+1$ aggregates two nodes at level $i$, so covers $2^{i+1}$ time points. The SBT includes a shifted sublevel to each base level above level 0. In the shifted sublevel $i$, the corresponding windows are still of length $2^i$ but those windows are shifted by $2^{i-1}$ from the base level. Figure 12 shows an example of a Shifted Binary Tree.

The overlapping between the base levels and the sublevels guarantees that all the windows of length $w$, $w \leq 1 + 2^i$, are included in one of the windows at level $i+1$. Because the aggregation function $A$ is monotonically increasing, $A[x_t \cdots x_{t+w-1}] \leq A[x_t \cdots x_{t+w+c}]$, for all $w$ and $c$. So if $A[x_t \cdots x_{t+w+c}] \leq f(w)$, then surely $A[x_t \cdots x_{t+w-1}] \leq f(w)$. The Shifted Binary Tree takes advantage of this monotonic property as follows: each node at level $i+1$ is associated with the threshold value $f(2 + 2^{i-1})$. If more than $f(2 + 2^{i-1})$ events are found in a window of size $2^{i+1}$, then a detailed search must be performed to check if some subwindow of size $w$, $2 + 2^{i-1} \leq w \leq 1 + 2^i$, has $f(w)$ events. Thus, any burst in a window of size $w$ will be found.

When bursts are very rare, detailed searches whether fruitful (they confirm a burst) or not will also be rare, so the structure will incur an amortized constant time work per time point[18]. However,

- When bursts are rare but not very rare, the number of fruitless detailed searches grows, suggesting that we may want more levels than offered in the Shifted Binary Tree.
- Conversely, when bursts are exceedingly rare we may need fewer levels than offered in the SBT.

In other words we want a specific structure that fits a specific problem.

In this paper, we present a generalized framework [45] for efficient elastic burst detection, which includes a family of data structures, called *shifted aggregation trees (SAT)*, and a heuristic algorithm to find an efficient SAT given the inputs. Experiments show the SAT significantly outperforms the SBT over a variety of inputs.

Table 1: Comparing SAT with SBT

| | SBT | SAT |
|---|---|---|
| Number of children | 2 | $\geq 2$ |
| Levels of children for level $i+1$ | $i$ | $\leq i$ |
| Shift at level $i+1$: $S_{i+1}$ | $2 * S_i$ | $k * S_i$ $k \geq 1$ |
| Overlapping window size at level $i+1$: $O_{i+1}$ | window size at level $i$: $w_i$ | $\geq w_i$ |

### 5.3 Shifted Aggregation Tree (SAT)

In [45], we generalized the Shifted Binary Tree to a family of data structures, *Shifted Aggregation Tree (SAT)*, which provides a pool of data structures to choose for different inputs.

Like a Shifted Binary Tree, a Shifted Aggregation Tree (SAT) is a hierarchical tree structure. It has several levels, each of which contains several nodes. The nodes at level 0 are in one-to-one correspondence with the original time series. Any node at level $i$ is computed by aggregating some nodes below level $i$. Two consecutive nodes in the same level overlap in time.

A SAT is different from a SBT in two ways:

- The parent-child structure
  This defines the topological relationship between a node and its children, i.e. how many children it has and their placements.
- The shifting pattern
  This defines how many time points apart two neighboring nodes at the same level are. We called this distance the *shift*.

In a SBT, the parent-child structure for each node is always the same, one node aggregates two nodes at one level lower; the shifting pattern is also fixed, two neighboring nodes in the same level always half-overlap. In a SAT, a node could have 3 children and be 2 time points away from its preceding neighbor, or could have 64 children and 128 time points away from its preceding one. Table 5.3 gives a side-by-side comparison of the difference between a SAT and a SBT. Clearly, a SBT is a special case of a SAT.

### 5.4 SAT Tradeoffs

A similar update-search framework can be used together with a Shifted Aggregation Tree to detect bursts. The total running time is the sum of the updating time and the detailed searching time. Intuitively, if a SAT has more levels and smaller shifts, it would take longer time to maintain the structure. On the other hand, more levels and smaller shifts reduce the chance to trigger a fruitless detailed search and reduce the time for such a search. Therefore a good SAT should balance the updating time with the detailed searching time to obtain the optimal performance. Depending on different inputs, different SAT structures should be used to achieve better running time.

### 5.5 Heuristic State-Space Algorithm to Find an Efficient SAT

Given the input time series and the window thresholds, we can use a heuristic state-space algorithm to find an efficient SAT structure. Each SAT is seen as a state and the growth from one SAT to another is seen as a transformation. We start from the SAT only containing the original time series, then keep growing the candidate set of SATs by adding one more level to the top of each candidate SAT. A cost is associated with each SAT, which is defined as the CPU time when running this SAT on a small sample data. This growing process stops when a set of final SATs covering the maximum window size of interest are reached. The final SAT with the minimum cost is picked as the desired SAT.

### 5.6 Empirical Results

We have used two real world data sets to test the new detection framework with the Shifted Aggregation Trees. The testing machine is 2Ghz Pentium 4 PC with 512M RAM, running Windows XP. The program is implemented in C++.

- The Sloan Digital Sky Survey (SDSS) Weblog data
  This data set records the web access requests to the SDSS website in 2003, total 17,432,468 records. The training data for the heuristic state-space algorithm consists of seven days of second-by-second data.

- The NYSE TAQ Stock Data
  This data set includes tick-by-tick trading activities of the IBM stock between Jan. 2001 to May 2004, total 6,134,362 ticks. Each record contains the time precise to the second, as well as each trade's price and volume. A week's (5 day) worth of data is used as the training data.

In the experiments, we set the thresholds for different window sizes as following. We used a week of the SDSS weblog data and the IBM stock data as the training data respectively. For each window size $w$, we compute the aggregates on the training data on a sliding window of size $w$. We set the threshold for size $w$ in such a way, the probability for a burst of size $w$ to happen is $p$.

We are interested in comparing the Shifted Aggregation Tree with the Shifted Binary Tree under different settings.

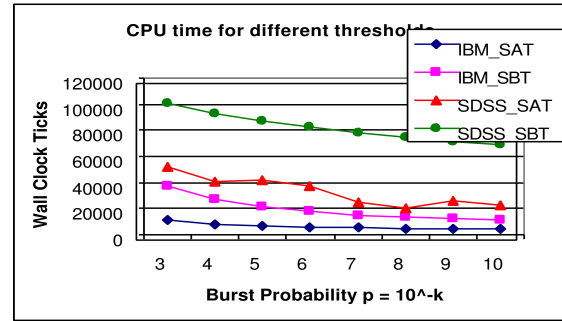



Figure 13: CPU time comparison under different thresholds

- Different thresholds
  The thresholds are set to reflect a burst probability ranging from $10^{-3}$ to $10^{-10}$. The maximum window size is set to 300 for SDSS, 500 for IBM. Bursts at every window size are detected.

  Figure 13 shows the results for both data sets. As the burst probability decreases, the CPU time for the SAT decreases quickly.

- Different maximum window sizes of interest
  The maximum window sizes are set from 10 seconds up to 1800 seconds. The burst probability is set to be $10^{-6}$. Bursts in every window size are detected.

  Figure 14 shows the results. As the maximum window size increases, the SAT can achieve increased speedup comparing to the SBT.

- Different sets of window sizes of interest
  Instead of detecting bursts for every window size, we detect bursts every $n$ size, i.e, detect bursts for window sizes $n, 2{*}n, 3{*}n, .....$ Here, $n$ is set to be 1, 5, 10, 30, 60, 120 respectively. The burst probability is set to be $10^{-6}$ and the maximum window size is set to be 600 for SDSS, 3600 for IBM.

  Figure 15 shows that as the set of window sizes becomes sparser, both the SAT and the SBT take less time to process.

Overall, the new framework, which includes a family of Shifted Aggregation Trees and a heuristic state-space algorithm, can be adaptive to different inputs. The Shifted Aggregation Tree overperforms the Shifted Binary Tree over a variety of inputs.

## 6 Summary and Future Work

High performance time series algorithms are needed in many previous applications. In this paper four incremental techniques are presented addressing fast windowed correlation, incremental matching pursuit, sparse null space discovery and elastic burst detection. The experimental results show that they can improve the efficiency dramatically compared to the previous algorithms. Furthermore the incremental idea described in this paper may be extended to other primitives.

The work demonstrated here is very much in progress. Our goal is to build a set of such primitives and make

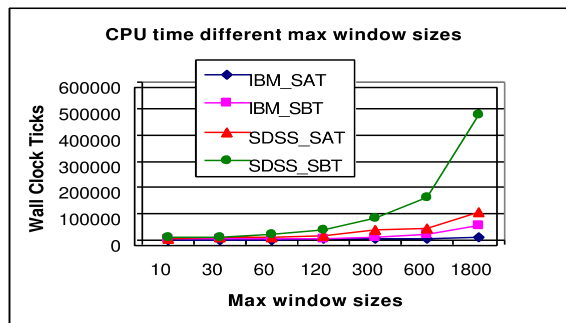



Figure 14: CPU time comparison under different maximum window sizes of interest

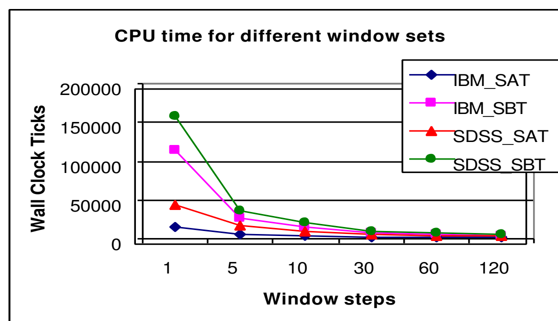



Figure 15: CPU time comparison under different sets of window sizes of interest

them available to any application needing rapid insights from time series.

## References

[1] A. Braverman, *Personal Communication*, 2003.

[2] R. Agrawal, C. Faloutsos, and A. Swami, "Efficient similarity searching in sequence databases," in *Proceedings of the 4th International Conference of Foundations of Data organization and Algorithms (FODO)*, Chicago, Illinois, MN, 1993, pp. 69–84, Springer Verlag.

[3] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos, "Fast subsequence matching in time-series databases," in *ACM SIGMOD*, Minneapolis, MN, May 1994, pp. 419–429.

[4] C. Li, P. Yu, and V. Castelli, "Hierarchyscan: A hierarchical similarity search algorithm for databases of long sequences," in *IEEE ICDE*, New Orleans, Louisiana, February 1996, pp. 546–553.

[5] D. Rafier and A. Mendelzon, "Similarity-based queries for time series data," in *ACM SIGMOD*, Tucson, Arizona, May 1997, pp. 13–25.

[6] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss, "Surfing wavelets on streams: One-pass summaries for approximate aggregate queries," in *VLDB*, 2001, pp. 79–88.

[7] I. Popivanov and R. Miller, "Similarity search over time series data using wavelets," in *IEEE ICDE*, San Jose, CA, March 2002, pp. 212–225.

[8] F. Korn, H.V. Jagadish, and C. Faloutsos, "Efficiently supporting ad hoc queries in large datasets of time sequences," in *ACM SIGMOD*, Tucson, Arizona, May 1997, pp. 289–300.

[9] E. Keogh, K. Chakrabarti, M. Pazzani, and S. Mehrotra, "Dimensionality reduction for fast similarity search in large time series databases," *Knowledge and Information Systems*, vol. 3, pp. 263–286, 2000.

[10] E. Keogh, K. Chakrabarti, S. Mehrotra, and M. Pazzani, "Locally adaptive dimensionality reduction for indexing large time series databases," in *ACM SIGMOD*, Santa Barbara, California, May 2001, pp. 151–162.

[11] T. Palpanas, M. Vlachos, E. Keogh, D. Gunopulos, and W. Truppel, "Online amnesic approximation of streaming time series," in *IEEE ICDE*, Boston, MA, March 2004, pp. 338–350.

[12] B. Yi and C. Faloutsos, "Fast time sequence indexing for arbitrary lp forms," in *VLDB*, Cairo, Egypt, September 2000, pp. 385–394.

[13] W. Johnson and J. Lindenstrauss, "Extensions of lipschitz mapping into hilbert space," *Contemporary Mathematics*, vol. 26, pp. 189–206, 1984.

[14] E. Kushikvitz, R. Ostrovsky, and Y. Ranbani, "Efficient search for approximate nearest neighbors in high dimensional spaces," in *ACM STOC*, Dallas, TX, May 1998, pp. 614–623.

[15] P. Indyk, "Stable distributions, pseudorandom generators, embeddings and data stream computation," in *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*. 2000, pp. 189–197, IEEE Computer Society.

[16] D. Achlioptas, "Database-friendly random projections," in *ACM SIGMOD*, Santa Barbara, California, May 2001, pp. 274–281.

[17] Y. Zhu and D. Shasha, "Statstream: Statistical monitoring of thousands of data streams in real time," in *VLDB*, Hong Kong, China, August 2002, pp. 358–369.

[18] D. Shasha and Y. Zhu, *High Performance Discovery in Time Series: Techniques and Case Studies*, Springer, 2003.

[19] P. Indyk, N. Koudas, and S. Muthukrishnan, "Identifying representative trends in massive time series data sets using sketches," in *VLDB*, Cairo, Egypt, September 2000, pp. 363–372.

[20] R. Cole, D. Shasha, and X. Zhao, "Fast window correlations over uncooperative time series," in *Submitted to ACM SIGKDD*, Chicago, IL, USA, August 2005.

[21] E. Keogh and T. Folias, "The ucr time series data mining archive. Riverside CA. University of California - Computer Science & Engineering Department," 2002, http://www.cs.ucr.edu/∼eamonn/TSDMA/index.html.

[22] "Wharton research data services(wrds)," http://wrds.wharton.upenn.edu/.

[23] S. Mallat and Z. Zhang, "Matching pursuit with time-frequency dictionary," *IEEE Transactions on Signal Processing*, vol. 12, no. 41, pp. 3397–3415, 1993.

[24] O. D. Escoda and P. Vandergheynst, "Video coding using a deformation compensation algorithm based on adaptive matching pursuit image decompositions," in *IEEE ICIP*, Barcelona, Spain, September 2003, pp. 77–80.

[25] Y. Wu and V. S. Batista, "Quantum tunneling dynamics in multidimensional systems: A matching-pursuit description," *The Journal of Chemical Physics*, vol. 121, pp. 1676–1680, 2004.

[26] L. Borcea, J. G. Berryman, and G. C. Papanicolaou, "Matching pursuit for imaging high-contrast conductivity," *Inverse Problems*, vol. 15, pp. 811–849, 1999.

[27] P.J. Durka and K.J. Blinowska, "Analysis of eeg transients by means of matching pursuit," *Ann.Biomed.Engin.*, vol. 23, pp. 608–611, 1995.

[28] K.J. Blinowska, P.J. Durka, and W. Szelenberger, "Time-frequency analysis of nonstationary eeg by matching pursuit," World Congress of Medical Physics and Biomedical Engineering, August 1994.

[29] R. Gribonval, "Fast matching pursuit with a multiscale dictionary of gaussian chirps," *IEEE Transaction on Signal Processing*, vol. 49, no. 5, pp. 994–1001, MAY 2001.

[30] F. Moschetti, L. Granai, P. Vandergheynst, and P. Frossard, "New dictionary and fast atom searching method for matching pursuit representation of displaced frame difference," in *IEEE ICIP*, Rochester, NY, September 2002, pp. 685–688.

[31] K. Cheung and Y. Chan, "A fast two-stage algorithm for realizing matching pursuit," in *IEEE ICIP*, Thessaloniki Greece, October 2001, pp. 431–434.

[32] A.J. Hoffman and S.T. McCormick, "A fast algorithm that makes matrices optimally sparse," in *Progress in Combinatorial Optimization*, William R. Pulleyblank, Ed., pp. 185–196. Academic Press, 1984.

[33] A. Topcu, *A contribution to the systematic analysis of finite element structures through the force method*, Ph.D. thesis, University of Essen, Essen, Germany, 1979, (In German).

[34] T.F. Coleman and A. Pothen, "The null space problem I. complexity," *SIAM Journal on Algebraic and Discrete Methods*, vol. 7, no. 4, pp. 527–537, Oct. 1986.

[35] J.R. Gilbert and M.T. Heath, "Computing a sparse basis for the null space," *SIAM Journal on Algebraic and Discrete Methods*, vol. 8, no. 3, pp. 446–459, July 1987.

[36] T.F. Coleman and A. Pothen, "The null space problem II. algorithms," *SIAM Journal on Algebraic and Discrete Methods*, vol. 8, no. 4, pp. 544–563, Oct. 1987.

[37] M. Wang, T. Madhyastha, N. Chan, S. Papadimitriou, and C. Faloutos, "Data mining meets performance evaluation: Fast algorithms for modeling bursty traffic," in *IEEE ICDE*, San Jose, CA, March 2002, pp. 507–516.

[38] J. Kleinberg, "Bursty and hierarchical structure in streams," in *ACM SIGKDD*, Edmonton, Alberta, Canada, July 2002, pp. 91–101.

[39] M. Vlachos, C. Meek, and Z. Vagena, "Identifying similarity and periodicities and bursts for online search queries," in *ACM SIGMOD*, Paris, France, June 2004, pp. 131–142.

[40] Daniel B. Neill and Andrew W. Moore, "A fast multi-resolution method for detection of significant spatial disease clusters," in *Advances in Neural Information Processing Systems 16*, Sebastian Thrun, Lawrence Saul, and Bernhard Schölkopf, Eds., Cambridge, MA, 2004, pp. 651–658, MIT Press.

[41] D. Neill and A. Moore, "Rapid detection of significant spatial clusters," in *ACM SIGKDD*, Seattle, WA, August 2004, pp. 256–265.

[42] Daniel B. Neill, Andrew W. Moore, Francisco Pereira, and Tom Mitchell, "Detecting significant multidimensional spatial clusters," in *Advances in Neural Information Processing Systems 17*, Lawrence K. Saul, Yair Weiss, and Léon Bottou, Eds., Cambridge, MA, 2005, pp. 969–976, MIT Press.

[43] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani, "Maintaining stream statistics over sliding windows," *SIAM*, vol. 31, no. 6, pp. 1794–1813, September 2002.

[44] Phillip B. Gibbons and Srikanta Tirthapura, "Distributed stream algorithms for sliding windows," in *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, 2002, pp. 63–72.

[45] X. Zhang, "High performance burst detection," *Thesis Proposal*, 2005.