A Unified Framework for Monitoring Data Streams in Real Time *

Ahmet Bulut and Ambuj K. Singh
Department of Computer Science
UC Santa Barbara
Santa Barbara, CA 93106-5110
{bulut,ambuj}@cs.ucsb.edu

Abstract

Online monitoring of data streams poses a challenge in many data-centric applications, such as telecommunications networks, traffic management, trend-related analysis, web-click streams, intrusion detection, and sensor networks. Mining techniques employed in these applications have to be efficient in terms of space usage and per-item processing time while providing a high quality of answers to (1) aggregate monitoring queries, such as finding surprising levels of a data stream, detecting bursts, and to (2) similarity queries, such as detecting correlations and finding interesting patterns. The most important aspect of these tasks is their need for flexible query lengths, i.e., it is difficult to set the appropriate lengths a priori. For example, bursts of events can occur at variable temporal modalities from hours to days to weeks. Correlated trends can occur at various temporal scales. The system has to discover "interesting" behavior online and monitor over flexible window sizes. In this paper, we propose a multi-resolution indexing scheme, which handles variable length queries efficiently. We demonstrate the effectiveness of our framework over existing techniques through an extensive set of experiments.

1. Introduction

A growing number of real-world applications deal with multiple streams of data: performance measurements in network monitoring and traffic management, call detail records in telecommunications networks, transactions in retail chains, ATM operations in banks, log records generated by web servers, and sensor network data. Working with streams of data is like drinking from the proverbial fire hose: the volume is simply overwhelming. For example, in telecommunications network monitoring, a tremendous number of connections are handled every minute by switches. As collected, stream data is almost always at low level and too large to maintain in main memory. Instead, one can maintain a synopsis of the stream at certain abstraction levels on the fly. The synopsis is a small space data structure, and can be updated incrementally as new values stream in. It is used to discover interesting behavior, which may prompt an in-depth analysis on the data at lower levels of abstraction [6].

In astrophysics, the sky is constantly observed for high-energy particles. When a particular astrophysical event happens, a shower of high-energy particles arrives in addition to the background noise. This yields an unusually high number of detectable events (high-energy photons) over a certain time period, which indicates the existence of a *Gamma Ray Burst*. If we know the duration of the shower, we can maintain a count on the total number of events over sliding windows of the known window size and raise an alarm when the moving sum is above a threshold. Unfortunately, in many cases, we cannot predict the duration of the burst period. The burst of high-energy photons might last for a few milliseconds, a few hours, or even a few days. Therefore, it is essential to design a scheme, which allows monitoring over a variable timescale [21].

In finance, finding similar patterns in a time series database is a well studied problem [7]. The features of a time series sequence are extracted using a sliding window, and inserted into an index structure for query efficiency. However, this approach is not adequate for data stream applications, since it requires a time consuming feature extraction step with each incoming data item. For this purpose, incremental computation techniques that use the previous feature in computing the new feature have been proposed to accelerate per-item processing [20]. A batch technique can further decrease the per-item processing cost by computing a new feature periodically instead of every time unit [14]. However, most of these existing techniques work well for queries of a fixed length, and do not consider queries of variable length. For example, a user might want to know all time periods during which the movement of a particular stock follows a certain interesting trend that can be generated automatically by the application [18]. Therefore, one may not have *a priori* knowledge regarding the query length. In order to ad-

*Work supported partially by NSF under grants EIA-0080134 and ANI-0123985.

dress this issue, a multi resolution indexing scheme has been proposed [9]. However, this work restricts itself to the off-line databases and does not consider how well the proposed scheme works in a streaming scenario. A unified framework, which inherits from these existing solutions, is necessary in order to extract features over data streams incrementally and handle variable length queries efficiently.

Continuous queries that run indefinitely, unless a lifetime has been specified, fit naturally into the mold of data stream applications. Examples of these queries include monitoring a set of conditions or events to occur, detecting a certain trend in the underlying raw data, or in general discovering relations between various components of a large real time system. We identified three different kinds of queries that are of interest from an application point of view: (1) monitoring aggregates, (2) monitoring or finding patterns, and (3) detecting correlations. This list is by no means complete, but covers a large number of real world applications that are described above. A common aspect of these queries is that each of them requires data management over some history of values, and not just over the most recently reported values [5]. For example in case of aggregate queries, the system monitors whether the current window aggregate deviates significantly from that aggregate in most time periods of the same size. In case of correlation queries, the self-similar nature of sensor measurements may be reflected as temporal correlations at some resolution over the history of the stream [16]. Therefore, the system has to maintain historical data along with the current data in order to be able to answer these queries.

The types of queries we address within the scope of this paper have been addressed separately in the database research community. However, we envision that all these queries are interconnected in a monitoring infrastructure. For example, an unusual volatility of a time series may trigger an in-depth trend analysis to discover the concepts hidden in the data. One can devise efficient schemes for each of these tasks separately. To the best of our knowledge however; a general scheme that accommodates all these tasks in a single body has not been addressed. We try to fill this gap by proposing a unified system solution called "Stardust" that realizes this vision.

1.1. Our contribution

The core part of our system involves feature extraction over data streams. The features are extracted at multiple resolutions in order to handle queries of variable length efficiently. A dynamic index structure is used to index these features for query efficiency. We summarize our algorithmic contributions, which mainly address the maintenance issues in the system, as follows:

- The features at higher resolutions are computed using the features at lower resolutions; therefore all features are computed in a single pass.
- The system guarantees the accuracy provided to the user queries by provable error bounds.
- The index structure has tunable parameters to trade accuracy for speed and space. The per-item processing cost and the space overhead can be tuned according to the specific application requirements by varying the update rate and the number of coefficients maintained in the index structure.

2. Data and Query Models

2.1. Stream computation model

A data stream consists of an ordered sequence of data points $\ldots, x[i], \ldots$ such that the value of each data point $x[i]$ lies in a bounded range, i.e., $x[i] \in [R_{min}, R_{max}]$. We assume $R_{min} = 0$ unless otherwise stated. We consider a system that has M input streams, and that maintains summary information over a time window of size N for each stream.

In the rest of the paper, we use $x[i]$ to refer to the i-th entry of stream x, and $x[i_1 : i_2]$ to refer to the subsequence of entries at positions i_1 through i_2.

2.2. Aggregate monitoring queries

In this class of queries, aggregates of data streams are monitored over a set of time intervals [21]: "Report all occurrences of Gamma Ray bursts from a timescale of minutes to a timescale of days". Formally, given a window size w, an aggregate function F, and a threshold τ associated with the window, the goal is to report all those time instances such that the aggregate applied to the subsequence $x[t-w+1 : t]$ exceeds the corresponding window threshold, i.e., check if

$$F(x[t-w+1:t]) \geq \tau \quad (1)$$

where t denotes the current time. The thresholds can be estimated from historical data or a model of the stream.

2.3. Pattern monitoring queries

In this class of queries, a pattern database is continuously monitored over dynamic data streams: "Identify all temperature sensors in a weather monitoring sensornet that currently exhibit an interesting trend". Formally, given a query sequence Q and a threshold value r, find the set of streams that are within distance r to the query sequence Q. The distance measure we adopt is the Euclidean distance (L_2) between the corresponding normalized sequences. We normalize a window of values $x[1], \ldots, x[w]$ as follows:

$$\hat{x}[i] = \frac{x[i]}{\sqrt{w} * R_{max}} \qquad i = 1, \ldots, w \quad (2)$$

thereby mapping it to the unit hyper-sphere. This specific normalization is for the purpose of experimental consistency with previous research work [9].

2.4. Correlation monitoring queries

In this class of queries, all stream pairs that are correlated within a user specified threshold r at some level of abstraction are reported continuously. The correlation between two sequences x and y can be reduced to the Euclidean distance between their z-norms [20]. The z-norm of a sequence $x[1], \ldots, x[w]$ is defined as follows:

$$\hat{x}[i] = \frac{x[i] - \mu_x}{\sqrt{\sum_{i=1}^{w}(x[i] - \mu_x)^2}} \qquad i = 1, \ldots, w \quad (3)$$

where μ_x is the arithmetic mean. The correlation coefficient between sequences x and y is computed using the L_2 distance between $\hat{x}$ and $\hat{y}$ as $1 - L_2^2(\hat{x}, \hat{y})/2$.

3. Related Work

Zhu and Shasha consider burst detection using a summary structure called SWT for Shifted-Wavelet Tree [21]. For a given set of query windows w_1,w_2,...,w_m such that $2^L W \leq w_1 \leq w_2 \leq \ldots \leq w_m \leq 2^U W$, SWT maintains $U - L$ moving aggregates using a wavelet tree for incremental computation. A window w_i is monitored by the lowest level j, $L \leq j \leq U$, that satisfies $w_i \leq 2^j W$. Therefore, associated with each level j, $L \leq j \leq U$, there is a threshold τ_j equal to the smallest of the thresholds of windows $w_{i_1}, \ldots, w_{i_j}$ monitored by that level. Whenever the moving sum at some level j exceeds the level threshold τ_j, all query windows associated with this level are checked using a brute force approach.

Kahveci and Singh proposed MR-Index for answering variable length queries over time series data [9]. Wavelets are used to extract features from a time series at multiple resolutions. At each resolution, a set of feature vectors are combined into an MBR and stored sequentially in the order they are computed. A given query is decomposed into multiple sub-queries such that each sub-query has resolution corresponding to a resolution at the index. A given set of candidate MBRs are refined using each query as a filter to prune out non-potential candidates. However, the authors consider and experiment on a time-series database available off-line, where per-item processing time is not an issue. In a streaming scenario, computing a transformation at each data arrival for each resolution at the index is very costly.

Keogh et al. [10] propose a version of piecewise constant approximation for time series data. The so-called *Adaptive Piecewise Constant Approximation* (APCA) represent regions of great fluctuations with several short segments, while regions of less fluctuations are represented with fewer, long segments. Very recently, they have extended their approximation to allow error specification for each point in time [15]. The resulting approach can approximate data with fidelity proportional to its age. Moon et al. [14] proposed GeneralMatch, a refreshingly new idea in similarity matching. It divides the data sequences into disjoint windows, and the query sequence into sliding windows. This approach is the dual of the conventional approaches, i.e., dividing the data sequence into sliding windows, and the query sequence into disjoint windows. The overall framework is based on answering pattern queries using a single-resolution index built on a specific choice of window size. The allowed window size depends on the minimum query length, which has to be provided a-priori before the index construction.

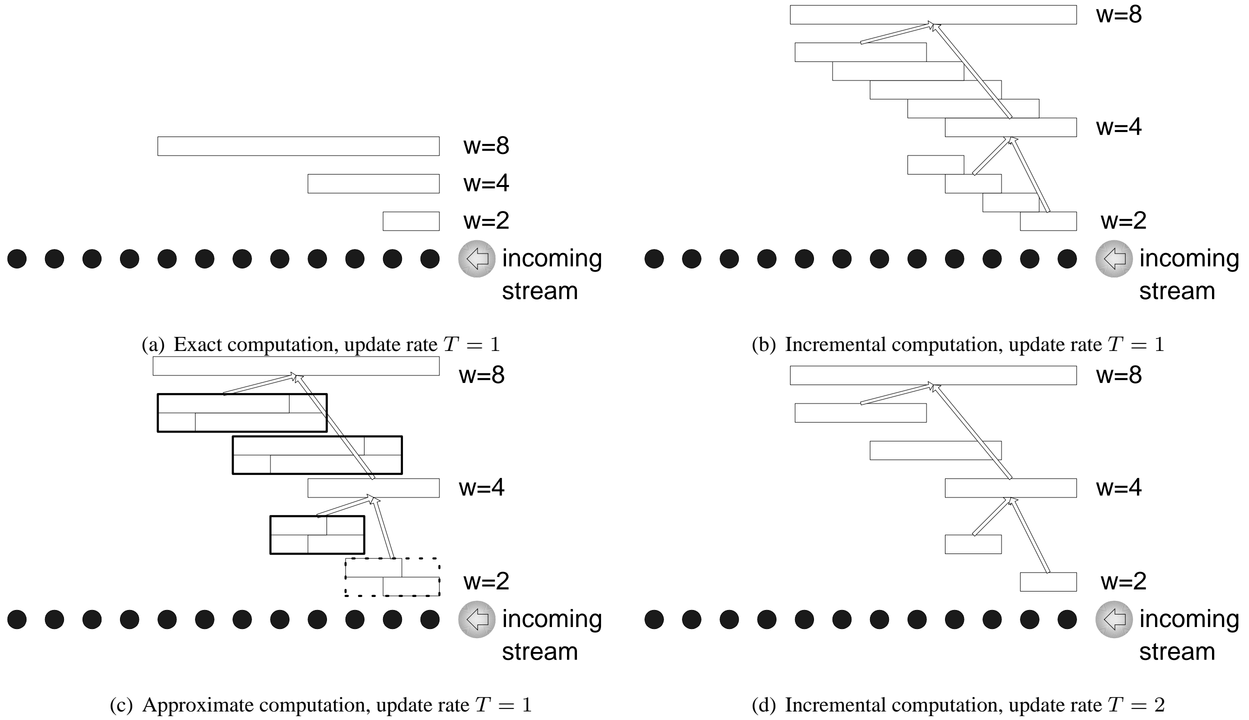
Yi et al. [19] consider a method to analyze co-evolving time sequences. They model the problem as a multi-variate linear regression. For a given stream S, they try to estimate its current value (dependent variable) as a linear combination of values of the same and other streams (independent variables) under sliding window model. Given v independent variables, and a dependent variable y with N samples each, they find the best b independent variables to compute the current value of the dependent variable in $O(Nbv^2)$ time.

Zhu and Shasha proposed StatStream for monitoring a large number of streams in real time [20]. It subdivides the history of a stream into a fixed number of basic windows and maintains Discrete Fourier Transform (DFT) coefficients for each basic window. This allows a batch update of DFT coefficients over the entire history. It superimposes an orthogonal regular grid on the feature space, and partitions the space into cells of diameter r, the correlation threshold. Each stream is mapped to a number of cells (exactly how many depends on the "lag time") in the feature space based on a subset of its DFT coefficients. It uses proximity in this feature space to report correlations [3].

In our earlier work SWAT [4], we examined how to summarize a single data stream in an online manner. The technique uses $O(\log N)$ space for a stream of size N. The update cost for each incoming data point is constant, which follows from an amortized analysis. For an inner product query, the technique computes an approximate answer in polylogarithmic time. However, this work focuses on distributed monitoring of a single stream rather than answering queries over multiple data streams, and does not fully address data management over a history of values.

4. Our Solution

In this section, we introduce our techniques to extract features at multiple resolutions over data streams. We adopt a multi-resolution approach since we assume that we do not have *a priori* information regarding queries except the minimum query length. The features at a specific resolution are obtained with a sliding window of a fixed length w. The sliding window size doubles as we go up a resolution, i.e., a level. In the rest of the paper, we will use the terms "level" and "resolution" interchangeably. We denote a newly computed feature at resolution i as $\mathcal{F}_i$. Figure 1(a) shows an example where we have three resolutions with corresponding sliding window sizes of 2, 4 and 8. With each arrival of a new stream value, we can compute features $\mathcal{F}_0$, $\mathcal{F}_1$, and $\mathcal{F}_2$, one for each resolution. However, this requires maintaining



(a) Exact computation, update rate $T = 1$

(b) Incremental computation, update rate $T = 1$

(c) Approximate computation, update rate $T = 1$

(d) Incremental computation, update rate $T = 2$

Figure 1. Different feature extraction schemes at multiple resolutions over a data stream.

all the stream values within a time window equal to the size of the largest sliding window, i.e., 8 in our running example. The per-item processing cost and the space required is linear in the size of the largest window [9].

For a given window w of values $y = x[t-w+1], \ldots, x[t]$, we use an "incremental" transformation $F(y)$ to compute features. The type of transformation F we use depends on the monitoring query. For example, we use SUM for burst detection, MAX$-$MIN for volatility detection, and Discrete Wavelet Transformation (DWT) for detecting correlations and finding surprising patterns. For most real time series, the first f ($f << w$) DWT coefficients retain most of the energy of the signal. Therefore, we can safely disregard all but the very first few coefficients to retain the salient features (e.g., the overall trend) of the original signal.

Using an incremental transformation leads to a more efficient way of computing features at all resolutions. We can compute level-1 features using level-0 features, and level-2 features using level-1 features. In general, we can use lower level features to compute higher level features [1]. Figure 1(b) depicts this new way of computation. This new algorithm has a lower per-item processing cost, since we can compute $\mathcal{F}_1$ and $\mathcal{F}_2$ in constant time. The following lemma establishes this result.

Lemma 4.1 *The new feature $\mathcal{F}_j$ at level j for the subsequence $x[t-w+1:t]$ can be computed "exactly" using the features $\mathcal{F}'_{j-1}$ and $\mathcal{F}_{j-1}$ at level $j-1$ for the subsequences $x[t-w+1:t-w/2]$ and $x[t-w/2+1:t]$ respectively.*

Proof $\mathcal{F}_j$ is $max(\mathcal{F}'_{j-1}, \mathcal{F}_{j-1})$, $min(\mathcal{F}'_{j-1}, \mathcal{F}_{j-1})$, $\mathcal{F}'_{j-1} + \mathcal{F}_{j-1}$ for MAX, MIN, and SUM respectively. For DWT, See Lemma A.1 in Appendix A. ∎

However, the space required for this scheme is also linear in the size of the largest window. The reason is that we need to maintain half of the features at the lower level to compute the feature at the upper level incrementally. If we can trade accuracy for space, then we can decrease the space overhead by computing features approximately. At each resolution level, we combine every c of the feature vectors into a box, or a minimum bounding rectangle (MBR). Figure 1(c) depicts this scheme for $c = 2$. Since each MBR B contains c features, it has an extent along each dimension. In case of SUM, $B[1]$ corresponds to the smallest sum, and $B[2]$ corresponds to the largest sum among all c sums. In general, $B[2i]$ denotes the low coordinate and $B[2i+1]$ denotes the high coordinate along the i-th dimension. Note that for SUM, MAX and MIN, B has a single dimension. However, for DWT the number of dimensions f is application dependent.

This new approach decreases the space overhead by a factor of c. Since we use the extent information of the MBRs in the computation, the newly computed feature will also be an extent. The following lemma proves this result.

Lemma 4.2 *The new feature $\mathcal{F}_j$ at level j can be computed "approximately" using the MBRs B_1 and B_2 that contain the features $\mathcal{F}'_{j-1}$ and $\mathcal{F}_{j-1}$ at level $j-1$ respectively.*

Proof

$$max(B_1[1], B_2[1]) \leq \mathcal{F}_j \leq max(B_1[2], B_2[2])$$
$$min(B_1[1], B_2[1]) \leq \mathcal{F}_j \leq min(B_1[2], B_2[2])$$
$$B_1[1] + B_2[1] \leq \mathcal{F}_j \leq B_1[2] + B_2[2]$$

See Lemma A.2 in Appendix A

for MAX, MIN, SUM and DWT respectively. ∎

Using MBRs instead of individual features exploits the fact that there is a strong spatio-temporal correlation between the consecutive features. Therefore, it is natural to extend the computation scheme to eliminate this redundancy. Instead of computing a new feature at each data arrival, one can employ a batch computation such that a new feature is computed periodically, at every T time unit. This allows us to maintain features instead of MBRs. Figure 1(d) shows this scheme with $T = 2$. The new scheme has a clear advantage in terms of accuracy; however it can dismiss potentially interesting events that may occur between the periods.

Depending on the box capacity and the update rate T_j at a given level j (the rate at which we compute a new feature) we have the following two general computation algorithms:

- **Online algorithm:** Update rate T_j is equal to 1. The box capacity c is variable. We use this update rate for aggregate monitoring queries.
- **Batch algorithm:** For $T_j > 1$, we have a batch algorithm. The box capacity is set to $c = 1$. We use T_j=W for finding surprising patterns and detecting correlations. This specific choice of parameter settings is for complying with experiments in previous research work. Our earlier work SWAT [4] is a batch algorithm with $T_j = 2^j$.

The following theorem establishes the time and space complexity of a given algorithm in terms of c and T_j. We assume that W denotes the sliding window size at the lowest resolution, J denotes the highest resolution, and f denotes the dimensionality of $\mathcal{F}$.

Theorem 4.3 *The new feature $\mathcal{F}_j$ at level j for a stream can be computed incrementally in time $\Theta(f)$ and in space $\Theta(2^{j-1}W/cT_{j-1})$.*

Proof We compute $\mathcal{F}_j$ at level j using the features at level $j-1$ in time $\Theta(f)$ as shown in Lemmas 4.1 and 4.2. The number of features that need to be maintained at level $j-1$ for incremental computation at level j is $2^{j-1}W$. Therefore, depending on the box capacity and update rate, the space complexity at level $j-1$ is $\Theta(2^{j-1}W/cT_{j-1})$. ∎

As new values stream in, new features are computed and inserted into the corresponding index structures while features that are out of history of interest are deleted to save space. Coefficients are computed at multiple resolutions starting from level 0 up to level J: at each level a sliding window is used to extract the appropriate features. Computation of features at higher levels is accelerated using the MBRs at lower levels. The MBRs belonging to a specific stream are threaded together in order to provide a sequential access to the summary information about the stream. This approach results in a constant retrieval time of the MBRs. The complete algorithm is shown in Algorithm 1.

Algorithm 1 Compute_Coefficients(Stream S)

Require: $B^S_{j,i}$ denotes the i-th MBR at level j for stream S.
begin procedure
  $w := W$ (the window size at the lowest resolution);
  t_{now} denotes the current discrete time;
  for $j := 0$ to J **do**
    $B^S_{j,i}$:= the current MBR at level j for stream S;
    if $j = 0$ **then**
      $y := S[t_{\text{now}} - w + 1 : t_{\text{now}}]$;
      normalize y if $F = DWT$;
      $\mathcal{F}_j := F(y)$;
    else
      find MBR B^S_{j-1,i_1} that contains the feature
        for the subsequence $S[t_{\text{now}} - w + 1 : t_{\text{now}} - \frac{w}{2}]$;
      find MBR B^S_{j-1,i_2} that contains the feature
        for the subsequence $S[t_{\text{now}} - \frac{w}{2} + 1 : t_{\text{now}}]$;
      $\mathcal{F}_j := F(B^S_{j-1,i_1}, B^S_{j-1,i_2})$;
    end if
    if number of features in $B^S_{j,i} < c$ (box capacity) **then**
      insert $\mathcal{F}_j$ into $B^S_{j,i}$;
    else
      insert $B^S_{j,i}$ into index at level j;
      start a new MBR $B^S_{j,i+1}$;
      insert $\mathcal{F}_j$ into $B^S_{j,i+1}$;
    end if
    adjust the sliding window size to $w := w * 2$;
  end for
end procedure

We maintain features at a given level in a high dimensional index structure. The index combines information from all the streams, and provides a scalable access medium for answering queries over multiple data streams. However, each MBR inserted into the index is specific to a single stream. We use the R*-Tree family of index structures for indexing MBRs at each level [2]. In order to support frequent updates, the techniques outlined in [12] can be employed to accelerate our implementation.

5. Advanced Search Algorithms

In this section, we describe what and how to monitor in our framework, namely aggregate monitoring queries, pattern monitoring queries, and correlation monitoring queries.

5.1. Monitoring an aggregate query

Without loss of generality, we assume that the query window size is a multiple of W. An aggregate query with window size w and threshold τ is answered by first partitioning the window into multiple sub-windows, w_1,w_2,. . .,w_n such that $0 \leq j_1 < \ldots < j_i < j_{i+1} < \ldots < j_n \leq J$, and $w_i = W2^{j_i}$. For a given window of length bW, the partitioning corresponds to the ones in the binary representation of b such that $\sum_{i=1}^{n} 2^{j_i} = b$. The current aggregate over a window of size w is computed using the sub-aggregates for sub-windows in the partitioning. Assume that $W = 2$ and $c = 2$. Consider a query window $w = 26$. The binary representation of $b = 13$ is 1101, and therefore the query is partitioned into three sub-windows $w_0 = 2$, $w_2 = 8$, and $w_3 = 16$. Figure 2 shows the decomposition of the query and the composition of the aggregate together. The current aggregate over a window of size $w = 26$ is approximated using the extents of MBRs that contain the corresponding sub-aggregates. The computation is approximate in the sense that the algorithm returns an interval $\mathcal{F}$ such that the upper coordinate $\mathcal{F}[2]$ is always greater than or equal to the true aggregate. If $\mathcal{F}[2]$ is larger than the threshold τ, we retrieve the most recent subsequence of length w, and compute the true aggregate. If this value exceeds the threshold, we raise an alarm. The complete algorithm is shown in Algorithm 2.

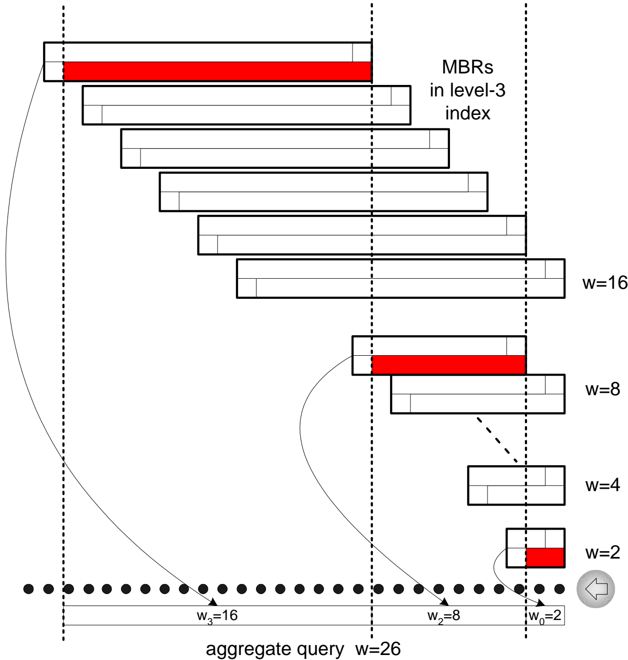


Figure 2. Aggregate query decomposition and approximation composition for a query window of size $w = 26$.

In order to quantify the false alarm rate of our approximation, we assume that we monitor bursts of events and that our aggregate function is SUM. Let X denote the sum within sliding window $w = bW$. We set τ to $\mu_X(1 - \Phi(p))$ such that

$$Pr\left(\frac{X - \mu_X}{\mu_X} \geq \frac{\tau - \mu_X}{\mu_X}\right) \leq p \tag{4}$$

holds for a given sufficiently small p, where Φ denotes the normal cumulative distribution function. Zhu and Shasha [21] address monitoring the burst based on windows with size Tw such that $1 \leq T < 2$, where $2^{j-1}W < w \leq 2^j W$. This approach corresponds to monitoring the burst via one of the levels in the index structure. Let Z denote the sum within sliding window Tw. We assume that

$$\frac{Z - \mu_Z}{\mu_Z} \sim \text{Norm}(0, 1) \tag{5}$$

Assuming $\mu_Z = T\mu(X)$, one can show that the false alarm rate is equal to $Pr(Z > \tau)$ such that

$$Pr\left(\frac{Z - T\mu(X)}{T\mu(X)} \geq \frac{\tau - T\mu(X)}{T\mu(X)}\right) = \Phi\left(1 - \frac{1 - \Phi^{-1}(p)}{T}\right) \tag{6}$$

According to Equation 6, for a fixed value of p, the smaller T is, the smaller is the false alarm rate. In our case, we use sub-aggregates for sub-windows w_1,w_2,. . .,w_n for computing the final aggregate on a given query window of size w and threshold τ. The sub-aggregate for sub-window w_i is stored in an MBR at level j_i. An MBR at level j_i corresponds to a monitoring window of size $2^{j_i}W + c - 1$. Then, effectively we monitor a burst using a window of size $bW + \log b * (c-1)$ such that:

$$T' = \frac{bW + \log b * (c-1)}{bW} = 1 + \frac{\log b * (c-1)}{bW} \tag{7}$$

where T' decreases with increasing b. For example, for $c = W = 64$ and $b = 12$, we have $T' = 1.2987$ and $T = 1.3333$.

Algorithm 2 Aggregate_Query(Stream S, Window w, Threshold τ)

begin procedure
  initialize t to t_{now}, the current discrete time;
  partition w into n parts as $w_1, w_2, \ldots, w_n$;
  initialize aggregate $\mathcal{F}$;
  for $i := 1$ to $i := n$ **do**
    find the resolution level j such that $w_i = W2^j$;
    MBR B contains the feature on $S[t - w_i + 1 : t]$;
    merge sub-aggregate B to $\mathcal{F} := F(B, \mathcal{F})$;
    adjust offset to $t := t - w_i$ for next sub-window;
  end for
  if $\tau \leq \mathcal{F}[2]$ **then**
    retrieve $S[t_{\text{now}} - w + 1 : t_{\text{now}}]$;
    if $\tau \leq F(S[t_{\text{now}} - w + 1 : t_{\text{now}}])$ **then**
      raise an alarm;
    end if
  end if
end procedure

This implies that our approximation reduces the false alarm rate to a minimal amount with the optimal being at $T' = 1$. In fact, for $c = 1$ we have the optimal algorithm. However the space consumption in this case is much larger.

5.2. Monitoring a pattern query

There are two different ways to answer a pattern query depending on the algorithm used for index construction. We first consider the case of the online algorithm. Given a query sequence Q and a threshold value r, we first partition it into multiple sub-queries, $Q_1, Q_2, \ldots, Q_n$ such that $0 \leq j_1 < \ldots < j_i < j_{i+1} < \ldots < j_n \leq J$, and $|Q_i| = W2^{j_i}$. Assume that the first sub-query Q_1 has resolution j_1. We perform a range query with radius r on the index constructed at resolution j_1. On the initial candidate box set R, we perform the hierarchical radius refinement technique proposed in [9]. Briefly, for each MBR $B \in R$, this technique is used to refine the original radius r to $r' = \sqrt{r^2 - d_{\min}(Q_1, B)^2}$ for the next sub-query Q_2, where $d_{\min}(p, B)$ for a point p and an MBR B is defined as the minimum Euclidean distance of the query point p to the MBR B [17]. We apply the same procedure recursively until we process the last sub-query Q_n, and get a final set of MBRs C to check for matches. The complete algorithm is shown in Algorithm 3.

Algorithm 3 Pattern_Query_Online(Query Q)

begin procedure
partition Q into n parts as $Q_1, Q_2, \ldots, Q_n$;
find the resolution level j_1 such that $|Q_1| = W2^{j_1}$;
R := Range_Query(Index$_{j_1}$, DWT($\hat{Q}_1$),$Q.r$);
C := Hierarchical_Radius_Refinement(R,Q);
post-process C to discard false alarms;
end procedure

Assume that a batch algorithm with $T_j = W$ is used for index construction. Therefore, the stream is divided into W-step sliding windows of size w. Let $|S|$ denote the size of the stream. Then, there are $\lfloor(|S| - w + 1)/W\rfloor$ such windows. Given a query sequence Q, we extract W prefixes of size w as $Q[0 : w-1], Q[1 : w], \ldots, Q[W-1 : w+W-1]$. We use each prefix query to identify potential candidates. In order to clarify the ensuing development, we note that a single prefix query would suffice in case an online algorithm with $T_j = 1$ was used for index construction. Our approach is similar to a recent work by Moon et al [14]. The authors construct a single resolution index using a sliding window of maximum allowable size w that satisfies $1 \leq \lfloor(\min(Q) - W + 1)/w\rfloor$, where min(Q) is the *a priori* information regarding the minimum query length. However, in our case, a given query can be answered using any index at resolution j that satisfies $1 \leq \lfloor(|Q| - W + 1)/(2^j W)\rfloor$. In order to improve the accuracy of our search algorithm, we extract disjoint windows along with each prefix in order to refine the original query radius as in the multi-piece search technique proposed by Faloutsos et al [7]. The number of such disjoint windows is at most $p = \lfloor(|Q| - W + 1)/w\rfloor$. We illustrate these concepts on a query window of size $|Q| = 9$ as shown in Figure 3, where $J = 1$ and $W = 2$. The prefixes are shown as $i = 0$ and $i = 1$ along with the corresponding disjoint windows. We insert each and every feature extracted over Q into a query MBR B. The MBR B is extended in each dimension by a fraction of the query radius, i.e., $r/\sqrt{p}$. Later, we perform a range query over the index at level j using B and retrieve a set R of candidate features. We post-process R to discard false alarms. The complete algorithm is shown in Algorithm 4.

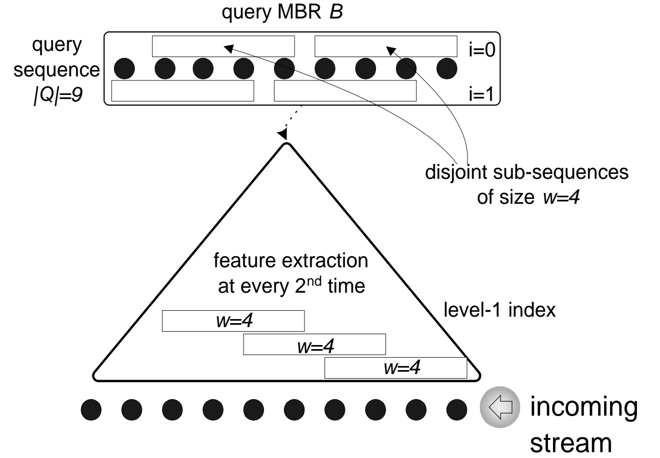


Figure 3. Subsequence query decomposition for a query window of size $|Q| = 9$.

Algorithm 4 Pattern_Query_Batch(Query Q)

begin procedure
find the largest level j such that $2^j W + W - 1 \leq |Q|$;
initialize query MBR B to empty;
let w be equal to $2^j W$, level-j sliding window size;
for $i := 0$ to $i := W - 1$ **do**
  for $k := 0$ to $k := \lfloor(|Q| - i)/w\rfloor$ **do**
    extract k^{th} disjoint subsequence of the query
      sequence into $y := Q[i + kw : i + (k+1)w - 1]$;
    insert DWT($\hat{y}$) into MBR B;
  end for
end for
compute radius refinement factor $p := \lfloor(|Q|-W+1)/w\rfloor$;
enlarge query MBR B by $Q.r/\sqrt{p}$;
R := Range_Query(Index$_{j}$, B);
post-process R to discard false alarms;
end procedure

5.3. Monitoring a correlation query

Whenever a new feature $\mathcal{F}_j$ of a stream S is computed at level j, we perform a range query with $\mathcal{F}_j$ as the center and the radius set to r. In a system with M synchronized streams,

this involves execution of $O(M)$ range queries at every data arrival. One important thing to note here is that the difference between a correlation query and a pattern query comes from the type of normalization performed (see Sections 2.3 and 2.4). We omitted the details of the algorithm due to space constraints.

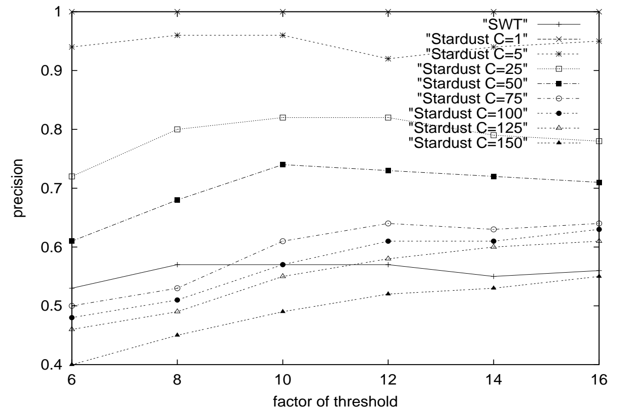
6. Performance Evaluation

We used synthetic and real data in our experiments. The synthetic data streams were generated using a random walk model. For a stream x, the value at time i $(0 < i)$ is $x[i] = R + \sum_{j=1}^{i}(u_j - 0.5)$ where R is a constant uniform random number in $[0, 100]$, and u_j is a set of uniform random real numbers in $[0, 1]$. The first set of real data is the Host Load trace data [8]. A total of 570 traces (each of size $3K$) were collected in late August 1997 at Carnegie Mellon University (CMU) on a group of machines. The second set of real data consists of two real time series: (1) `burst.dat` of size $9,382$ and (2) `packet.dat` of size $360,000$ from UCR Time Series Data Mining Archive [11].

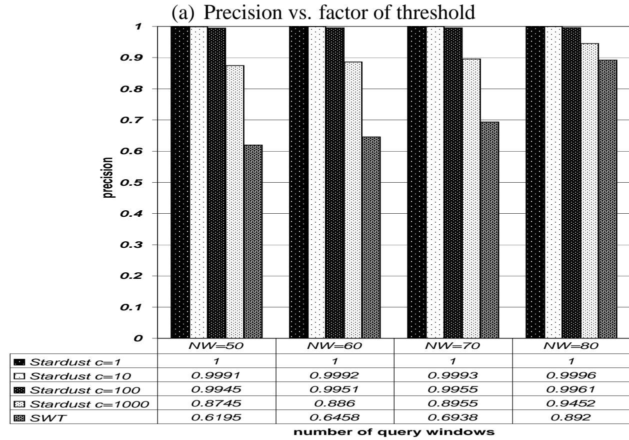
In the experiments, we measured precision and response time. Precision is the ratio of the number of relevant records retrieved to the total number of records retrieved (or the ratio of true alarms raised to the total alarms raised). Response time is query response time and/or maintenance time: query response time is the time that elapses from query issue time to the time of an answer, and maintenance time is the time spent on maintaining summary structures. In most of the experiments, we are interested in the average behavior of the algorithms, therefore we collected our measurements on queries of uniformly random length that were generated using the random walk model. Measurements were collected on machines with dual AMD Athlon MP 1600+ processors, 2 GB of RAM, and running Linux 2.4.19.

6.1. Monitoring aggregates

In this set of experiments, we computed aggregates of data streams based on a given a set of window sizes w_1,w_2,. . .,w_m, an aggregate function F and a threshold τ_{w_i} associated with each window size. We used the second set of the real datasets. We set the thresholds of different window sizes as follows: we used 1K of burst data and 8K of network packet data as training data. For a window of size w, we computed the aggregates on the training data with a sliding window of size w. This gives another time series y. The thresholds are set to be $\mu_y + \lambda * \sigma_y$, where μ_y and σ_y are the mean and standard deviation respectively. The window sizes are $K, 2K, \ldots, m * K$ time units, where K is a constant and m is the number of query windows. We measured the true alarm rate (precision) by the ratio of the true alarms raised to the total alarms raised, and the number of alarms raised for an increasing query set size. We compared our technique to SWT, a burst detection scheme for data streams [21].

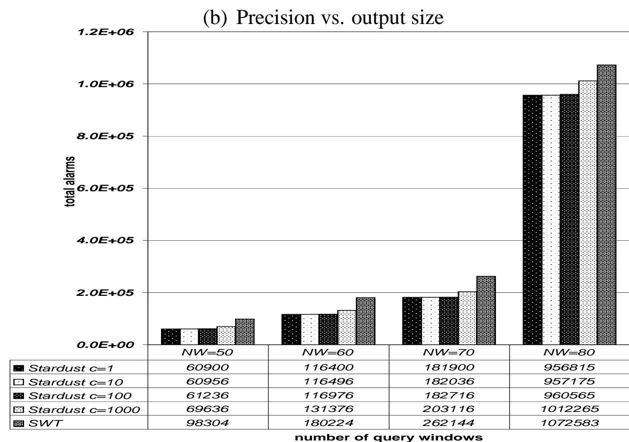


(a) Precision vs. factor of threshold



	NW=50	NW=60	NW=70	NW=80
Stardust c=1	1	1	1	1
Stardust c=10	0.9991	0.9992	0.9993	0.9996
Stardust c=100	0.9945	0.9951	0.9955	0.9961
Stardust c=1000	0.8745	0.886	0.8955	0.9452
SWT	0.6195	0.6458	0.6938	0.892

(b) Precision vs. output size



	NW=50	NW=60	NW=70	NW=80
Stardust c=1	60900	116400	181900	956815
Stardust c=10	60956	116496	182036	957175
Stardust c=100	61236	116976	182716	960565
Stardust c=1000	69636	131376	203116	1012265
SWT	98304	180224	262144	1072583

(c) Total number of alarms vs. number of query windows

Figure 4. Performance results on (a) detecting bursts of events on `burst.dat`, (b) and (c) on measuring volatility of `packet.dat`.

6.1.1 Burst detection

In order to detect bursts of events, we chose our aggregate function F to be SUM. We set $K = 20$, $m = 50$, and varied λ, the factor of threshold. The larger λ is, the higher the thresholds are, and therefore fewer alarms will be sounded. Fewer alarms also imply a faster response time. We experimented with box capacities c from 1 to 150. Since the small-

est query window size is 20, a box capacity larger than this adversely affects the performance of our algorithm, based on the analysis in Section 5.1. Figure 4(a) shows the results on the dataset `burst.dat`. Our algorithm has a superior quality of answers compared to SWT, which itself is more than ten times faster than the linear scan [21]. Stardust with $c = 1$ is the exact algorithm with no false alarms. For all box capacities except the largest $c = 150$, Stardust is more selective than SWT with increasing factor of threshold due to the best effort approximation. For example, Stardust with $c = 25$ offers a precision of 0.82 compared to 0.57 offered by SWT for the case of $\lambda = 10$.

The false alarm rate of a technique is the ratio of false alarms raised to the total alarms raised. For example, Stardust with $c = 5$ has a true alarm rate of 0.95 for $\lambda = 16$; therefore the false alarm rate of Stardust in this case is 0.05 compared to the false alarm rate 0.44 of SWT.

6.1.2 Volatility detection

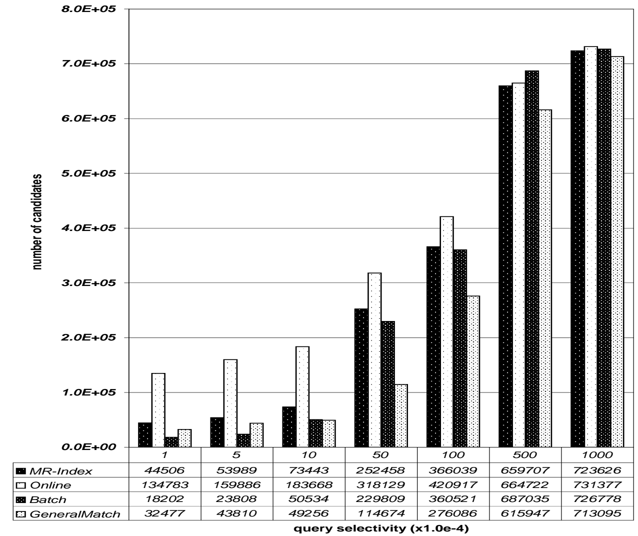
In this experiment, we are interested in abnormal volatility F=SPREAD(x), which is the measure of MAX(x)−MIN(x) for a given time series x. We intentionally set the threshold λ to a small value, i.e., to 0.12, and produced many more alarms than what domain experts are interested in. We set $K = 100$, varied m over $\{50, 60, 70, 80\}$, and the box capacity c over $\{1, 10, 100, 1000\}$. Figure 4(b) shows the results on the dataset `packet.dat`. In all test cases, our algorithm outperformed SWT, which itself is up to 100 times faster than the linear scan. The improvement is again due to our best-effort approximation, which reduces the false alarm rate to a minimal amount. For the example case of $NW = 60$, Stardust with $c = 100$ offers a precision of 0.89 with a total of $116,976$ alarms raised, compared to a precision of 0.64 with a total of $180,224$ alarms raised by SWT as shown in Figure 4(c). The false alarm rate of Stardust with $c = 10$ for $NW = 50$ is 0.0009 compared to the false alarm rate 0.39 of SWT. We note that the number of false alarms directly affects the response time of the algorithm.

6.2. Monitoring for surprising patterns

In this set of experiments, we measured the average precision offered to a set of one-time pattern queries of uniformly random length that were generated using the random walk model. We compared our system performance to MR-Index, a novel multi-resolution index structure [9] and to GeneralMatch, a recent state-of-the-art single-resolution index structure [14].

6.2.1 Varying query length and radius

We performed a set of experiments on the host load dataset. We set N to 1024, and observed up to 3K data arrivals for each stream. We computed the average precision on a query workload of 100 uniformly generated variable length queries of length $192, \ldots, 64 \times k, \ldots, 1024$ at each measurement point. Figure 5 shows the results. The online algorithm is the worst among all four competing techniques, which suggests that a batch algorithm is the algorithm of choice for pattern matching. For low selectivity queries, our batch algorithm outperformed the other two techniques by more than two times. The reason behind this is that our algorithm uses larger windows in answering the query, therefore it has a higher precision. This also implies that our batch algorithm will have a superior quality for point queries. However, for high selectivity queries, GeneralMatch is the clear winner. The reason for this difference is the value of the radius refinement factor p (see Section 5.2 for a discussion) because the smaller the window size used for index construction is the larger the factor p is; therefore, fewer candidates may be returned for high selectivities. However, the difference is marginal, as shown for increasing values of selectivity. Furthermore, it is not difficult to modify our algorithm so that it becomes adaptive and works better for all selectivities: one can decrease the resolution of the index used for high selectivity queries. We confirmed this claim with more experiments; however, omitted these uninteresting results.



	1	5	10	50	100	500	1000
MR-Index	44506	53989	73443	252458	366039	659707	723626
Online	134783	159886	183668	318129	420917	664722	731377
Batch	18202	23808	50534	229809	360521	687035	726778
GeneralMatch	32477	43810	49256	114674	276086	615947	713095

Figure 5. Average precision on the Host Load dataset for $N = 1024$, $W = 64$, $M = 25$, $c = 64$, and $f = 2$.

6.3. Monitoring correlations

Finally, we considered correlation queries in our experiments. We set the box capacity c to 1, and updated each and every level at every W time units by using a batch algorithm. We detect correlations at resolution J where $N = 2^J W$. We compared our algorithm with StatStream, a recent correlation detection scheme proposed for data streams [20].

StatStream uses an orthogonal grid of cells to detect correlations. A cell in an f-dimensional space has $3^f - 1$ neighbor cells, leading to a search volume of $(3^f - 1)r^f$. A range query in an f-dimensional space with radius r has volume

Scalability	$r = 0.01$		$r = 0.02$		$r = 0.04$		$r = 0.08$	
Number of streams	StatStream	Stardust	StatStream	Stardust	StatStream	Stardust	StatStream	Stardust
256	711	911	1422	912	9093	1232	90550	1532
512	1442	1672	2854	1743	17565	2434	182933	4587
1024	2183	3224	5448	3445	41790	5759	377202	15042
2048	4226	6279	11987	7080	90310	14561	802234	50923
4096	9103	12578	30143	16364	200398	44534	1865893	225444
8192	35751	27350	101986	43332	600103	186855	4845718	956756

Table 1. Total time (ms) spent on correlation detection for an increasing number of streams.

$\pi^{f/2} r^f/(f/2)!$ assuming that f is even. The ratio of the search volume of the cell technique to the volume of a range query with radius r is $(3^f - 1)(f/2)!/\pi^{f/2}$. Furthermore, if we use the same grid structure to detect correlations with threshold larger than r (e.g., $2r$, $3r$,...,br), we need to check distant cells as well. The ratio increases sharply for a threshold of br, since a cell in f-dimensional space in this case has $(2b + 1)^f - 1$ neighbor cells.

6.3.1 Scalability comparison

We compared the total wall clock time of both techniques under a varying number of streams and a varying correlation threshold. The total wall clock time is the time spent for maintaining the summary structures and computing the correlations. The cell radius in StatStream is set to 0.01. We set $N = 256$, $W = 16$, and $f = 2$. We monitored $M \in \{256, 512, 1024, 2048, 4096, 8192\}$ synthetic data streams and observed 256 arrivals for each stream.

As shown in Table 1, Stardust detected correlations faster than StatStream for various threshold values. The performance improvement is from 5 times up to 60 times.

6.3.2 Effect of increasing dimensionality

We compared the average precision and average correlation detection time under varying dimensionality on 1000 synthetic data streams of 2048 data points each. The cell radius in StatStream was set to 0.1. We work on a history of size $N = 1024$ and a window W of size 64. We extracted $f = 2$ coefficients for each stream in both techniques. Stardust outperformed StatStream when the correlation threshold exceeds 0.5 in these particular settings. For example, Stardust detected the correlations with threshold $r = 1$ in 325.90 seconds with a precision of 0.29, whereas StatStream detected the same correlations in 1430.51 seconds with a precision of 0.22.

The performance of Stardust improved significantly both in precision and in correlation detection time by increasing the number of coefficients f. We observed this behavior in Figures 6 (a) and (b). For the correlation threshold of $r = 1$, the precision of Stardust increased from 0.29 to 0.74, and the correlation detection time decreased from 325.90 seconds to 135.80 seconds for $f = 2$ and $f = 16$ respectively.

The performance of StatStream degraded with increasing number of coefficients. Therefore, we omitted the results on StatStream under varying number of coefficients. In the experimental results, Stardust with $f = 16$ outperformed StatStream with $f = 2$ by up to 20 times in precision, and by up to 18 times in correlation detection time.

7. Concluding Remarks

In this paper, we have presented a space and time efficient framework `Stardust` to summarize and index multiple data streams. We reduce the maintenance cost of our index structure by computing transformation coefficients online: we compute coefficients at higher levels on the index that stores the coefficients at lower levels. This approach decreases per-item processing time considerably, and minimizes the space required for incremental computation. The index structure has an adaptive time-space complexity depending on the update rate and the number of coefficients maintained, and guarantees the approximation quality by provable error bounds. We have provided an extensive set of experiments showing the effectiveness of our framework. In the future, we will explore fitting incremental regression models in our framework in order to enable parameter estimation, e.g., determining the right window sizes to monitor, for different kinds of queries.

References

[1] A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. In *VLDB*, pages 336–347, 2004.

[2] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An effi cient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.

[3] J. Bentley, B. Weide, and A. Yao. Optimal expected time algorithms for closest point problems. In *ACM Trans. on Math. Software*, volume 6, pages 563–580, 1980.

[4] A. Bulut and A. K. Singh. SWAT: Hierarchical stream summarization in large networks. In *ICDE*, pages 303–314, 2003.

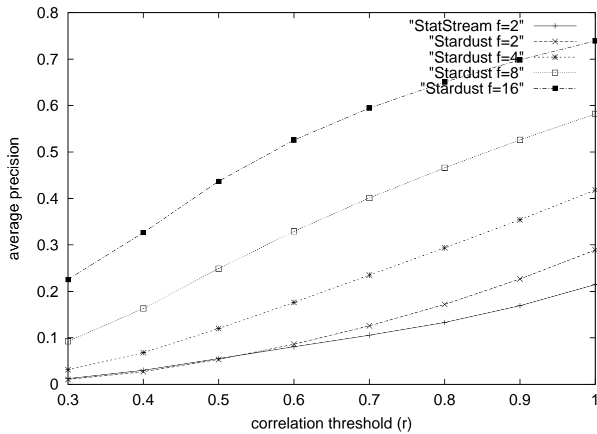
[5] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Monitoring streams - a new class of data management applications. In *VLDB*, pages 215–226, 2002.

[6] Y. Chen, G. Dong, J. Han, J. Pei, B. Wah, and J. Wang. Online analytical processing stream data: Is it feasible? In *DMKD*, 2002.

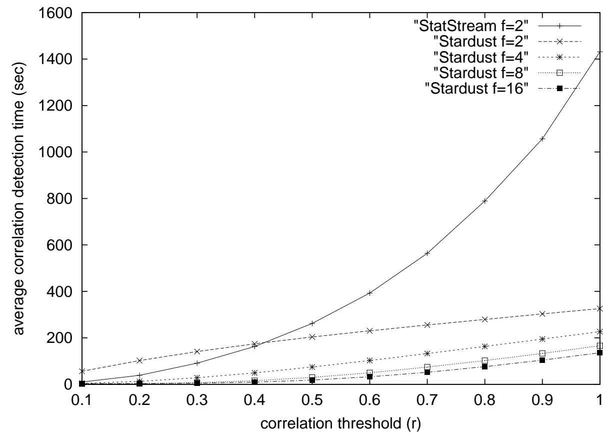
[7] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *SIGMOD*, pages 419–429, 1994.

[8] http://www.cs.nwu.edu/˜pdinda/LoadTraces/.

[9] T. Kahveci and A. K. Singh. Variable length queries for time series data. In *ICDE*, pages 273–282, 2001.



(a) average precision vs. varying f and r



(b) average correlation detection time vs. varying f and r

Figure 6. Dimensionality comparison on Synthetic dataset for $N = 1024$, $M = 1000$, and $W = 64$.

[10] E. Keogh, K. Chakrabarti, S. Mehrotra, and M. Pazzani. Locally adaptive dimensionality reduction for indexing large time series databases. In *SIGMOD*, pages 151 – 162, 2001.

[11] E. Keogh and T. Folias. Time Series Archive. In *http://www.cs.ucr.edu/~eamonn/TSDMA*, 2002.

[12] M. Lee, W. Hsu, C. S. Jensen, B. Cui, and K. L. Teo. Supporting frequent updates in R-Trees: A bottom-up approach. In *VLDB*, pages 608–619, 2003.

[13] S. Mallat. *A Wavelet Tour of Signal Processing*. Academic Press, 2 edition, 1999.

[14] Y. Moon, K. Whang, and W. Han. General match: A subsequence matching method in time-series databases based on generalized windows. In *SIGMOD*, pages 382–393, 2002.

[15] T. Palpanas, M. Vlachos, E. Keogh, D. Gunopulos, and W. Truppel. Online amnesic approximation of streaming time series. In *ICDE*, pages 338–349, 2004.

[16] S. Papadimitriou, A. Brockwell, and C. Faloutsos. AWSOM: Adaptive, hands-off stream mining. In *VLDB*, pages 560–571, 2003.

[17] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. pages 71–79, 1995.

[18] H. Wu, B. Salzberg, and D. Zhang. Online event-driven subsequence matching over fi nancial data streams. In *SIGMOD*, pages 23–34, 2004.

[19] B. Yi, N. D. Sidiropoulos, T. Johnson, H. V. Jagadish, C. Faloutsos, and A. Biliris. Online data mining for co-evolving time sequences. In *ICDE*, pages 13–22, 2000.

[20] Y. Zhu and D. Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *VLDB*, pages 358–369, 2002.

[21] Y. Zhu and D. Shasha. Effi cient elastic burst detection in data streams. In *SIGKDD*, pages 336–345, 2003.

A. Discrete Wavelet Transform

The approximation coefficients are defined through the inner product of the input signal with $\phi_{j,k}$, the shifted and dilated versions a low-pass scaling function ϕ_0. In the same vein, the detail coefficients are defined through the inner product of the input signal with $\psi_{j,k}$, the shifted and dilated versions the wavelet basis function ψ_0.

$$\phi_{j,k}(t) = 2^{-j/2}\phi_0(2^{-j}t - k),\ j,k \in Z \tag{8}$$

$$\psi_{j,k}(t) = 2^{-j/2}\psi_0(2^{-j}t - k),\ j,k \in Z \tag{9}$$

From this point on, we will only consider how to compute approximation coefficients. This is because detail coefficients at level j are computed using approximation coefficients at level $j-1$. Using Equation 8, we can obtain the approximation signal at level j for the signal x as follows

$$A_j^{(x)} = \sum_k \langle x, \phi_{j,k}\rangle \phi_{j,k}$$

In the same manner, the approximation signal at level $j+1$ for x is

$$A_{j+1}^{(x)} = \sum_k \langle x, \phi_{j+1,k}\rangle \phi_{j+1,k}$$

To compute $A_{j+1}^{(x)}$, we need to compute coefficients $\langle x, \phi_{j+1,n}\rangle$. Using the twin-scale relation for ϕ, we can compute $\langle x, \phi_{j+1,n}\rangle$ from $\langle x, \phi_{j,k}\rangle$ [13]. This can mathematically be expressed as

$$\langle x, \phi_{j+1,n}\rangle = \sum_k h_{k-2n}\langle x, \phi_{j,k}\rangle \tag{10}$$

$$C_{j,n} = \sum_k \tilde{h}_{n-k}\langle x, \phi_{j,k}\rangle \tag{11}$$

$$\langle x, \phi_{j+1,n}\rangle = C_{j,2n} \tag{12}$$

where h_k and $\tilde{h}$ are low-pass reconstruction and decomposition filters respectively. We use the terms "approximation signal" and "approximation coefficients" interchangeably.

Lemma A.1 *The approximation coefficients at level j, $1 \leq j \leq J$, for a signal $x[t-w+1:t]$ can be computed exactly using the approximation coefficients at level $j-1$ for the signals $x[t-w+1:t-w/2]$ and $x[t-w/2+1:t]$.*

Proof Let x, x_1 and x_2 denote signals $x[t-w+1:t]$, $x[t-w+1:t-w/2]$, and $x[t-w/2+1:t]$ respectively. At a particular scale $j-1$, the shape of the wavelet scaling function $\phi_{j-1,0}$ is kept the same, while it is translated to

obtain the wavelet family members at different positions, k. Since x_1 and x_2 constitute the two halves of the whole signal x, we can easily compute $\langle x, \phi_{j-1,k} \rangle$ using $\langle x_1, \phi_{j-1,k} \rangle$ and $\langle x_2, \phi_{j-1,k} \rangle$ as follows

$$x'_1 = \left\{ \begin{array}{ll} x_1[n], & 1 \leq n \leq w/2 \\ 0, & w/2+1 \leq n \leq w \end{array} \right\}$$

$$x'_2 = \left\{ \begin{array}{ll} 0, & 1 \leq n \leq w/2 \\ x_2[n-w/2], & w/2+1 \leq n \leq w \end{array} \right\}$$

$$\langle x, \phi_{j-1,k} \rangle = \langle x'_1 + x'_2, \phi_{j-1,k} \rangle = \langle x'_1, \phi_{j-1,k} \rangle + \langle x'_2, \phi_{j-1,k} \rangle$$

This result is due to the linearity of the wavelet transformation. Using Equations 11, 12, and the coefficients $\langle x, \phi_{j-1,k} \rangle$, we can obtain the approximation signal $A_j^{(x)}$ at level j for x. ∎

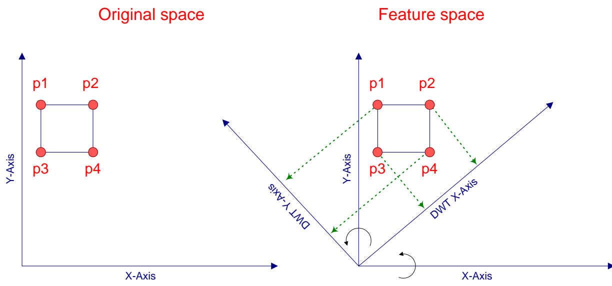


Figure 7. Transforming an MBR using discrete wavelet transform. Transformation corresponds to rotating the axes (the rotation angle = $45°$ for `Haar` wavelets)

In order to compute the most recent approximation coefficients for a sliding window of values x at a given resolution j, we can use the extent information of the MBRs B_{j-1,i_1} and B_{j-1,i_2} in $\Re^f$ at level $j-1$ that contain the coefficients of the corresponding two halves of x. These MBRs are merged together using Lemma A.1 to get an MBR B in $\Re^{f'}$, where f' is larger than f (e.g., f' is $2f$ for `Haar` wavelets). The MBR B approximates the coefficients at level $j-1$ for x. In order to compute the coefficients at level j for x, one can compute the coefficients for each one of the $2^{f'}$ corners of B, and find the tightest MBR in $\Re^f$ that encloses the resulting $2^{f'}$ coefficients in $\Re^f$. This is true for any such unitary transformation as wavelet transformation that rotates the axes as shown in Figure 7. However this algorithm (`Online I`) has a processing time of $\Theta(2^{f'} f)$. To reduce this cost, we propose a new technique (`Online II`) that uses only two of the corner points, namely the low and high coordinates of B. This approach achieves a faster processing time of $\Theta(f)$ at the cost of a lower accuracy.

Lemma A.2 *It is possible to compute approximation coefficients on a hyper-rectangle $B \in \Re^{f'}$ with low coordinates $[x_{l_1}, \ldots, x_{l_{f'}}]$ and with high coordinates $[x_{h_1}, \ldots, x_{h_{f'}}]$.*

Proof Approximation coefficients are computed by first convolving (*) the signal with the low-pass filter as in Equation 11 and then down-sampling (↓) the resulting signal as in Equation 12. Let x_{lo}, x_{hi} and x_i denote the signals corresponding to the low coordinates of B, high coordinates of B and an arbitrary point inside B respectively. Let $A^{(B)}$ denote the resulting transform. We can simply compute $A^{(x_{lo})}$ and $A^{(x_{hi})}$ to get the low coordinates, $A^{(B_{lo})}$, and the high coordinates, $A^{(B_{hi})}$, of $A^{(B)}$. This is true only if the low-pass filter contains all non-negative entries as in `Haar` wavelets. If not, we can use the linearity of convolution operation as follows:

$$A^{(x)} = \downarrow (x * (\tilde{h} + \delta - \delta)) = \downarrow (x * (\tilde{h} + \delta) - x * \delta) \quad (13)$$

where the amplitude scaling filter δ is a constant amplitude filter with the smallest positive amplitude that makes all the entries of $\tilde{h} + \delta$ nonnegative. With this insight, it is easy to verify for all $x_i \in B$ that:

$$\begin{array}{rcl} x_i * \tilde{h} & = & x_i * (\tilde{h} + \delta) - x_i * \delta \\ x_{lo} * (\tilde{h} + \delta) & \leq & x_i * (\tilde{h} + \delta) \\ x_{lo} * \delta & \leq & x_i * \delta \\ x_i * (\tilde{h} + \delta) & \leq & x_{hi} * (\tilde{h} + \delta) \\ x_i * \delta & \leq & x_{hi} * \delta \end{array}$$

which collectively imply a lower bound and an upper bound on any feature vector $A^{(x_i)}$ to be computed before down-sampling as

$$x_{lo} * (\tilde{h} + \delta) - x_{hi} * \delta \leq \min_i \{x_i * \tilde{h}\} \quad (14)$$

$$\max_i \{x_i * \tilde{h}\} \leq x_{hi} * (\tilde{h} + \delta) - x_{lo} * \delta \quad (15)$$

Using Equations 14 and 15, we can compute low and high coordinates of $A^{(B)}$ as

$$A^{(B_{lo})} = \downarrow (x_{lo} * (\tilde{h} + \delta) - x_{hi} * \delta) \quad (16)$$

$$A^{(B_{hi})} = \downarrow (x_{hi} * (\tilde{h} + \delta) - x_{lo} * \delta) \quad (17)$$

Lemma A.2 establishes that for a given $x_i \in B$, the inequality $A^{(B_{lo})} \leq A^{(x_i)} \leq A^{(B_{hi})}$ holds.

A.1. Error bound

Wavelet transformation corresponds to the rotation of the axes in the original space. An input MBR B in the original space is transformed to a new shape S in the feature space (see Figure 7). We project the resulting shape S on each dimension in the feature space, and find the tightest MBR $A^{(B)}$ that encloses S. The new MBR $A^{(B)}$ contains the feature $A^{(x)}$ we ultimately want to compute. The volume of $A^{(B)}$ is a function of the projection along each dimension. Since the wavelet transformation is a distance preserving transformation, the length along each dimension can be at most two times the original length.