SWAT: Hierarchical Stream Summarization in Large Networks

Ahmet Bulut Ambuj K. Singh

Department of Computer Science, University of California,

Santa Barbara, CA, 93106

{bulut,ambuj}@cs.ucsb.edu

February 26, 2003

Abstract

The problem of statistics and aggregate maintenance over data streams has gained popularity in recent years especially in telecommunications network monitoring, trend-related analysis, web-click streams, stock tickers, and other time-variant data. The amount of data generated in such applications can become too large to store, or if stored too large to scan multiple times. We consider queries over data streams that are biased towards the more recent values. We develop a technique that summarizes a dynamic stream incrementally at multiple resolutions. This approximation can be used to answer point queries, range queries, and inner product queries. Moreover, the precision of answers can be changed adaptively by a client.

Later, we extend the above technique to work in a distributed setting, specifically in a large network where a central site summarizes the stream and clients ask queries. We minimize the message overhead by deciding what and where to replicate by using an adaptive replication scheme. We maintain a hierarchy of approximations that change adaptively based on the query and update rates. We show experimentally that our technique performs better than existing techniques: up to 50 times better in terms of approximation quality, up to four orders of magnitude times better in response time, and up to five times better in terms of message complexity.

1 Introduction

Data streams are becoming an increasingly important class of datasets. They find applications in sensor networks, stock tickers, news organizations, telecommunications, and data networks. Network service providers channelize logs of network usage in great detail from routers into data processing centers to use them in trend-related analysis. Consider the following application scenario that arises in telecommunications network monitoring. A tremendous number of connections are handled every minute by switches. Typically, for each call, a switch dumps a *Call Detail Record*. The amount of data generated in such applications can become very large to maintain in memory. This prompts a need to maintain summary or statistics information that can be used to answer queries approximately without reading the whole dataset from disk.

Applications in forecasting involve predicting the future conditions using the last few measurements. For example, in the case of banner-hits data, the number of hits in the immediate past can be used to gauge the popularity of an advertisement. In fraud and security monitoring, the recent data has more predictive value compared to the old data. Therefore a system which maintains better approximations for the recent data is useful. The challenge is to maintain these biased approximations continuously as new data arrives in an online manner.

A fundamental requirement of any good aggregate maintenance technique is to approximate the underlying data distribution in a provably good manner. The technique has to be efficient in terms of (1) the space used to maintain summary structures, (2) the time required to process incoming data, and (3) the time needed to answer posed queries. Furthermore, to be of any practical value, the precision offered by the system has to satisfy the accuracy measures specified by the users.

In this paper, we develop a new wavelet-based approximation scheme that maintains multiple levels of information about a dynamic stream. Multiple levels correspond to varying precision. We keep more precise approximations for more recent data and less precise approximations for older data. This information is maintained incrementally using a unique time varying tree structure.

Besides the usual point and range queries, we consider the case of *inner product queries*. In such queries, the client supplies a query vector and an inner product is computed of this vector with the data stream. Inner product queries allow an easy specification of the relevant data points and their weighted contribution to the final result.

In the latter half of the paper, we show how to extend our technique in a real-world setting where clients are distributed and desire varying degrees of consistency about a stream. Let us consider the telecommunications network monitoring scenario mentioned above. When local buffers get full, switches dump *Call Detail Records* into a central or distributed data processing facility. Later on, these records are channelled into network operations centers. Hypothetically, we can think of data processing facility as the central site which maintains summary of the stream, and the operation centers as the clients that ask queries over the stream for analysis. A system that uses least amount of resources to manage this task is crucial. Our extended approach can be used in such scenarios to minimize the message overhead, and reduce network latency.

1.1 Related work

There has been a significant amount of previous work on maintaining approximations to a data stream. The usual metric in evaluating such algorithms has been the time and space complexity of generating (and maintaining) the approximation and the error bound. Alon, Matias, and Szegedy [1] present the idea of *random sketches* and show how they can be used to compute frequency moments of a data stream. The frequency moments can later be used to compute statistics such as the number of unique values, size of self-join etc.

Dobra et al. [5] consider how to generalize the basic sketching technique proposed in [1] to compute approximate answers to complex aggregate queries over data streams. Specifically, they consider building pseudo-random sketch summaries to answer multi-join queries over multiple streaming relations with error guarantees. The variance of the sketches increases exponentially as the number of joins involved increase. To this end, they show how to partition the multi-dimensional join-attribute space so as to minimize the overall space allocated to the sketches with guarantees on the approximation error.

Guha et al. [10] have presented a deterministic algorithm that solves the k-median problem for a given data stream in a single pass within a polylog approximation. They also present a randomized algorithm that achieves a constant approximation to the optimal solution. These medians can then be used to cluster the stream. O'Callaghan et al. [14] have presented another clustering algorithm that computes the k-median, and show theoretically as well as experimentally that it performs well.

A histogram-based synopsis of a stream can be computed in $O(B^2N)$ time and $O(N)$ space where B is the number of buckets [12]. Guha, Koudas, and Shim [9] present an algorithm for computation of histograms that trades accuracy for speed: a $(1+\epsilon)$ approximate histogram of a stream of length N is computed in $O(B^2N\log N/\epsilon)$ time and $O(B^2\log N/\epsilon)$ space. In a recent paper, Guha and Koudas [8] extend this idea to incremental computation of histograms. By restricting the computation to a smaller set of points and by using a binary search, they reduce the time complexity of computing approximate histograms to $O((B^3\log^3 N)/\epsilon^2)$ for every arriving value.

Datar et al. [4] consider the problem of maintaining statistics such as count, sum, and L_p-norm over a sliding window of last N elements. For the case of count, the authors propose a scheme in which the sliding window is divided into buckets of exponentially increasing sizes, while ensuring that the number of buckets of the same size varies in a narrow range based on the maximum allowed error, ϵ. The total number of bits needed is $\frac{1}{\epsilon}\log^2 N$ and the estimate of count is maintained within an error of $(1+\epsilon)$. The authors then extend this basic scheme to maintain other statistics such as sum and L_p norm.

Gilbert et al. [7] investigate the summarization of streams under different models. For the *ordered aggregate model* (the one that we consider here), the authors present a wavelet-based scheme that summarizes a stream of length N through its largest B wavelet coefficients in $O(B+\log N)$ space as the stream arrives. No such summarization is possible in the other streaming models considered by the authors. This prompts them to develop *sketch-based* methods, which can be used to answer point queries, range queries, and more importantly to compute the wavelet coefficients themselves. The wavelet coefficients can then be used to reconstruct the data stream and answer queries. Our approach is different in that we consider the incremental computation of a summary over the most recent window rather than the complete stream.

Gehrke, Korn, and Srivastava [6] consider the computation of correlated aggregates over multiple streams. They present a technique in which such queries can be computed approximately in a single pass over the streams. They use histograms as a summary structure. They approximate aggregates by estimating the overlap with the existing buckets. Bucket allocation degrades as new values arrive. Therefore, they propose two methods to tune the old buckets: wholesale approach and piecemeal approach. In the wholesale approach, the buckets are revised from scratch, while in the piecemeal approach, the existing bucket allocation is preserved whenever possible.

A recent paper by Zhu and Shasha [17] considers the problem of monitoring a large number of streams in real time. The authors subdivide a sliding window into a fixed number of basic windows and maintain DFT coefficients for each basic window. This transforms each stream into a feature space consisting of a set of DFT coefficients. Correlation between two streams can be estimated by normalizing the values within the sliding window and computing their L2 distance. The feature vectors are mapped to a grid structure and proximity within this structure is used to reduce the computational overhead.

Babcock et al. [2] consider models and issues towards designing a *Data Stream Management System (DSMS)*. They argue that the traditional DBMS's are ill-equipped to support continuous queries, which are typical for data streams. The main concern in a DBMS is to provide exact answers, while *approximation* and *adaptivity* are key features for a DSMS. Typically, the amount of memory required for evaluating a data stream query is unbounded due to the unboundedness of the stream itself. Therefore, it is not always possible to produce exact answers under realistic assumptions, i.e. limited memory; however high-quality approximate answers are often acceptable. The authors discuss the general schemes such as sampling, batch processing, and synopsis data structures etc., to keep up with the data stream rate and to produce timely answers. Finally, a query language, an extension of the standard SQL, is proposed. The new query language effectively expresses the temporal constructs, more specifically captures the notion of *sliding windows*.

The second part of our paper considers the problem of maintaining approximations adaptively in a large network. There is a considerable body of literature on this problem. We cite only the most relevant papers here. Huang, Sloan, and Wolfson [11] consider the problem of adaptive caching of a single object value by estimating the write rate at the server and the read rate at the client. A read operation defines its tolerance as how recent a version of the object is required. A read tolerance of one means that the most recent version is needed; in general a read tolerance of k implies that any of the last k versions of the object will suffice. The authors first consider a static version of the problem in which the write rate and the read rates are known for each level of tolerance. They compute the expected cost for each refresh rate (the rate at which updates are transmitted from the server to the client), and the refresh rate with the minimum expected cost is chosen by the server. Later they generalize to the case when the read and write rates are not known a priori. Here, a finite window is used to compute the likely read and write rates, and the minimum cost refresh rate is found as before. This scheme is called *divergence caching*. We adapt the central idea of this algorithm to the problem of maintaining approximations. We define tolerance of a client in terms of the desired precision rather than how recent the value is.

Olston, Loo, and Widom [15] consider the adaptive caching of a single value in a client server architecture. An approximation is maintained for a value in terms of a range. If a client finds the current range inadequate, it requests the exact value and a possibly tighter range through a *query-initiated refresh*. If a data update at the server leads to the current cached range at a client becoming invalid, a correct range, possibly wider, is transmitted to the client through a *data-initiated refresh*. The authors discuss the appropriate choice of parameters to optimize the frequency of these refreshes.

Wolfson, Jajodia, and Huang [16] propose a distributed dynamic replication algorithm called Adaptive Data

Replication (ADR). The ADR algorithm changes the replication scheme of an object as the read-write pattern of the object changes. The algorithm changes the scheme in a way that the communication cost, the average number of inter-processor messages required for the access of the object, is minimized. The replication scheme is a variable-size amoeba that stays connected throughout the execution, and moves toward the center of ”access activity”. The amoeba expands as the read activity increases, and it contracts as the write activity increases. We use this algorithm as the underlying basis in one of our techniques.

1.2 Our contribution

We propose a novel technique to summarize a data stream incrementally. The summaries over the stream are computed at multiple resolutions, and together they induce a unique approximation tree. The resolution of approximations increases as we move from the root of the approximation tree down to its leaf nodes. The tree has space complexity $O(\log N)$, where N denotes the current size of the stream. The amortized processing cost for each new data value is $O(1)$. These bounds are currently the best known for the algorithms that work under a biased query model where the most recent values are of a greater interest. In experimental results, our technique's response time was four orders of magnitude better than current techniques. Furthermore, its approximation quality was up to 50 times better.

Later, we extend this approach to data stream replication in large networks. We consider a network where a central source site summarizes a data stream at multiple resolutions. The clients distributed across the network pose queries over the data stream. The summaries computed at the central site are cached adaptively at the clients. The access pattern, i.e. reads and writes, over the stream results in multiple replication schemes at different resolutions. Each replication scheme expands as the corresponding read rate increases, and contracts as the corresponding write rate increases. This adaptive scheme minimizes the total communication cost, the number of inter-site messages. We compare this technique with two other techniques — Adaptive Precision Setting [15] and Divergence Caching [11]. Since Divergence Caching was not proposed for approximations, we first adapt the general idea to our setting. In our experiments, our technique performed up to five times better in terms of message complexity. It also adapted well to varying read-write patterns.

1.3 Outline of the paper

The remainder of the paper is structured as follows. In Section 2, we introduce our query model, and describe our multi-resolution summarization technique. Later, we present theoretical and experimental analysis to show the efficiency of the proposed technique. In Section 3, we show how to extend our unique approach, and develop a novel system for distributed and adaptive data stream replication. In Section 4, we introduce two competitive caching algorithms for performance comparison. In Section 5, we present the results of an extensive set of experiments, and show the effectiveness of our technique. Finally, in Section 6, we conclude and discuss avenues for future work.

2 Proposed Solution

2.1 Preliminaries and definitions

A data stream consists of an ordered sequence of data points $\ldots, d_i, \ldots, d_1, d_0$, where d_0 is the most recent value. The value of each data point lies in a bounded range. A sliding window of size N includes the last N points, and is updated after each arrival.

The typical query we consider is an *inner product query* that is specified by a triple (I, W, δ), where I denotes the index vector (the data items of interest), W denotes the weight vector (the individual weights corresponding to each data item), and a precision δ within which the result $I \cdot W$ needs to be computed. In other words, any value returned to the client must be within δ of the correct value of $I \cdot W$. If i^{th} data point has actual value d_i and approximation value a_i, then $\sum_i (W[i] \times |d_{I[i]} - a_{I[i]}|) \leq \delta$.

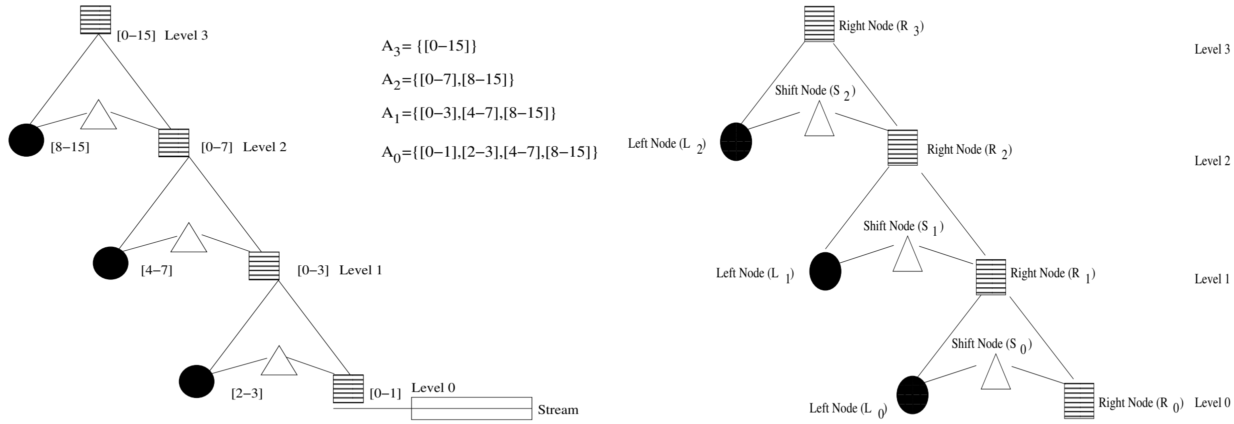
There are two types of inner product queries that are of special interest to us: *exponential* and *linear* inner product queries. An exponential inner product query has a weight vector with exponentially decreasing weights. On the other hand, a linear inner product query has a weight vector with weights that are linearly decreasing. For example, $([0, 1, 2, 3], [8, 4, 2, 1], 20)$ is an exponential inner product query over data points at indices $0, 1, 2,$ and 3, whereas $([8, 9, 10, 11], [4, 3, 2, 1], 40)$ is a linear inner product query over data points at indices $8, 9, 10,$ and 11. In both of the above query types, the data items of interest are consecutive starting with the most recent value.

Point queries are special inner product queries. For example, $([12], [1], \delta)$ is a point query asking for an δ-approximation of the data point at index 12. The details of how to evaluate a range query will be explained in Section 2.4.

Our queries are one-time, but we can extend our algorithms to continuous queries quite easily. We assume that the data values arrive incrementally, and we update a summary of the data stream after the arrival of each data value. We work under a fixed sliding window model of size N. However, our techniques are also applicable in a model where the entire stream (and not just the last N values) are of interest.

2.2 Multi-resolution approximations

We use a wavelet-based scheme to compute multi-resolution approximations of a single stream. A client can choose to approximate the stream at any level $0 \leq i \leq n-1$, where $n = \log N$. (N is usually the size of the sliding window; if the entire stream is of interest then N is the current size of the data stream.) These approximations are shown pictorially in Figure 1(a) for the case $N = 16$. A level 3 approximation denoted as A_3 stores a set of values that summarize $[0, \ldots, 15]$. A level 2 approximation denoted as A_2 stores two sets of values: one summarizing $[0, \ldots, 7]$ and the other summarizing $[8, \ldots, 15]$. A level 1 approximation denoted as A_1 stores three sets of values: the first summarizing $[0, \ldots, 3]$, the second summarizing $[4, \ldots, 7]$, and the third summarizing $[8, \ldots, 15]$. In general, a level i approximation partitions the last N values of the stream, $[0, \ldots, N-1]$, into $n - i$ segments. If A_i



(a) Multi-resolution approximations for a window of 16 most recent values

(b) Multi-resolution approximation tree prototype

Figure 1: Proposed solution

denotes the approximation at level i, then $A_{n-1} = [0 \ldots 2^n - 1]$ and

$$A_i = A_{i+1} - \{[0, \ldots, 2^{i+2} - 1]\} \cup \{[0, \ldots, 2^{i+1} - 1], [2^{i+1}, \ldots, 2^{i+2} - 1]\} \quad 0 \leq i \leq n - 2 \tag{1}$$

As evident from the above recursion, a level $i + 1$ approximation can be obtained from a level i approximation by combining the first two segments of the approximation into a single segment. This combination can be achieved computationally using wavelets.

Consider the tree in Figure 1(b). At every level of the tree, there are 3 nodes to keep the corresponding approximations, *Left Node* (L), *Shift Node* (S) and *Right Node* (R) from left to right. The approximation stored at the *Right Node* will be shifted to the *Left Node* after some time units to represent an older approximation. *Shift Node* acts as an intermediary in this process. In general, L_i,S_i and R_i denote *Left Node*, *Shift Node* and *Right Node* at level i respectively.

We call our structure SWAT (Stream Summarization using Wavelet-based Approximation Tree). A SWAT approximating N values has height equal to $\log N$. To compute the approximations, we can use any of the wavelet bases such as Haar, Daubechies, Coiflets, Symlets and Meyer etc. [13]. Furthermore, within a given wavelet basis, we can use any number of wavelet coefficients to approximate the signal. The wavelet coefficients of a node v at level l are obtained by applying a forward transformation on the coefficients of v's two children at level $l - 1$. For example in Figure 1(b), the coefficients stored in R_2 are obtained by applying a forward transformation on coefficients for $[0, \ldots, 3]$ and $[4, \ldots, 7]$ stored in R_1 and L_1 respectively. Alternatively, we can think of the coefficients at level l as being obtained by $l + 1$ forward transformations on the data values $[\triangle, \ldots, \triangle + 2^{l+1} - 1]$, where $\triangle$ is the lower boundary of a level l node at some point in time. In this way, the coefficients stored in L_2 are the result of applying 3 discrete wavelet transformation on $[8, \ldots, 15]$ if level 2 is up to date. This is explained in more detail in Section 2.3.

In the rest of the paper, we will assume that Haar wavelets are being used and that a single coefficient (representing the average) is being maintained.

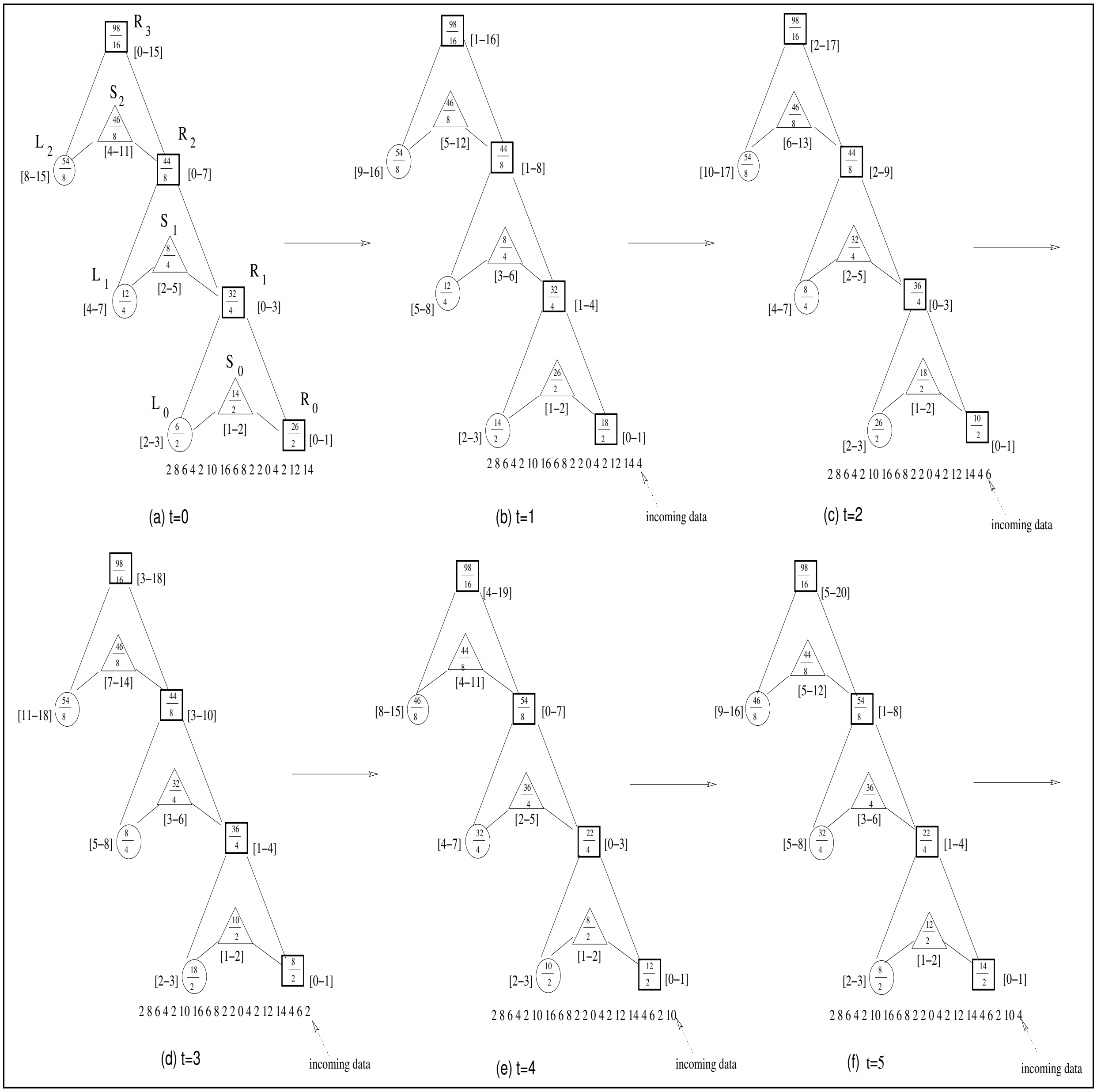


Figure 2: Execution trace on SWAT with a sliding window $N = 16$ for 5 new data arrivals

2.3 Updating the tree

As new data arrives, the window for the last N elements changes and the approximations in the SWAT need to be recomputed. We design the update algorithm so that the approximations in the tree are updated at different rates depending on the level of a node. Level 0 nodes are updated every time unit, level 1 nodes are updated every 2 time units, and generally level i nodes are updated every 2^i time units. This delayed update implies that the current approximation stored at a non-updated node summarizes a segment from the past. For example, a level 2 node L_2 summarizes $[8, \ldots, 15]$ initially. After 1 time unit, L_2 summarizes $[9, \ldots, 16]$, after 2 time units, it summarizes $[10, \ldots, 17]$, and after 3 time units, it summarizes $[11, \ldots, 18]$. It gets updated after 4 time units to summarize the interval $[8, \ldots, 15]$ again. Hereon, we will use notations $[8, \ldots, 15]$ and $[8 - 15]$ interchangeably.

In order to motivate the update algorithm, consider the execution trace shown in Figure 2 for $N = 16$. At $t = 0$, every node is up-to-date as shown in Figure 2(a). At $t = 1$, a new data value, 4, arrives. At $t = 1$, L_0 gets the summary stored in S_0, $14/2$, and S_0 gets $26/2$ from R_0. R_0 computes the average of 14 and 4. The average $18/2$ is stored in R_0. All nodes at higher levels are shifted up by 1 time unit. For example, L_2 now stores an approximation to $[9 - 16]$ instead of $[8 - 15]$. Figure 2(b) shows the resulting tree. At $t = 2$, 6 arrives. At level 0, L_0 gets $26/2$ from S_0, and S_0 gets $18/2$ from R_0. The new average of $[0, 1]$, $10/2$, is stored in R_0. At level 1, L_1 gets $8/4$ from S_1, and S_1 gets $32/4$ from R_1. Lastly, R_1 computes and stores the average of R_0 and L_0, which is $36/4$. Figure 2(c) shows the resulting tree. Figures 2(d),(e) and (f) show the resulting trees after the arrival of $2, 10$, and 4.

The complete algorithm for incremental computation is shown in Figure 3(a). For each incoming datum, we first determine the levels that will be updated. Call the highest level that will be updated max_level. We traverse the tree starting from level 0, and update the nodes at each and every level including max_level. During the update at level l, L_l gets the old summary in S_l, and then S_l gets the old summary in R_l. Finally, R_l gets its summary by applying a forward wavelet transformation on the summaries stored in R_{l-1} and L_{l-1}.

procedure Update_Tree
  max_level := highest propagation level for incoming data;
  $l := 0$;
  while $l \le max\_level$ **do**
    $contents(L_l) := contents(S_l)$;
    $contents(S_l) := contents(R_l)$;
    $contents(R_l) := DWT(R_{l-1}, L_{l-1})$; [a]
    $l := l + 1$;
  end while
end procedure

[a] R_{-1} and L_{-1} are data values d_0 and d_1 respectively.

(a) Algorithm for updating tree

procedure Query_Handler(Q:query)
  S := Sorted order of nodes in tree in lexicographic order, first on level number and then in the order $R \to S \to L$;
  while there exist uncovered entries in Q **do**
    $node := next(S)$;
    $level$:= level of $node$;
    if node covers any new entries in Q **then**
      mark the corresponding entries in Q as covered;
      $V := V \cup \{(node, level)\}$;
    end if
  end while
  for each pair $(node, level)$ in V **do**
    $signal :=$
      $inverse\_DWT(node.summary, level)$;
    use $signal$ to reconstruct the entries covered by $node$;
  end for
end procedure

(b) Algorithm for answering queries

Figure 3: Algorithms for maintaining the summaries and evaluating queries

If the entire data stream (and not just the last N values) is of interest, then the number of levels of the approximation tree will grow logarithmically with the size of the stream.

2.4 Answering queries

SWAT can be used to answer point queries, range queries, and inner product queries regarding last N values of the stream. A point query is a simple inner product query with one data value and a corresponding weight equal to 1. To answer a point query, it takes $O(1)$ time to find the node that approximates the point. Applying inverse transformation on the coefficients to approximate the signal and extract the point value takes $O(\log N)$ time. Range queries are specified using a point value p, a radius ϵ, and a time interval $[t_{start} - t_{end}]$, and ask for all values that lie within the specified time and value ranges. Thus, range queries induce a rectangle in time-value space with lower coordinates $(t_{end}, p - \epsilon)$ and with higher coordinates $(t_{start}, p + \epsilon)$. Our approximation tree induces a step function for a data stream in time-value space. The points that lie within the intersection of the rectangle and the step function are the points that satisfy the range query. This involves $O(\log N)$ approximations to be inspected, and $O(\log^2 N)$ time for applying inverse transformations on these approximations to extract the signal values.

A simple way to answer an inner product query would be to transform it into a number of point queries, and compute the weighted sum. However this involves $O(M \log N)$ operations, where M is the length of the query, and can become inefficient. In order to answer an inner product query more efficiently using $O(M + \log^2 N)$ operations, we adopt a different approach. We partition the query into at most $3 \log N$ pieces, one for every node in the tree, and compute the inverse transforms one piece at a time. We first illustrate this through an example.

Consider SWAT in Figure 2(d). Let Q be the triplet $([0, 3, 8, 13], [10, 8, 4, 1], 50)$. We search the tree from the smallest level onwards and the nodes at the same level in the order, $R \rightarrow S \rightarrow L$. We examine if a node stores the approximation for a data value in the query. We build a set of nodes V that will be used to answer the query. V is empty initially. We start at level 0. Since R_0 approximates $[0 - 1]$ and the query is interested in the 0^{th} item, we add R_0 to the set of nodes V. Since S_0 approximates $[1 - 2]$, which does not include a new item in the query, we skip it. Since L_0 approximates $[2 - 3]$ and the query is interested in the 3^{rd} item, we add L_0 to V. We now move to level 1. We skip nodes R_1 and S_1, since they do not contribute the answer set. Since node L_1 approximates $[5 - 8]$ and the query is interested in the 8^{th} item, we add L_1 to V. We skip R_2. Since node S_2 approximates $[7 - 14]$ and the query is interested in the 13^{th} item, we add S_2 to V. We stop at this point, since all values in Q are approximated by $V = \{R_0, L_0, L_1, S_2\}$. Now, we perform inverse transformation on the coefficients on the nodes in V to get the desired signal values. Since R_0 and L_0 are level 0 nodes, we can reconstruct the partial signal for $[0, 1, 2, 3]$ in a single inverse transformation, Since L_1 is a level 1 node, we reconstruct [5,6,7,8] in two inverse transformations. Finally, S_2 requires three inverse transformations to reconstruct $[7, 8, 9, 10, 11, 12, 13, 14]$. In general, V contains at most $3 \log N$ nodes (total number of nodes in SWAT). Since in the worst case, we need $O(\log N)$ inverse transformations to reconstruct a signal, it takes $O((\log N)^2)$ operations to reconstruct the approximate signal. Once the approximate signal has been constructed, its inner product can be computed using the weight vector in $O(M)$ time.

The complete algorithm for answering queries is shown in Figure 3(b). We traverse all the nodes of the tree in order from lowest to highest, and then nodes at the same level in the order $R \rightarrow S \rightarrow L$. For each node, we examine to see if it covers any new entries in the query Q. If so, the node is added to the set V along with its level. Once all the entries in Q have been covered, we carry out an inverse transformation to get the approximate values of the signals corresponding to the entries. For a node at level l, the inverse transform is applied $l + 1$ times and at each step a zero vector is used as the detail coefficient.

2.5 Number of levels

A sliding window of size N leads to an approximation tree of $\log N$ levels. The space complexity of this structure can be reduced by maintaining the approximations for only the top $k < \log N$ levels. Such a reduced space approximation can still answer all the queries, however with a larger amount of error. We will study this space-error tradeoff in Section 2.7.

2.6 Complexity analysis of SWAT

Let T denote a k-coefficient SWAT over a window of size N. We compute the space complexity, update complexity and error bounds for this tree.

Tree T has $3\log N - 2$ nodes. Each node keeps k wavelet coefficients. Therefore, the space complexity of our scheme is $O((3\log N - 2)k) = O(k \log N)$.

We compute the update complexity as the number of operations needed to update the tree during one complete cycle of N new updates. There are at most 3 nodes at each level. A level l node is updated after every 2^l data arrivals. Thus, a level l node will be updated $N/2^l$ times in one cycle. Each update takes $O(1)$ time. Therefore, the total update complexity is $\sum_{l=0}^{\log N - 1}(3\, O(k)\frac{N}{2^l}) = O(kN)$.

For error bound analysis, we assume that each incoming data point differs by ϵ from the previous value, i.e., $\forall\ i,\ 0 \le i \le N-1\ \ d_{i+1} - d_i = \epsilon$. Let $Q = (I, W, \delta)$ be an exponential inner product query, where $W = [1, \frac{1}{2}, , ..., \frac{1}{2^{M-1}}]$. Assume that to answer Q, we use R_2 at level 2 in Figure 2(a). R_2 approximates $\{d_0, \dots, d_7\}$ using their average. When consecutive data items differ by ϵ, $\forall\ i,\ 0 \le i \le 7\quad d_i = d_0 + i\epsilon$, and the average of $\{d_0, \dots, d_7\}$ is $(\sum_{i=0}^{7} d_0 + i\epsilon)/8 = d_0 + 7\epsilon/2$. The error incurred for data item d_i will be $(d_0 + i\epsilon) - (d_0 + 7\epsilon/2)$. This implies that d_0 and d_7 are approximated with error $3\epsilon + \epsilon/2$, d_1 and d_6 with error $2\epsilon + \epsilon/2$, d_2 and d_5 with error $\epsilon + \epsilon/2$, and d_3 and d_4 with error $\epsilon/2$. When we consider the query, the error incurred for each data value approximated will be weighted by the corresponding coefficient in the weight vector. Therefore, the total error will be a weighted sum of these individual errors. Using this insight, the weighted error incurred at any level l is:

$$\begin{aligned}
&\le \frac{1}{2^l}\sum_{i=0}^{2^l-1}\left(\frac{1}{2^i}\left((2^l-1-i)\epsilon + \frac{\epsilon}{2}\right) + \frac{1}{2^{2^l+i}}(i\epsilon + \frac{\epsilon}{2})\right) = \frac{1}{2^l}\sum_{i=0}^{2^l-1}\left(\frac{\epsilon}{2^i}\left(\frac{(2^{l+1}-(2+2i)+1)}{2}\right) + \frac{\epsilon}{2^{2^l+i}}\frac{2i+1}{2}\right) \\
&= \frac{\epsilon}{2^l}\sum_{i=0}^{2^l-1}\left(\frac{1}{2^{i+1}}(2^{l+1}-1-2i) + \frac{2i+1}{2^{i+1}}\frac{1}{2^{2^l}}\right) = \frac{\epsilon}{2^l}\sum_{i=0}^{2^l-1}\frac{1}{2^{i+1}}\left(2^{l+1} - (2i+1-\frac{2i+1}{2^{2^l}})\right) \\
&= \frac{\epsilon}{2^l}\sum_{i=0}^{2^l-1}\frac{2^{l+1}}{2^{i+1}} - \frac{\epsilon}{2^l}\sum_{i=0}^{2^l-1}\frac{2i+1}{2^{i+1}}(1-\frac{1}{2^{2^l}}) = 2\epsilon\sum_{i=0}^{2^l-1}\frac{1}{2^{i+1}} - \frac{\epsilon}{2^l}(1-\frac{1}{2^{2^l}})\sum_{i=0}^{2^l-1}\frac{2i+1}{2^{i+1}} \le 2\epsilon
\end{aligned}$$

Using this information, we can express the total error incurred to answer the exponential inner product query of length $O(M)$ as:

$$\sum_{l=0}^{\lceil \log M \rceil} 2\epsilon = O(\epsilon \log M). \tag{2}$$

Let $Q = (I, W, \delta)$ be a linear inner product query, where $W = [\frac{M}{M}, \frac{M-1}{M}, , ..., \frac{1}{M}]$. The weighted error incurred at any level l for answering this query is:

$$\begin{aligned}
&\le \sum_{i=0}^{2^l-1}\left(\frac{M-(2^l+i+1)}{M}((2^l-(i+1))\epsilon + \frac{\epsilon}{2})\right) + \sum_{i=0}^{2^l-1}\left(\frac{M-(i+1)}{M}(i\epsilon + \frac{\epsilon}{2})\right) \\
&\le \frac{\epsilon}{M}\sum_{i=0}^{2^l-1}(2^l M - 3 * 2^{l-1}) = \frac{\epsilon}{M}(2^l M - 3*2^{l-1})2^l = \frac{\epsilon}{M}(M-3/2)4^l \le 4^l\epsilon
\end{aligned}$$

Therefore, the total error for the linear inner product query of length $O(M)$ is:

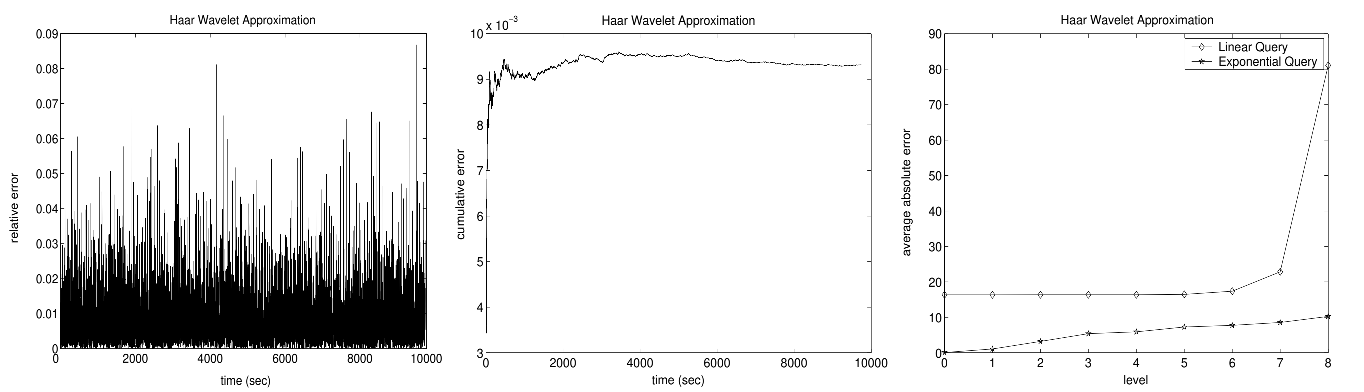
$$\sum_{l=0}^{\lceil \log M \rceil} 4^l \epsilon = O(\epsilon M^2). \tag{3}$$

2.7 Experimental results

In this section we present the performance results of our wavelet-based technique. We first examine the amount of relative error and absolute error of our algorithm under various conditions. After that, we compare the error bounds and the processing time of this technique to the recent histogram-based algorithm by Guha and Koudas [8].

We use synthetic data and real-world data in our experiments. *Synthetic* data is obtained by a uniformly distributed random number generator. The range of data values is $[0, 100]$. *Real* data is the daily measurement of the maximum temperature for the city of Santa Barbara, CA from 1994 to 2001 [3]. Its size is 3K.

To study our technique empirically, we built a discrete event simulator of an environment with a single data stream. We use a variety of window sizes, $N \in \{256, 512, 1024\}$. The data arrival period, T_d, is 1 second. The stream is queried repeatedly with a query period, $T_q = 1$ second, unless stated otherwise. At each query point, we either execute the same inner product query repeatedly (*fixed* query mode) or a new randomly chosen inner product query (*random* query mode). In the fixed query mode, we execute a query over the most recent values repeatedly whereas in the random query mode, we choose arbitrary data points repeatedly.

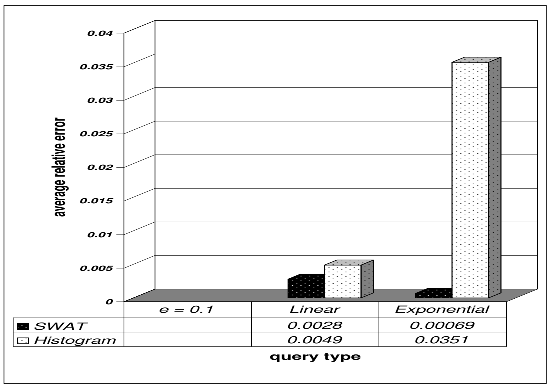


(a) $N = 256$, synthetic data (b) $N = 256$, synthetic data (c) $N = 512$, real data

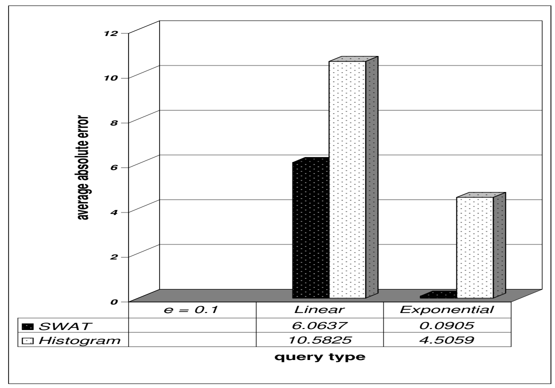
Figure 4: **fixed query mode**. (a) relative error for exponential inner product queries , (b) cumulative error for exponential inner product queries, and (c) average absolute error under varying number of levels for different types of queries

Our first set of experiments observes 10K incoming points and executes the same exponential inner product query at each arrival (fixed mode). Figure 4(a) shows the relative error for a window size $N = 256$. As can be noted, the error values exhibit a periodic behavior. This is due to our update scheme in which approximations at the upper levels in the tree can diverge for short durations. Figure 4(b) depicts another interpretation of the same results. The cumulative error at time t measures the average of the relative errors observed in queries at times $0, 1, \ldots, t$. As can be seen, the cumulative error is quite small, around 0.01.

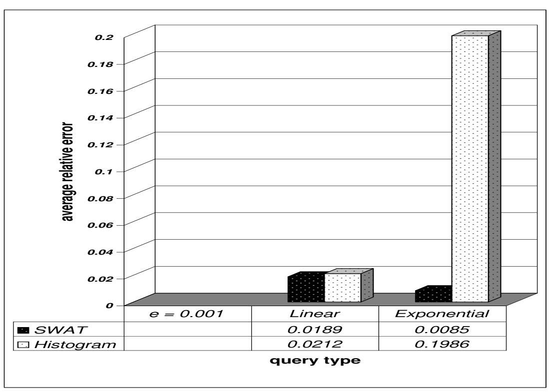
In the next experiment, we change the number of levels of a stream and measure the amount of average absolute error for a window size of 512. Figure 4(c) reports the results for an exponential and a linear inner product query. When we increase the degree of approximation, we observe an exponential increase in the average absolute error in case of the linear query, whereas the increase is linear in the exponential query case. This can be explained as follows. The error incurred for the same element *increases exponentially* as we increase the resolution level. This behavior is preserved in the final answer for the linear inner product query. However, the behavior is canceled out, when the weight coefficients *decrease exponentially* as in the case of the exponential inner product query.



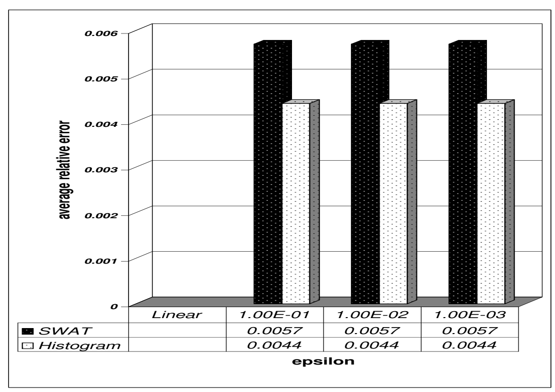
(a) Real data, $\epsilon = 0.1$, **fixed query mode**



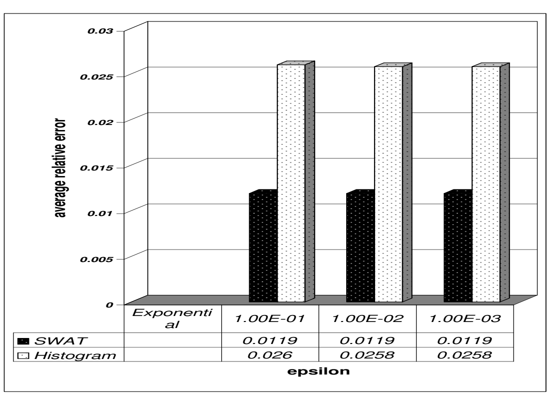
(b) Real data, $\epsilon = 0.1$, **fixed query mode**



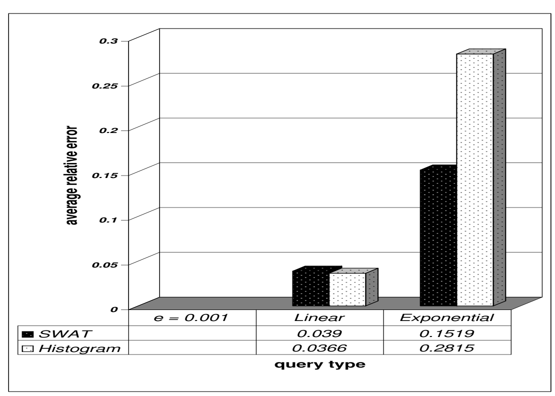
(c) Synthetic data, $\epsilon = 0.001$, **fixed query mode**



(d) Real data, linear query, **random query mode**



(e) Real data, exponential query, **random query mode**



(f) Synthetic data, $\epsilon = 0.001$, **random query mode**

Figure 5: (a)–(c) error vs query type in **fixed query mode**. (d)–(f) error vs query type/ϵ in **random query mode**

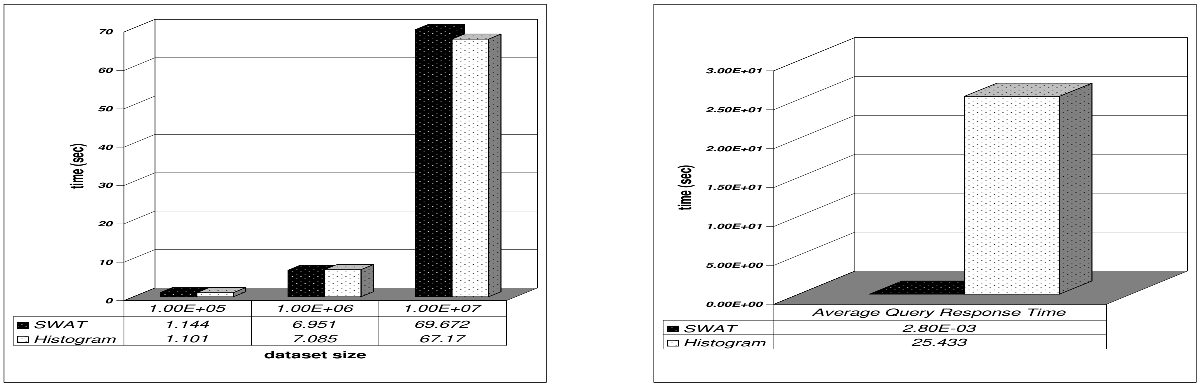
In our second set of experiments, we compare the error induced by our technique with the histogram-based technique proposed by Guha and Koudas [8], the most recent sliding-window algorithm proposed in the literature. In the rest of the paper, we refer to this algorithm by *Histogram*. The window size N is set to 1024. A new data arrives every second, and a fixed query is evaluated every second. We first observe 1K incoming data to warm up the summary structures of both techniques. The number of approximations that SWAT keeps is $3 \log N$, that is, approximately 30 in this case. Therefore, we set the bucket size $B = 30$ for *Histogram*. The histogram-based algorithm changes the quality of the approximation based an input parameter ϵ. As the error bound ϵ decreases, the quality of the summary becomes better; however, this is achieved at a higher cost of query execution. In contrast, the performance of SWAT does not depend on ϵ. We work with different values of $\epsilon \in \{0.1, 0.01, 0.001\}$.

First, we compare the two techniques, when the same inner product query, exponential or linear, is executed at each arrival (fixed query mode). We set ϵ to 0.1, and use real data in this experiment. Our technique outperforms *Histogram* for both type of queries as shown in Figure 5(a). The average relative error of our technique for the linear inner product query is 0.0028, while *Histogram* incurs an average relative error of 0.0049. The approximation quality that our scheme offers is two times better in this case. The improvement is in two orders of magnitude for the exponential inner product query. SWAT offers an average relative error of 0.00069, which is 50 times better than 0.0351, the error offered by *Histogram*. Figure 5(b) shows another interpretation of the same results. We would like to note that the average absolute error for linear inner product queries is less than the average absolute error for exponential inner product queries. This result is consistent with our error bound analysis in Section 2.6.

We performed a similar experiment on synthetic data as well, with ϵ set to 0.001. The large deviations in the underlying synthetic data distribution incur error larger in magnitude than error incurred on real data, which has small deviations. This is illustrated in Figure 5(c). Our technique again outperformed *Histogram* for both query types. The approximation quality for the exponential inner product query improves by a factor of 25 times. There is not much improvement for the linear inner product query.

Next, we present the results for inner product queries in the random query mode. Figure 5(d) shows the average relative error for different values of ϵ for linear queries. SWAT does slightly worse than *Histogram*. This is because our approximation scheme is designed for queries where more recent values in the data stream are of a higher interest and linear random queries do not fit this model. However, for exponential inner product queries our technique outperforms *Histogram* as expected. This is shown in Figure 5(e). The average relative error under our technique is 0.0119. The error of *Histogram* varies between 0.026 for (for $\epsilon = 0.1$) and 0.0258 (for $\epsilon = 0.001$).

We compared the performance of the techniques on synthetic data in the random query mode as well. We set ϵ to 0.001. Figure 5(f) shows the results. For linear inner product queries, the average relative error of our technique is 0.039, whereas the error incurred by *Histogram* is 0.0366. For exponential inner product queries, the average relative error of SWAT is 0.1519. The approximation quality of our technique is two times the quality offered by *Histogram*, which has the average relative error of 0.2815. The error incurred on synthetic data under both techniques is much larger in magnitude than the error on real data because of the large deviations in the underlying data distribution.



(a) Maintenance time comparison

(b) Average query response time comparison

Figure 6: Running time comparison of SWAT and *Histogram* Algorithm

Next, we compare the maintenance time of the techniques. This corresponds to the time required to incrementally maintain summary structures as the entire dataset is observed. No queries are executed in this experiment. In case of SWAT, the summaries are updated with each incoming data point. On the other hand, the *Histogram* technique computes only the sum and the squared sum with every arrival; the rest of the summary is computed at every query. We used three synthetic datasets, one of size 100K, another of size 1M, and a last one of size 10M in this experiment. Figure 6(a) shows the performance of the two techniques. The maintenance times of the techniques are very similar.

Finally, we compare the query response time of the techniques. We execute 100 uniformly generated exponential inner product queries, and compute the average response times of both techniques. We set ϵ to 0.1. *Histogram* can output an ϵ-approximate B-histogram of the previous N points in time $O((B^3/\epsilon^2)\log^3 N)$ [8]. SWAT computes a summary of the previous N points in time $O(\log^2 N)$. In time $O(N)$, both techniques output the answer to the query of length $O(N)$ using the computed summaries. For $B = 30$, $\epsilon = 0.1$ and $N = 1024$, SWAT is expected to be far superior than *Histogram*. As shown in Figure 6(b), SWAT has an average query response time of $2.8e - 3$ seconds. On the other hand, *Histogram* has an average query response time of 25.433 seconds. The speed-up by our technique is up to four orders of magnitude, which is consistent with our analysis.

Finally, we would like to note that the space requirement of our technique is $O(\log N)$ as compared to $O(N)$ required by the *Histogram* technique.

3 Adaptive Stream Replication in Large Networks

So far, we have considered a centralized model for streams in which the summary is maintained at a single central site and all queries are sent to this site. The central site can become a bottleneck if the queries are frequent. For example, consider the case of continuous queries. In this case, the central site will need to generate a set of streams, one for every continuous query. This can be expensive, especially in low-power or low-bandwidth environments such as sensor networks. In this section, we will examine the management of stream summaries across a distributed network.

In our model, there is one central cite S, the primary data source, where the data arrives. Clients across the network issue queries over the last N elements of the data stream. Each query has a precision requirement. When a client receives a query Q with $\delta(Q)$ as precision requirement, it first checks its cache to answer the query. If it can satisfy the query's precision requirement using the local cache, then it provides an answer to the query. Otherwise, it forwards the request towards S. If a summary cached at a client become invalid due to data updates, S has to send the new summary. Obviously, there is a trade-off between how often the server propagates the summaries (push), and how often a client requests the server (pull).

In order to capture the precision of a cached result, a client maintains a range for every cached value. In order to simplify the ensuing development, we assume that 1-coefficient approximation trees are being used along with the Haar wavelet transform. In this case, a client caches a range $[d_L, d_H]$ for value d, where $d_L \leq d \leq d_H$. The range $[d_L, d_H]$ is called an *approximation* for the data element d. In the general case of an arbitrary number of coefficients and a different wavelet transform, the client would maintain the desired number of coefficients and a range denoting the maximum deviation of the true value from that computed using inverse transform on the coefficients.

Table 1 shows the general directory structure for our scheme. Every row in the directory represents a data record with *window segment, range* and *subscription list* fields. Consider the first row of the directory, $\langle(0,1), [25,45], \{C_1, C_2\}\rangle$. By looking at the first entry, $(0,1)$, we see that this row contains approximation for the first two values in the data stream. The second entry states that every value in the segment $(0,1)$ lies in the range $[25,45]$. The third entry, namely the subscription list, tells us that an approximation to $(0,1)$ (same as $[25,45]$ or coarser) is cached at children C_1 and C_2. The window segments do not overlap and the approximation stored for a segment denotes the node's current approximation for it. The table has one row for every level (except level 0 which has two rows) of the approximation tree resulting in $O(\log N)$ rows.

Data		
window segment	data range	subscription list
(0,1)	[25,45]	C_1, C_2
(2,3)	[30,40]	C_2
(4,7)	[2,7]	C_2
(8,15)	[4,10]	C_2

Table 1: General directory structure of our scheme

Our stream replication algorithm is called SWAT-ASR (Adaptive Stream Replication). It is based on the adaptive data replication algorithm by Wolfson, Jajodia and Huang [16]. We first summarize the basic idea of their algorithm.

The replication scheme of an object is defined in [16] using a spanning tree. Each node satisfies its read requests locally using a cached value, if possible. Otherwise, the read request is forwarded to the parent of the node, which in turn either answers it using a cached value or forwards it again. In this way, a read request proceeds up the tree (and towards the source S) until it is satisfied. A write request updates the local value and all other cached values. The replication scheme at any time consists of a subtree R induced by processors which have a replica of the object. A leaf node of R is called an *R-fringe* node. An R node that is adjacent to a non-R node is called a $\overline{R}$*-neighbor*. The algorithm executes in phases. At the end of a phase, every *R-fringe* node u executes a *contraction* test: the number of reads at u, r, is compared with the number of remote writes, w, received during the phase. If $r < w$, u deletes its cached copy of the object. An $\overline{R}$-*neighbor* node u executes an *expansion* test for every adjacent node v not in R at the end of a phase. It compares r, the total number of reads issued by v with w, the number of writes issued by all nodes except v during the phase. If $w < r$, u sends a copy of the object to v and v joins the replication scheme. (The algorithm in [16] also includes a *switch* test at the end of a phase. This is not needed for our setting since the source is always a member of the replication scheme.) The general idea of the above approach is that read requests lead to an expansion and write requests lead to a contraction of the replication scheme. We base our replication of stream summaries on the same idea.

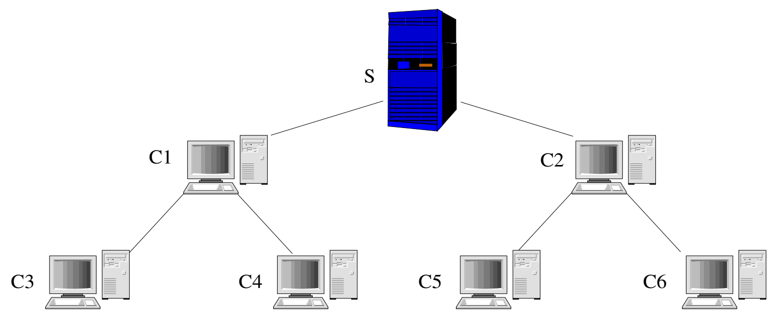


Figure 7: Example system topology

Consider the network topology shown in Figure 7. Assume that S has the directory shown in Table 1. Consider the following global execution scenario. A point query $Q_0([3],[1],20)$ arrives at C_3. Since there are no approximations in the local cache, Q_0 will not be satisfied. C_3 forwards Q_0 to its parent C_1. Client C_1 has not cached any approximations either, therefore it forwards Q_0 to S. Source S satisfies the query using the approximation value $[30,40]$ stored in its directory for segment $(2,3)$, marks C_1 as being "interested" for $(2,3)$, and increments $read\_count_{(2,3),C_1}$ to 1. Assume that the current phase ends. During the current phase, S has not received any updates on $(2,3)$, therefore $write\_count_{(2,3)}$ is 0. S executes an *expansion* test at the end of the phase. Since $write\_count_{(2,3)} < read\_count_{(2,3),C_1}$, S sends a copy of the approximation $[30,40]$ for segment $(2,3)$ to C_1, and adds C_1 to the subscription list. The row for the window segment $(2,3)$ in the directory at server S is now $\langle(2,3),[30,40],\{C_1,C_2\}\rangle$.

Assume that during the next phase, C_3 sends the same query three times. C_1 answers all the queries successfully, and sets $read\_count_{(2,3),C_3}$ to 3. At the end of the second phase, C_1 sends $\langle(2,3),[30,40],\emptyset\rangle$ to C_3 after completing the *expansion* test. At this point, site C_1 has the row $\langle(2,3),[30,40],\{C_3\}\rangle$ in its directory.

During the next phase, incoming data updates the approximation for segment $(2,3)$ to $[32,38]$ at S. Source S does not propagate $[32,38]$ to the children subscribed to $(2,3)$ since the old approximation $[30,40]$ encloses the new approximation $[32,38]$. Now, a point query $Q_1([3],[1],8)$ is issued at C_1 four times, and query Q_0 is issued at C_3. C_1 cannot satisfy Q_1, since the precision offered to 3^{rd} value of the stream $(40-30=10)$ is less than the required precision, 8. Therefore, it forwards all Q_1 queries to S, and S sets $read\_count_{(2,3),C_1}$ to 4. At the end of the phase, S executes an *expansion* test, and finds that $write\_count_{(2,3)} < read\_count_{(2,3),C_1}$. Since C_1 is already in the subscription list, S sends the new approximation $[32, 38]$ to C_1. When C_1 receives the update $[32, 38]$ on $(2, 3)$, it checks its directory and finds that the old approximation $[30, 40]$ on $(2, 3)$ encloses the new approximation $[32, 38]$. Therefore, no updates are propagated to the children. All Q_0 queries issued at C_3 are satisfied locally. Therefore, no changes are made to the approximation in C_3. At this point, $\langle(2, 3), [32, 38], \{C_1, C_2\}\rangle$, $\langle(2, 3), [32, 38], \{C_3\}\rangle$, and $\langle(2, 3), [30, 40], \emptyset\rangle$ are the corresponding rows in the directories at S, C_1 and C_3 respectively.

During the next phase, incoming data updates the approximation for segment $(2, 3)$ to $[34, 35]$ at S. No updates are propagated following the same reasoning explained above. Assume that all queries issued locally at the clients are satisfied by the local replicas and no queries are forwarded. The tests at the end of the phase do not result in any changes. The resulting rows at S, C_1 and C_3 are $\langle(2, 3), [34, 35], \{C_1, C_2\}\rangle$, $\langle(2, 3), [32, 38], \{C_3\}\rangle$, and $\langle(2, 3), [30, 40], \emptyset\rangle$. As can be seen, the precision offered for the same stream segment $(2, 3)$ decreases as we traverse down the replication tree.

Our stream caching algorithm partitions the window into segments and runs the replication algorithm for each segment independently. The replication scheme for a segment is a subtree of the spanning tree of the graph. Each node in the replication scheme stores an approximation to the segment. The precision for a segment either remains the same or decreases as we descend down the tree. Every read request that a node cannot satisfy locally (precision of cached value not being adequate) is forwarded to the parent. At the same time, a node in the replication scheme is responsible for ensuring the cached approximations at its descendents. Since only the source node issues updates, the *expansion* and *contraction* tests simply compare the number of local reads with the number of updates. In addition to Table 1, each node u in the replication scheme also maintains enough information to perform the *expansion* and *contraction* tests at the end of a phase. This consists of an *interested* list storing the children that have asked queries over the segment, but not subscribed to the segment, the number of updates, the local read count, and a read count for every adjacent node v in one of the lists, namely *subscribed* and *interested.*.

The subroutine for responding to different messages is shown in Figure 8(a). If the message updates the approximation over segment S to range R then this update is made and then the new approximation R is compared to the existing approximation $oldR$. If $oldR$ does not enclose R then the new approximation is sent to all subscribing children. If the message is a query then it is decomposed into sub-queries over different data segments. If the resulting error from the different approximations is within the specified error bound then the read count is incremented. Otherwise, the query is forwarded to the parent.

At the end of each phase, *expansion* and *contraction* tests are executed for each stream segment S. If a node u is an *R-fringe* node for a segment S, it executes the contraction test for the segment S: if the read count r for S is less than the write count w for S, then the local copy of the segment S is decached. On the other hand, if u is an $\overline{R}$*-neighbor* node for segment S, then an expansion test is performed for each adjoining node v. First, consider the case in which v has subscribed to the segment S but its approximation is not adequate. In this case, u sends the current approximation to v. The other case is one in which v is interested in segment S. If the read count of v exceeds the write count, then a replica is sent to v. The read and write counts are reset at the end of each phase. The complete algorithm is shown in Figure 8(b).

procedure Message_Handler(m:message)
  if m is $update(S : segment, R : range)$ **then**
    $oldR$:= existing approximation;
    existing approximation := R;
    if $oldR$ does not enclose R **then**
      $S.writeCount :=$
        $S.writeCount + 1$;
      for each $client$ in $S.subscribed$ **do**
        send $update(S, R)$ to C;
      end for
    end if
  else if m is $query$ from node v **then**
    if $query$ is satisfied locally **then**
      if v is not in any of the lists **then**
        add v in $S.interested$;
      end if
      $v.readCount :=$
        $v.readCount + 1$;
    else
      forward $query$ to parent;
    end if
  end if
end procedure

(a) Algorithm for handling messages over stream segment S at node u during a phase

procedure Phase_End
  if u is an *R-fringe* node for stream segment S **then**
    if $u.readCount < S.writeCount$ **then**
      decache segment S;
    end if
  else if u is an $\overline{R}$*-neighbor* node for stream segment S **then**
    for each v in $S.subscribed$ **do**
      if $S.writeCount < v.readCount$ **then**
        send *update* message to v;
      end if
    end for
    for each v in $S.interested$ **do**
      remove v from $S.interested$;
      if $S.writeCount < v.readCount$ **then**
        add v to $S.subscribed$;
        send *insert* message to v;
      end if
    end for
  end if
  reset read and write counts;
end procedure

(b) Actions taken at the end of a phase for stream segment S at node u

Figure 8: Adaptive stream replication algorithms

4 Competing Techniques for Adaptive Stream Replication

In this section, we discuss two competing techniques for managing approximations in client-server systems. The first called *Divergence Caching* [11] models the read and write requests as originating from Poisson processes, and computes the best way to transmit updates. Since this technique was not proposed for managing approximations per se, we first adapt it to work in our setting. The second technique called *Adaptive Precision Setting* [15] has been explicitly proposed for managing approximations and requires no adaptation.

4.1 Divergence caching

Divergence Caching is a mechanism to reduce the number of object transmissions in Client-Server architectures [11]. The main idea behind the algorithm is to compute the optimal refresh rate using a window of past reads and writes on an object assuming Poisson processes for reads and writes. A read operation defines its tolerance as how recent a version of the object is required. A read tolerance of one means that the most recent version is needed; in general a read tolerance of k implies that any of the last k versions of the object will suffice. The authors first consider a static version of the problem in which the write rate and the read rates are known for each level of tolerance. They compute the expected cost for each refresh rate (the rate at which updates are transmitted from the server to the client), and the refresh rate with the minimum expected cost is chosen by the server. Later they generalize to the case when the read and write rates are not known a priori. Here, a finite window is used to compute the likely read and write rates, and the minimum cost refresh rate is found as before.

We adapt the central idea of this algorithm to the problem of maintaining approximations. We define tolerance of a client in terms of the desired precision rather than how recent the value is. Since a read miss for a data d is now caused when the current precision (width of interval) is inadequate, we choose the refresh rate to mean $d_H - d_L$, the width of the interval for d. The formulas used in the original algorithm to find the optimal refresh rate have been adapted to work under our model as follows.

- Every object has a *refresh rate*.
- Reads issued by clients have *tolerance* values.
- If *tolerance* $<$ *refresh rate*, the read is successful. Otherwise, the read is forwarded to server. The current value of the object is sent to client with a newly calculated *refresh rate*.
- When the server performs a write on the object, the new value is transmitted only if it is not within the *refresh rate*. This process is called *unsolicited refresh*.
- Each message containing the data object costs 1.
- Each control message costs w.
- Let M be the maximum possible range.
- Let $k = (d_H - d_L)$ be the refresh rate (width) for an object with range approximation $[d_L, d_H]$.
- For $k = \infty$, we pay $(1 + w)$ for every read, and nothing for writes.
- For $k = 0$, we pay 1 for every write, but nothing for reads.
- We pay for reads with tolerance less than k, and possibly for writes. These requests are called *relevant*. Reads with tolerance at least k are called *irrelevant*. All writes are considered relevant.
- λ_{r_t} is the average number of read requests with tolerance t per time unit. λ_w is the average number of writes per time unit.
- $r(k) = \sum_{t=0}^{k-1} \lambda_{r_t}$ is the intensity of Poisson process generating *relevant* reads.
- $\theta_k = \frac{\lambda_w}{\lambda_w + r(k)}$ is the probability that a relevant request is a write.
- The probability of an arbitrary request being relevant is $\frac{\lambda_w + r(k)}{\lambda_w + r(k) + \sum_{t=k}^{M-1} \lambda_{r_t}}$.
- For a relevant request that is a write, we pay with a probability $\frac{M-k}{M}$.
- A relevant request is a read with probability $1 - \theta_k$.
- The expected cost of an arbitrary relevant request is $(1 - \theta_k)(1 + w) + \frac{M-k}{M}$.
- The expected cost of an arbitrary request is

$$\frac{\lambda_w + r(k)}{\lambda_w + r(k) + \sum_{t=k}^{M-1} \lambda_{r_t}} \left((1 - \theta_k)(1 + w) + \frac{M-k}{M} \right) = \frac{r(k)(1 + w) + \frac{M-k}{M}(\lambda_w + r(k))}{\lambda_w + r(k) + \sum_{t=k}^{M-1} \lambda_{r_t}}$$

  where $(1 - \theta_k)(\lambda_w + r(k)) = r(k)$ by definition.

- The expected cost per unit time is the expected cost of an arbitrary request times the number of requests per unit time, which is the denominator in the equation above. This equals

$$\left\{\begin{array}{l} \lambda_w \text{ for } k = 0 \\ r(k)(1+w) + \frac{M-k}{M}(\lambda_w + r(k)) \text{ for } 1 \leq k \leq M-1 \\ (w+1)\sum_{j=0}^{M-1} \lambda_{r_j} \text{ for } k = M \end{array}\right\}$$

When the exact read and write rates are not known a priori, a window of past events is used to estimate them. The authors in [11] used a window of size 23; we use the same in our experiments.

4.2 Adaptive precision setting

Adaptive Precision Setting [15] is a parameterized algorithm for adjusting the precision of cached approximations in a client-server system. Caching exact values at a client incurs rapid updates when data values change rapidly, and not caching at all causes requests to be sent to the server every time the object is read. The algorithm works as follows.

- For every data value d, its range is approximated by an interval $[L, H]$, where $L \leq d \leq H$ and $W = H - L$.
- When data value d changes to d', if $d' \notin [L, H]$, then $[L', H']$ is sent to client. The width of the approximation is enlarged: $W \leq (H' - L')$. This process is called a *value initiated refresh*.
- When a query (read) Q is executed on d with $\delta(Q)$ as the precision requirement, and if $\delta(Q) < W$, then the query is forwarded to the server. The server in turn sends a newly computed approximation $[L', H']$ to the client. The width of the approximation is shrunk: $(H' - L') \leq W$. This process is called a *query initiated refresh*.

The above basic idea can be tuned using parameters to give different choices and likelihoods of interval enlargement and shrinkage. We used the algorithm with its recommended and appropriate settings ($\alpha = 1, \tau_\infty = \infty, \tau_0 = 2, p = 1$). Parameter α is the adaptivity parameter and dictates the amount of change in the interval width. Let $W' = H' - L'$ denote the width of the newly computed interval. If $\tau_\infty < W'$, then the interval is set to $(-\infty, \infty)$ (effectively deleting the cached value). If $W' < \tau_0$, then the interval is set to $[d, d]$ (exact caching). Parameter p is the cost factor, and is used to determine how often to enlarge and shrink the interval width. The refresh actions can be summarized as follows:

- *value initiated refresh*: with probability $min\{p, 1\}$, set $W' = W.(1 + \alpha)$,
- *query initiated refresh*: with probability $min\{1/p, 1\}$, set $W' = W/(1 + \alpha)$.

5 Performance Measurements for Adaptive Stream Replication

In this section, we compare our distributed stream summarization technique with the adapted version of Divergence caching [11] and Adaptive Precision Setting [15]. Our simulation environment consists of a server site processing a single data stream, and a number of clients asking linear inner product queries at regular intervals. We use a window size $N \in \{32, 64\}$. We run Divergence Caching and Adaptive Precision Settting caching algorithms independently for each data item in the window. We schedule periodic tasks to initiate data and query arrivals. The sizes of the queries and the specific data points of interest are chosen uniformly (random query mode). The system is allowed to warm up initially before measurements are made.

We denote Adaptive Precision Setting as *APS* and Divergence Caching as *DC* in figures. First, we compare the space complexity of the three schemes. Then, we compare their performance in a single client setting. Finally, we compare them in a multiple client setting.

5.1 Space complexity

In a single-client network, both Adaptive Precision and Divergence Caching maintain exactly one approximation for each d_i, where $0 \leq i \leq N-1$. However in SWAT-ASR, a single approximation is maintained for each stream segment and there are at most $\log N$ such segments. Thus, our space complexity is $O(\log N)$ as compared to the $O(N)$ complexity of the other two schemes.

In a multiple-client network of M clients, the number of approximations maintained by Adaptive Precision and Divergence Caching is $O(MN)$, whereas our technique maintains $O(M \log N)$ approximations.

5.2 Single client system

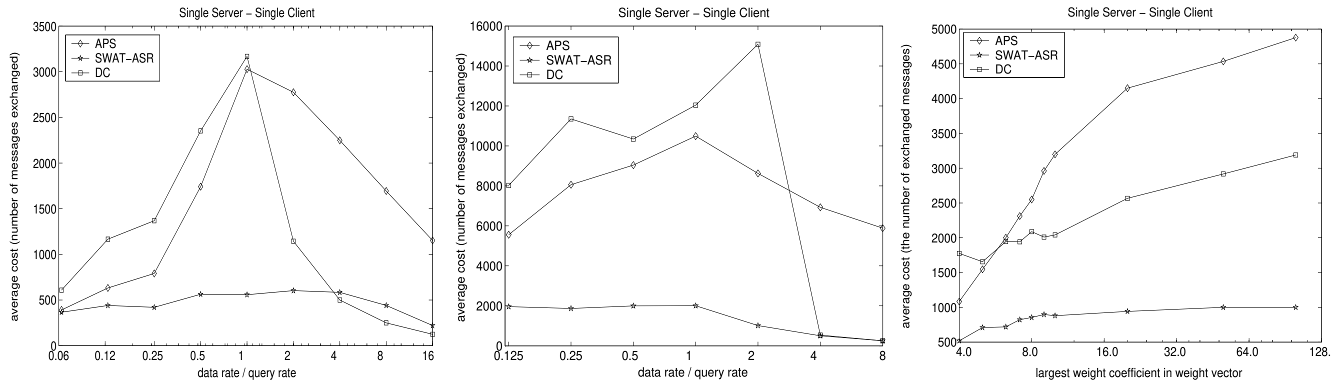
We first consider a system consisting of one server (S), and one client (C), and consider the effect of varying data and query rates and the precision requirement of queries. We want to measure how adaptive the techniques are in deciding whether to cache a value.

5.2.1 Varying data rate/query rate

Our first experiment is to analyze the behavior of the replication techniques when the ratio $\frac{data\ rate}{query\ rate}$ varies. As before, we denote the *query rate* by T_q and the *data rate* by T_d. The results for the weather dataset are shown in Figure 9(a). We measure the cost of an algorithm as the number of exchanged messages.

When $T_d < T_q$, it is better to cache approximations. In fact, all three algorithms achieve this. However, the number of approximations sent by the server in our case is much smaller since we cache stream segments instead of individual data values. This is the reason behind the improved performance of our technique.

When $T_d > T_q$, it is better not to cache. Divergence Caching and SWAT-ASR quickly choose not to cache since writes are far more than reads. The reason SWAT-ASR does not converge very quickly is that since intervals are for a stream segment, and since we are operating on real world data, approximations do not change rapidly even under high T_d. When we run a similar experiment with synthetic data, where interval changes are relatively more rapid compared to real data, then SWAT-ASR chooses not to cache quickly: Figure 9(b) shows clearly that for $T_q << T_d$, SWAT-ASR and Divergence Caching perform similar. However, Adaptive Precision Setting is a little slow in adapting because of the way it computes the next interval. It does converge eventually by choosing bigger intervals that approach the upper threshold τ_∞.

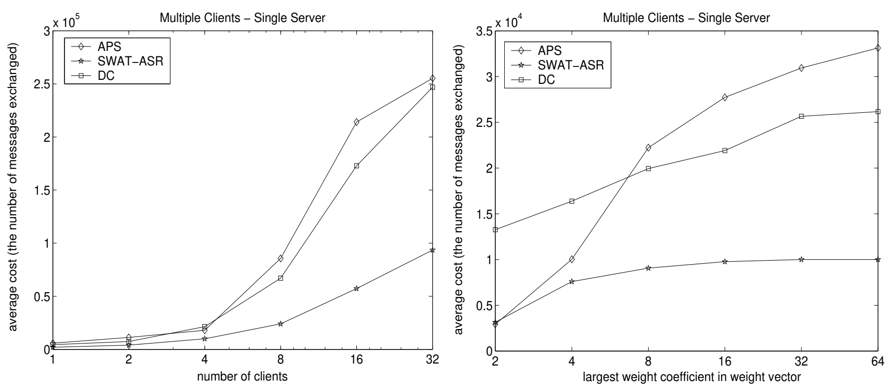


(a) Average cost vs data rate/query rate on weather dataset

(b) Average cost vs data rate/query rate on synthetic data

(c) Average cost vs precision on weather dataset

Figure 9: Single client experiments with a sliding window of size 32



(a) Average cost vs number of clients on weather dataset

(b) Average cost vs precision on synthetic data

Figure 10: Multiple client experiments on a sliding window of size 64

5.2.2 Varying precision requirements

Under fixed input rates ($T_q = 1\ sec,\ T_d = 2\ sec$), we varied the precision requirement of queries. We expect to observe a degrading performance when the precision increases since more number of messages need to be sent. The experimental results on real data are shown in Figure 9(c). SWAT-ASR has a performance gain of up to 5 times as compared to Adaptive Precision Setting and up to 4 times as compared to Divergence Caching. This is again due to the much smaller number of approximations used by our technique.

5.3 Multiple client system

Finally, we evaluate the algorithms for the case of multiple clients. We measure the effect of changing the number of clients and the precision of queries. We assume in the following experiments that a client can hold the $O(N)$ approximations needed for Divergence Caching and Adaptive Precision. When this is not the case, the cost of these schemes is going to be much higher since more read requests will be forwarded to the server. Our technique maintains only $O(\log N)$ approximations, and so space requirement is not as much of an issue.

5.3.1 Increasing number of clients

Our simulation topology consists of a complete binary tree with the source at the root. Each node asks queries with independent precision requirements. Figure 10(a) shows the results on the weather dataset. SWAT-ASR outperformed the other techniques on account of two reasons: first, it reduces the number of approximations by approximating stream segments instead of individual values, and second since its hierarchical structure allows multiple nodes to share the same approximation. Divergence caching sends up to 3 times more messages than the the number of messages sent by SWAT-ASR, while Adaptive Precision sends up to 4 times more messages.

5.3.2 Varying precision requirements

In this experiment, we varied the precision of queries and measured the cost of the three algorithms for 6-node client system with a binary tree topology. We chose the synthetic dataset for the experiments since the large variations in values make it difficult for SWAT-ASR to summarize a stream segment effectively. Figure 10(b) shows the results. As can be seen, SWAT-ASR is better than competing algorithms by a factor of 3-4 on account of its hierarchical structure.

6 Concluding Remarks

In this paper, we have presented a space and time efficient technique to summarize a data stream. Our solution addresses keeping summaries of the last N values of a data stream at multiple resolutions. We use *Discrete Wavelet Transform (DWT)* to compute the summaries. These summaries are used to answer point queries, range queries, and specifically inner product queries.

We have provided theoretical and experimental analysis showing the effectiveness of our technique. The amortized processing cost for each incoming data point is $O(1)$. For any query, our technique computes an approximate answer with precision logarithmically proportional to the length of the query and in polylogarithmic time. The number of approximations that our technique keeps for any stream is fixed, totally $O(\log N)$ approximations. However, a client can dynamically choose to keep different number of approximations depending on the precision requirement. The space usage of our technique is logarithmic, $O(\log N)$ in the size of the sliding window N. In experimental results, our technique's response time was four orders of magnitude better than current techniques. Furthermore, its approximation quality was up to 50 times better.

Later, we extended our technique to stream replication in large networks. The summaries computed at a central site are cached adaptively at the multiple clients. This adaptive scheme minimizes the total communication cost, the number of inter-site messages. In our experiments, our technique performed up to five times better in terms of message complexity than existing techniques. It also adapted well to varying read-write patterns.

Our future work will explore possible variations of the proposed technique in case of multiple streams. We plan to develop efficient techniques to find correlations over multiple data streams, and new indexing methods.

References

[1] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *In Proc. of the 1996 Annual ACM Symp. on Theory of Computing*, pages 20–29, 1996.

[2] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, pages 1–16, 2002.

[3] California Weather Database. http://www.ipm.ucdavis.edu/WEATHER.

[4] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. In *ACM Symposium on Discrete Algorithms*, pages 635–644, 2002.

[5] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi. Processing complex aggregate queries over data streams. In *ACM SIGMOD Conference*, pages 61–72, 2002.

[6] J. Gehrke, F. Korn, and D. Srivastava. On computing correlated aggregates over continual data streams. In *ACM SIGMOD Conference*, pages 13–24, 2001.

[7] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *The VLDB Journal*, pages 79–88, 2001.

[8] S. Guha and N. Koudas. Approximating a data stream for querying and estimation: Algorithms and performance evaluation. In *IEEE International Conference on Data Engineering*, pages 567–576, 2002.

[9] S. Guha, N. Koudas, and K. Shim. Data-streams and histograms. In *ACM Symposium on Theory of Computing*, pages 471–475, 2001.

[10] S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams. In *IEEE Symposium on Foundations of Computer Science*, pages 359–366, 2000.

[11] Y. Huang, R. H. Sloan, and O. Wolfson. Divergence caching in client server architectures. In *PDIS*, pages 131–139, 1994.

[12] H. V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. C. Sevcik, and T. Suel. Optimal histograms with quality guarantees. In *VLDB*, pages 275–286, 1998.

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

[14] L. O'Callaghan, N. Mishra, A. Meyerson, S. Guha, and R. Motwani. High-performance clustering of streams and large data sets. In *IEEE International Conference on Data Engineering*, pages 685–694, 2002.

[15] C. Olston, J. Widom, and B. T. Loo. Adaptive precision setting for cached approximate values. In *ACM SIGMOD Conference*, pages 355–366, 2001.

[16] O. Wolfson, S. Jajodia, and Y. Huang. An adaptive data replication algorithm. *ACM Transactions on Database Systems*, 22(2):255–314, 1997.

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