Maintaining Wavelet Synopses for Sliding-Window Aggregates

Ioannis Mytilinis
Computing Systems Laboratory, National Technical University of Athens
gmytil@cslab.ece.ntua.gr

Dimitrios Tsoumakos
Department of Informatics, Ionian University
dtsouma@ionio.gr

Nectarios Koziris
Computing Systems Laboratory, National Technical University of Athens
nkoziris@cslab.ece.ntua.gr

ABSTRACT

The IoT era has brought forth a computing paradigm shift from traditional high-end servers to "edge" devices of limited processing and memory capabilities. These devices, together with sensors, regularly produce very high data volumes nowadays. For many real-time applications, storing and indexing an unbounded stream may not be an option. Thus, it is important that we design algorithms and systems that can both work at the edge of the network and be able to answer queries on distributed, streaming data. Moreover, in many streaming scenarios, fresh data tend to be prioritized. A sliding-window model is an important case of stream processing, where only the most recent elements remain active and the rest are discarded. In this work, we study the problem of maintaining basic aggregate statistics over a sliding-window data stream under the constraint of limited memory. As in IoT scenarios the available memory is typically much less than the window size, queries are answered from compact synopses that are maintained in an on-line fashion. For the efficient construction of such synopses, in this work, we propose wavelet-based algorithms that provide deterministic guarantees and produce almost exact results. Our algorithms can work on any kind of numerical data and do not have the positive-numbers constraint of techniques such as the exponential histograms. Our experimental evaluation indicates that, in terms of accuracy and space-efficiency, our solution outperforms the exponential histograms and deterministic waves techniques.

1 INTRODUCTION

A significant part of the digital information currently produced comes in the form of data streams, i.e., continuous sequences of items of unbounded size. Since unbounded streams cannot be wholly stored in bounded memory, streaming applications usually work in an on-line fashion. The requirement of real-time processing of continuous data in high-volumes has triggered a flurry of research activity in the area. Some typical applications include sensor networks [5, 27, 38], datacenter monitoring [12], financial data trackers [39], and real-time analysis of various transaction logs [10].

Unlike conventional database query processing that allows several passes over static data, streaming algorithms are generally restricted to allow only a single pass. In order to achieve this, they often rely on building, in real-time, concise synopses of the underlying streams. These synopses typically need small space, update and query time (sub-linear to the input size), and can be used to provide approximate, yet accurate answers. Due to the exploratory nature of many data-analytics tasks, there exist a number of scenarios in which we are interested in discovering statistical patterns rather than obtain answers precise to the last decimal.

Furthermore, as for most applications there is more value in real-time information, recent data tend to be prioritized; statistics in fresh data items should be represented with higher precision than in older ones. For this purpose, various time-decay models have been proposed in the literature [8]. The *sliding-window* model [11] is one of the most intuitive ones as it only considers the most recent data items seen so far. Several algorithms have been proposed for maintaining different types of statistics over sliding-windows while requiring time and space poly-logarithmic to the window size [11, 15, 32, 37].

While a lot of work has been done for estimating basic aggregates in the sliding-window setting, the problem has not attracted much attention when using wavelets. Wavelet decomposition [35] provides a very effective data reduction tool, with applications in data mining [25], selectivity estimation [29], approximate and aggregate query processing of massive relational tables [6, 36] and data streams [9, 17]. In simple terms, a wavelet synopsis is extracted by applying the wavelet decomposition on an input collection (considered as a sequence of values) and then summarizing it by retaining only a selected subset of the produced wavelet coefficients. The original data can be approximately reconstructed based on this compact synopsis. Previous research has established that reliable and efficient approximate query processing can then be performed solely over such concise synopses [6].

In this work, we investigate the capacity of wavelets to efficiently approximate basic aggregates over a data stream under the sliding-window model. We focus on queries like COUNT, SUM and AVG, since more complex queries in sliding-windows [31] usually need to compute such basic aggregates under the hood. In order to provide theoretical guarantees, traditional techniques for the problem, like exponential histograms [11] and deterministic waves [15] are restricted to work only on streams of positive numbers. Moreover, they can only support a very specific type of queries: range queries where the end of the range is always the end of the active window. In this work, we opt for a more generic and practical solution that is able to handle streams of arbitrary numerical values and supports more generic query types. To the best of our knowledge, we are the first to investigate the use of wavelets for range queries over sliding-window streams. We present efficient algorithms for answering point and range queries in a single stream and experimentally

SSDBM '19, July 23–25, 2019, Santa Cruz, CA, USA

ACM ISBN 978-1-4503-6216-0/19/07…$15.00
https://doi.org/10.1145/3335783.3335793

evaluate them against state-of-the-art techniques. In summary, we make the following contributions:

- We investigate the efficiency of wavelets for summarizing a sliding window stream. While we consider workloads of both point and range queries, we put particular emphasis on basic aggregates like COUNT, SUM and AVG. This is the most common query type in the sliding window context and the performance of wavelets in such queries has not been studied before.
- We propose a new wavelet-based algorithm for answering range queries over a single stream in the sliding-window model and provide deterministic guarantees. The complexity of our algorithm is theoretically analyzed.
- We apply and validate our approach in a distributed setting, where multiple streams compute individual synopses and a single coordinator merges them in real-time to produce global answers.
- We experimentally evaluate our approach in both synthetic and real data and show that it outperforms, in terms of accuracy, state-of-the-art techniques such as exponential histograms and deterministic waves.

The remainder of this paper is organized as follows: Section 2 makes a literature review for the applications of wavelets in approximate query processing and for sliding-window techniques. Section 3 provides formal definitions and theoretical background on the problem. In Section 4, we present our algorithms for maintaining the synopsis in real-time and in Section 5 we show how basic statistics are computed. In Section 6 we present an extension for distributed environments. Section 7 demonstrates the experimental evaluation of our work and Section 8 concludes the paper.

2 RELATED WORK

Wavelets. In the seminal work of [6], the authors show how relational operators can be computed directly on wavelet synopses. For constructing a synopsis that optimizes the L_2-error, the authors retain a set of B wavelet coefficients, where B is a user-defined space budget.

While computationally efficient, a L_2-optimal synopsis cannot provide strong guarantees for individual queries. For this reason, a lot of prior work has focused on designing algorithms which target maximum error metrics. The construction of an optimal synopsis with respect to a non-Euclidean error is a cumbersome and computationally intensive process. Many dynamic programming algorithms [13, 14, 19, 23, 24, 30] have been proposed for this task. In order to alleviate the complexity burden, greedy algorithms [22, 28] have also been proposed.

All approaches discussed thus far refer to batch jobs, where algorithms are applied to static data. In [17, 18], the authors compute L_2-optimal wavelets on streams. As they find it more challenging, they put more emphasis on handling the unordered cash register stream model. In [9], a similar sketching technique, that allows more efficient updates, is presented for the same problem. Streaming techniques have also been proposed for the optimization of the L_∞ norm. In [20, 21], the authors present optimal algorithms for computing the optimal error in a streaming way for a broad category of non-Euclidean errors. Nevertheless, as dynamic programming needs a recursive top-down procedure in order to construct the final synopsis, these algorithms are not suitable for the scenario of an unbounded stream where inactive elements are permanently discarded. For L_∞-minimization, a greedy streaming algorithm has appeared in [22]. However, it does not support sliding-window queries.

The only wavelet-based algorithm that exists in the literature and considers the sliding-window model is the work presented in [26]. This work mainly covers point queries and it does not take into account range queries like COUNT and SUM, which are the most basic and common queries in sliding-window streams.

Sliding-Window Stream Queries. The bulk of existing work on the sliding-window model has focused on algorithms for efficiently maintaining simple statistics, such as COUNT and SUM. By *efficiently*, we mean sub-linear space and time (typically, poly-logarithmic) in the window size W. Exponential histograms [11] are a state-of-the-art deterministic technique for maintaining ϵ-approximate counts and sums over sliding windows, using $O\left(\frac{1}{\epsilon} log^2 W\right)$ space. Deterministic waves [15] solve the same basic aggregates problem with the same space complexity as exponential histograms, but improve the worst-case update time complexity to $O(1)$. In the same work [15], Gibbons also presents randomized waves to tackle COUNT-DISTINCT queries. Randomized waves, as most randomized sketching techniques, are easily parallelizable and composable (in distributed settings), but come with increased space requirements. In [37], the authors describe a randomized, sampling-based synopsis, very similar to randomized waves, for tracking sliding-window COUNT and SUM queries with out-of-order arrivals. As in randomized waves, the space requirements are also quadratic in the inverse approximation error. To address the high cost associated with randomized data structures, Busch and Tirthapura propose a deterministic structure for handling out-of-order arrivals in sliding windows [4]. Similar to other deterministic structures, this structure does not allow composition and focuses only on basic counts and sums. Finally, Chan et al. [7] investigate continuous monitoring of exponential-histogram aggregates over distributed sliding windows. The main contribution of their work lies in the efficient scheduling of the propagation of the local exponential-histogram summaries to a coordinator, without violating prescribed accuracy guarantees.

Work has also been done on sketching techniques that are suitable to answer more complex queries like k-medians [3], heavy hitters, inner products and self-joins [31, 33, 34]. However, as the majority of these techniques employ under the hood algorithms for computing basic aggregates, in this work we focus only on point queries and basic aggregates like COUNT, SUM and AVG. We develop wavelet-based algorithms that support these query types and evaluate them against other state-of-the-art techniques.

3 PRELIMINARIES

In this Section, we formally define the problem we solve, the streaming model we work on, and provide the theoretical background needed for understanding the proposed ideas.

3.1 Definitions and Problem Statement

Our goal is to evaluate the ability of wavelets to accurately compute point queries and basic range statistics (SUM, COUNT, AVG) in a data stream that follows the time-based sliding-window model and where data elements are expected to arrive in the stream-order.

Such a stream is formally defined in Definition 3.1. Henceforth, we are going to simply use the term *stream* in order to describe such a data sequence.

Definition 3.1 (Ordered Time-based Stream). An ordered, time-based data stream is an infinite sequence of tuples in the form: $S = \{(t_1, v_1), (t_2, v_2), \ldots\}, t_1 \leq t_2 \leq \ldots$, where t_i denotes the arrival time of tuple i and v_i its value.

Both the sliding-window point and range queries we tackle are defined in Definition 3.2. A point query can ask for the stream value at any time moment lying within the active window. Similarly, a range query has always the current time as the end of its interval, while the start of it can be any time moment within the window.

Definition 3.2. Let S be a stream, t the current time and W the window size.

- A **sliding-window point query** $P\left(t_q\right)$ on S returns an estimation for the value v_q that arrived at time t_q, $t_q \in [t - W, t]$.
- A **sliding-window range query** $AGG\left(t_q\right)$ on S returns an estimation for an aggregate $AGG \in \{SUM, COUNT, AVG\}$ computed over the time range: $\left[t_q, t\right]$, where $t_q \in [t - W, t]$.

While we mainly consider range queries of the described form, in order to demonstrate the general applicability of our approach, in Section 7, we also investigate queries of the form $[s, e]$, where $t - W \leq s \leq e \leq t$.

3.2 Wavelets

Wavelet analysis is a major mathematical technique for hierarchically decomposing functions. The wavelet decomposition of a function consists of a coarse overall approximation together with detail coefficients that influence the function at various scales [35]; it is computationally efficient and has excellent energy compaction and decorrelation properties, which can be used to effectively generate compact representations that exploit the structure of data.

Haar wavelets constitute the simplest possible orthogonal wavelet system. Assume a one-dimensional data vector A containing $N = 8$ data values $A = [8, 6, 7, 7, 12, 12, -1, -3]$. The Haar wavelet transform of A can be considered as a sequence of pairwise averaging and differencing operations. We first average the values in a pairwise fashion to get a new "lower-resolution" representation of the data with the following average values: $[7, 7, 12, -2]$. The average of the first two values (i.e., 8 and 6) is 7, the average of the next two values (i.e., 7 and 7) is 7, etc. It is obvious that, during this averaging process, some information has been lost and thus the original data values cannot be restored. To be able to restore the original data array, we need to store some *detail coefficients* that capture the missing information. In Haar wavelets, the detail coefficients are the half-difference of the corresponding data values. In our example, for the first pair of values, the detail coefficient is 1 (since $(8 - 6)/2 = 1$) and for the second is 0 ($(7 - 7)/2 = 0$). After applying the same process recursively, we generate the full wavelet decomposition that comprises a single overall average followed by three hierarchical levels of 1, 2, and 4 detail coefficients respectively (see Table 1). In our example, the wavelet transform (also known as the wavelet decomposition) of A is $W_A = [6, 1, 0, 7, 1, 0, 0, 1]$. The

Table 1: Wavelet decomposition example

Resolution	Averages	Detail Coef.
3	$[8, 6, 7, 7, 12, 12, -1, -3]$	–
2	$[7, 7, 12, -2]$	$[1, 0, 0, 1]$
1	$[7, 5]$	$[0, 7]$
0	$[6]$	$[1]$

complete Haar wavelet decomposition W_A of a data array A is a representation of equal size as the original array. Each entry in W_A is called a *wavelet coefficient*. The main advantage of using W_A instead of A is that, for vectors containing similar values, most of the detail coefficients tend to have very small values. Therefore, eliminating such small coefficients from the wavelet transform (i.e., treating them as zeroes) introduces only small errors when reconstructing the original array and thus results to a very effective form of lossy data compression. Given a budget constraint $B < N$, the problem of *wavelet thresholding* is to select a subset of at most B coefficients that minimize an aggregate error measure in the reconstruction of data values.

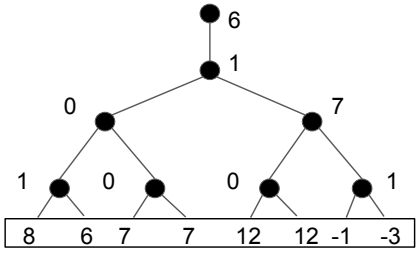


Figure 1: An error tree that illustrates the hierarchical structure of the Haar wavelet decomposition

The *error-tree*, introduced in [29], is a hierarchical structure that illustrates the key properties of the Haar wavelet decomposition. Figure 1 depicts the error-tree for our simple example data vector A. Each internal node c_i ($i = 0, \ldots, 7$) is associated with a wavelet coefficient value, and each leaf d_i ($i = 0, \ldots, 7$) is associated with a value in the original data array. Given an error-tree T and an internal node c_k of T, we let leaves_k denote the set of data nodes in the sub-tree rooted at c_k. This notation is extended to leftleaves_k (rightleaves_k) for the left (right) sub-tree of c_k. We denote $path_k$ as the set of all nodes with non-zero coefficients in T which lie on the path from a node c_k (d_k) to the root of the tree T. We also denote $path_{[l,h]} = path_l \cup path_h$.

Given the error-tree representation of a one-dimensional Haar wavelet transform, we can reconstruct any data value d_i using only the nodes that lie on $path_i$. That is

$$d_i = \sum_{c_j \in path_i} \delta_{ij} \cdot c_j, \delta_{ij} = \begin{cases} 1 & d_i \in \text{leftleaves}_j \\ -1 & otherwise \end{cases}$$

For example, in Figure 1, value $d_5 = 6 - 1 + 7 - 0 = 12$. A range sum $d(l : h)$ can be computed using only nodes $c_j \in path_{[l,h]}$, by $d(l : h) = \sum_{c_j \in path_{[l,h]}} c_j \cdot x_j$, where:

$$x_j = \begin{cases} (h - l + 1) & j = 0 \\ \left(|\text{leftleaves}_{j,l:h}| - |\text{rightleaves}_{j,l:h}|\right) & otherwise \end{cases} \quad (1)$$

Here, $\text{leftleaves}_{j,l:h} = \text{leftleaves}_j \cap \{d_l, d_{l+1}, .., d_h\}$ and $\text{rightleaves}_{j,l:h} = \text{rightleaves}_j \cap \{d_l, d_{l+1}, .., d_h\}$. That means that node c_j contributes to the range sum $d\,(l:h)$ positively as many times as there are leaf nodes of the left sub-tree of c_j in the summation range, and negatively as many times as there are leaf nodes of the right sub-tree of c_j, while the value of c_0 contributes positively for each leaf node in the summation range. In our example, $d\,(3:6) = -1 \cdot 0 + (-1) \cdot 0 + (-2) \cdot 1 + 4 \cdot 6 + 1 \cdot 7 + 1 = 30$.

Thus, reconstructing a single data value involves summing at most $logN + 1$ coefficients and reconstructing a range sum involves summing at most $2logN + 1$ coefficients, regardless of the width of the range.

4 DYNAMIC SYNOPSIS MAINTENANCE

In this Section, we present our efficient algorithms for the computation and online maintenance of a wavelet synopsis on a single stream. The construction process should be constrained to a limited memory budget, that is usually much smaller than the window size ($B << W$). This is a realistic requirement in many real-life applications. For example, embedded devices that are often met in IoT scenarios, have a memory capacity of only a few MB. Keeping track of the average value, in a stream of 8 byte long numbers and a window size of $W = 100M$ data elements, needs 800 MB which may not be available. Thus, a space budget B should be defined and cap the number of retained wavelet coefficients. In the analysis of this Section, we consider synopses of logarithmic size, i.e., $B = O\,(logW)$.

In the following, we describe our algorithm for supporting sliding-windows and theoretically analyze its complexity. The requirements we need to cover are:

- Support for sliding-windows.
- Deterministic error guarantees for both point and range queries.
- Real time[1] query and update operations.

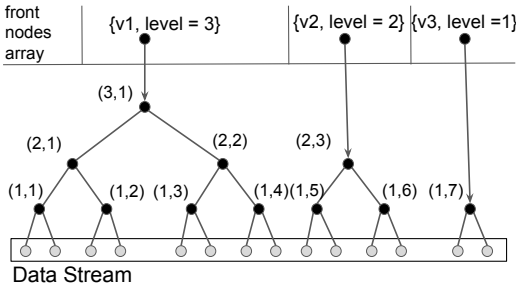


Figure 2: Error-tree for streaming data.

Streaming Error-Tree. Similarly to previous works, we operate on the streaming version of an error-tree [22, 26]. Each pair of newly arrived items is subjected to the wavelet transform and inserted into the error-tree. During this construction process, at some time t, the number of stream data that have arrived may be unequal to a power of two. That means that the error-tree has not formed a full binary tree as in the static case and unconnected sub-trees of different heights may exist. That means that there can be at most one such sub-tree rooted at each error-tree level (thus, $\lfloor logW \rfloor$ sub-trees). Figure 2 depicts an example, where there are three unconnected sub-trees of height one, two and three respectively. In order to avoid information loss and be able to continue the decomposition process, we need to keep track of all sub-trees in the active window. For this purpose, we use the *front nodes array* structure and its elements *fnodes*. For each sub-tree, that we want to track, we create a fnode annotated with: (i) the timestamp of the first supported item, (ii) the level of the sub-tree and (iii) the average value of its data. We then set the created fnode to point to the sub-tree and append it in the front nodes array, as shown in Figure 2.

Indexing Coefficients. In the streaming error-tree, a wavelet coefficient c_i is indexed by a tuple (l_i, o_i), where l_i is the level of the coefficient in the error-tree and o_i its order in the specific level. Figure 2 illustrates the indexing scheme for our example. Given two coefficients c_i, c_j, where c_i is an ancestor of c_j, c_j belongs to the left sub-tree of c_i if: $2 \cdot o_j - 1 < (2 \cdot o_i - 1) \cdot 2^{l_i - l_j}$.

In this work we exploit the sliding-window and propose an efficient representation that minimizes the space overhead for a coefficient. The key observation is that we do not have to index an infinite stream but, at any given time, the synopsis approximates a single window of size W. As the level of a coefficient can be at most $logW$, for l_i we need at most $loglogW$ bits. For reducing the size of the o_i values, which are infinite in an unbounded stream, we use a wrap around counter $o'_i = \left\lceil (o_i - 1)\ mod \frac{2W}{2^{l_i}} + 1 \right\rceil$ that uses $log \frac{2W}{2^{l_i}}$ bits for a coefficient in level l_i. With this scheme and for a window of size 1 billion, a coefficient needs at most 35 bits for storing both l_i and o_i.

Algorithm Outline. Algorithm 1 shows the outline of the streaming algorithm for the construction of a wavelet synopsis. Each pair of newly arrived data is transformed into a wavelet coefficient and inserted into the error-tree. The addition of a new coefficient may trigger the creation of more coefficients in higher levels. In Figure 2, when two more items arrive, a new wavelet coefficient will be inserted in the first level of the error-tree. As there is already one node in the first level, the two coefficients will be averaged and differenced and create a new coefficient in level two. The process will be recursively repeated and new wavelet coefficients are expected to be also added in levels three and four. In general, every new item in the stream can fire up to $\lceil logW \rceil$ insert-updates in the wavelet structure.

Algorithm 1: Streaming Algorithm for Constructing a Sliding-Window Wavelet Synopsis

```
   input: Stream S, Budget B, Window size W
1  currTime = 0; wSynopsis = new WaveletSynopsis();
2  for data items in S do
3      currTime = currTime + 2; d1, d2 = read(S);
4      wSynopsis.deleteExpired(currentTime, W);
5      wSynopsis.insert(currTime,W,d1, d2);
6      while wSynopsis.size > B do
7          wSynopsis.discardNext();
```

In line 4, we first check whether there are coefficients that lie outside the active window and thus have expired. If such coefficients exist, we can safely discard them releasing this way space without

[1] As common practice in the literature, by *real time*, we mean at most $O\,(logW)$ cost.

compromising accuracy (they support a range we are no longer interested in).

Next, we insert the new elements. Depending on the data distribution, the wavelet transform may produce some zero coefficients. These coefficients are never inserted in the structure we maintain. If after the insert-step, the size of the synopsis still exceeds B, we discard coefficients according to a greedy criterion (will be later discussed) until the size of the synopsis respects the available budget.

Algorithm 2: Insert

```
   input: Number of arrived items N, window size W, item
          d1,item d2
1  f = fnode with lowest level; tmp = null; l = 0
2  maxLevel = log(W/logW)
3  while N > 0 and N mod 2 = 0 do
4      N = N / 2; l = l + 1
5      if l > maxLevel then break
6      if tmp = null then
7          avg = (d1 + d2) / 2; v = (d1 - d2) / 2
8          minCf = maxCf = v
9      else
10         avg = (avg + tmp) / 2; v = tmp - avg
11         minCf = min (prevFnode.minCf, tmpMin, v)
12         maxCf = max (prevFnode.maxCf, tmpMax, v)
13     ci = new WaveletCoef(li = l, oi = N, value = v)
14     ci.maxCoefInSubtree = maxCf
15     ci.minCoefInSubtree = minCf
16     if ci ≠ 0 then put ci in min-heap
17     delete fnode below f
18     if no fnode in level l then
19         f = new Fnode(level = l, value = avg)
20         f.minCf = minCf; f.maxCf = maxCf
21         if l < maxLevel then frontNodesArray.add(f)
22         else topLevelFnodes.add(f)
23     else
24         tmp = f.value
25         tmpMin = f.minCf; tmpMax = f.maxCf;
26     if f.pointer = null then f.pointer = ci
27     f = fnode at next level
```

We now delve into the internals of each of the *insert*, *deleteExpired* and *discardNext* functions.

Insert. The algorithm for the insertion of new coefficients in the synopsis is presented in Algorithm 2. For each pair of arrived items d_1, d_2, we perform averaging and differencing (line 7) and create a new wavelet coefficient c_i. If c_i is non-zero, we add it to a min-heap (line 16) in order to specify its order of deletion. In line 18, we check if c_i is the only node at level l. If this is the case, we create a new fnode (line 19) that points to c_i, else we continue the process at the next level of the error-tree, as explained in the example of Figure 2.

According to our algorithm, all fnodes that support a part of the active window are retained in the synopsis. This is the reason why fnodes are not inserted into the min-heap. As we will explain in Section 5, this design choice improves the approximation quality of range queries.

Moreover, in line 5 of the algorithm, we notice that a cap is enforced on the maximum level of a sub-tree; the wavelet decomposition is not allowed to continue further than *maxLevel* levels. This decision permits the existence of more than one fnodes with *maxLevel* levels. We store these fnodes in a separate structure called *topLevelFnodes* (line 22). We claim that a limit on the maximum level of the error-tree offers two advantages: i) lower bounded update times, and ii) allows for the accurate computation of range queries.

Our first claim can be trivially verified. From the *while* condition of Algorithm 2, we can see that an insert operation can trigger up to $logW$ updates. For a $maxLevel < logW$, we directly restrict the number of updates at every time unit. The impact of $maxLevel$ in the accuracy of range queries will be discussed in Section 5, where we describe the query answering mechanism.

Now, we are going to investigate what is an appropriate value for *maxLevel*. A small value offers the advantages we just mentioned. Nevertheless, as all fnodes are retained in the synopsis, a cap on the maximum level increases the space we need to dedicate to the front nodes array. Thus, we need to set a value such that we enjoy the benefits of a short tree without significantly increasing space complexity. The value we select is $log\left(\frac{W}{logW}\right)$. The following Lemma shows that with this choice we only require poly-logarithmic space in the window size for storing the front nodes array.

Lemma 4.1. *Consider a wavelet error-tree T built over W data points. Setting the constraint that each sub-tree of T cannot have more than $log\left(\frac{W}{logW}\right)$ levels, results in storing at most $log\left(\frac{W^2}{logW}\right) - 1 = O(logW)$ fnodes.*

Proof. Let k denote the maximum permitted size for a sub-tree. Thus, within a window of size W there can be up to $\lceil\frac{W}{k}\rceil$ such sub-trees, and thus $\lceil\frac{W}{k}\rceil$ fnodes. As the given budget B is usually poly-logarithmic in W, we want to store at most $O(logW)$ fnodes. So, it should hold: $\frac{W}{k} \leq c \cdot logW, c \geq 1 \Rightarrow k \geq \frac{W}{c \cdot logW}$. Thus, the minimum sub-tree size we can tolerate without violating the constraint of $O(logW)$ fnodes is $\frac{W}{c \cdot logW}$ and has $M = log\left(\frac{W}{c \cdot logW}\right)$ levels. However, the construction process of a wavelet tree is such that is impossible to cover the whole window only with sub-trees of level M. As it is known that $\Sigma_{i=0}^{n-1} 2^i = 2^n - 1$, we can substitute a sub-tree of size k with up to M sub-trees of levels $l = 1, .., M-1$. This way, there are at most $\frac{W}{k} - 1 + M = c \cdot logW - 1 + logW - c \cdot loglogW = (c+1)\, logW - c \cdot loglogW - 1$ sub-trees and thus fnodes in the window. As we want to save space, we set $c = 1$ which is the minimum possible value, and get: $2logW - loglogW - 1 = log\left(\frac{W^2}{logW}\right) - 1 = O(logW)$. □

The cost for inserting new elements in the wavelet synopsis is given by Lemma 4.2.

Lemma 4.2 (Insertion Time). *Considering a synopsis size of $B = O(logW)$, an arriving pair of data items leads to a worst case insertion time of $O\left(log\frac{W}{logW} \cdot loglogW\right)$ and $\Theta\left(\left(1 - \frac{1}{W}\right) logW - loglogW\right)$ in the average case.*

Proof. The cost of an insert-update consists of the cost of creating new coefficients and the cost of re-configuring the binary heap. The proof for the worst-time case is straightforward: As we discussed, an insert-update can lead to the creation of L new wavelet coefficients, where L is the size of the tree. Since our algorithm permits only sub-trees of height up to $log\left(\frac{W}{logW}\right)$, it follows that this is also the maximum number of operations that an insert-update can cause. Moreover, since the synopsis should occupy only polylogarithmic space, we assume a min-heap of size $B = O(logW)$. Thus, the worst-case insertion in the heap is $O(loglogW)$. It follows that the total needed worst-case time for updating the synopsis when two new data items arrive is $O\left(log\frac{W}{logW} \cdot loglogW\right)$.

We now compute Θ complexity. The insertion in a binary heap needs $\Theta(1)$ time on average. The question is how many wavelet coefficients are created with every new arrival in the average case. Without loss of generality, we assume a tree of size N, where N is a power of two. Each arriving item can trigger the creation of $1 \leq i \leq logN$ coefficients. Since there are N items within the window, we first compute how many of them create 1 coefficient, how many 2, etc. Let $a(j)$ denote the number of coefficients within a window that lead to the creation of paths of length $logN - j$. We observe that only the last element can create a path of length $logN$, i.e., $a(0) = 1$. The same holds for a path of length $logN - 1$. There are two paths in the window that have length at least $logN - 1$. However, the one of them has length $logN$ and thus, $a(1) = 1$. With similar reasoning, we observe that the following recursion holds: $a(0) = 1$ and $a(j) = \sum_{i=0}^{j-1} a(i)$. As the first two elements of the $a(j)$ sequence add up to 2, it is easy to derive that:

$$a(j) = \begin{cases} 1 & j = 0 \\ 2^{j-1} & j \neq 0 \end{cases}$$

Since it is known that $\sum_{i=0}^{n-1} 2^i = 2^n - 1$, we observe that:

$$\sum_{j=0}^{logN-1} \frac{a(j)}{N} = \frac{1 + \sum_{j=0}^{logN-1} 2^j}{N} = \frac{1 + 2^{logN} - 1}{N} = 1$$

and thus the term $\frac{a(j)}{N}$ can represent the probability of creating a path of length $logN - j$. Let the random variable X express the number of updates a newly arriving data pair yields. The expected value of X can be expressed as:

$$\mathbb{E}(X) = \sum_{j=0}^{logN-1} \frac{a(j)}{N} \cdot (logN - j) =$$

$$\frac{logN}{N} + \frac{logN}{N} \sum_{j=1}^{logN-1} 2^{j-1} - \frac{1}{N} \sum_{j=1}^{logN-1} j \cdot 2^{j-1} =$$

$$\frac{logN}{N}\left(1 + \sum_{j=1}^{logN-1} 2^{j-1}\right) - \frac{1}{N} \sum_{j=1}^{logN-1} j \cdot 2^{j-1} \quad (2)$$

We use again the fact that $\sum_{i=0}^{n-1} 2^i = 2^n - 1$ in order to compute the first term. For $k = j - 1$, we have: $\sum_{j=1}^{logN-1} 2^{j-1} = \sum_{k=0}^{logN-1-1} 2^k = 2^{logN-1} - 1 = \frac{N}{2} - 1$ and the first term of Equation 2 is equal to $\frac{logN}{2}$. For the second term, it is easily proven that when n is a finite number, it holds: $\sum_{j=1}^{n} j \cdot x^{j-1} = 1 - \frac{x^n}{(1-x)^2} + \frac{nx^n}{1-x}$. For $x = 2$ and $n = logN - 1$, we get that:

$$\sum_{j=1}^{logN-1} j \cdot 2^{j-1} = 1 - 2^{logN-1} - (logN - 1) \cdot 2^{logN-1} = 1 - \frac{NlogN}{2}$$

Thus, Equation 2 becomes:

$$\mathbb{E}(X) = \frac{logN}{2} - \frac{1}{N}\left(1 - \frac{NlogN}{2}\right) = logN - \frac{1}{N}$$

Thus, the total update time for every arrived pair in the stream is $\Theta(1) \cdot \Theta\left(logN - \frac{1}{N}\right)$. As our sub-trees have a size of $N = \frac{W}{logW}$, Θ complexity becomes: $log\left(\frac{W}{logW}\right) - \frac{logW}{W} = \left(1 - \frac{1}{W}\right) logW - loglogW$. □

Delete Expired. We first check if all fnodes still support the active window. As an fnode f supports $2^{f.level}$ data points beginning from f.start, we have to discard all fnodes with: f.start $+ 2^{f.level} < currTime - W$. If a fnode is deleted, so is the whole sub-tree underneath it.

We then scan all the remaining elements to check if there are coefficients that also need to be removed. The criterion for removing a coefficient c_i is: $o_i \cdot 2^{l_i} - 1 < currTime - W$. As we require $B = O(logW)$, the cost of this scan operation is also $O(logW)$.

Discard Next. When budget is exceeded, we need to discard some coefficients. The heuristic for selecting coefficients to discard depends on the error metric we need to optimize. If L_2-norm is the targeted metric, we should always keep the B largest coefficients in normalized value. If the minimization of L_∞ is required, we select each time the coefficient c_k with the minimum *maximum potential absolute error* MA_k [22]. The MA_k value is defined as: $max_{d_j \in leaves_k}\{|err_j - \delta_{jk} \cdot c_k|\}$, where err_j is the signed error for item j, and shows the maximum error that the removal of c_k would produce. In either case, for efficiently identifying the node that should be discarded and assist the greedy selection, the synopsis is organized as a min-heap structure. In our work, we use the L_∞ norm and implement min-heap as a binary heap.

Lemma 4.3 gives the cost of deletions either due to expiration or budget excess.

Lemma 4.3 (Deletion Time). *The time spent in delete operations every time the synopsis is updated is $O(logW)$ in both worst and average case.*

Proof. Delete operations occur due to either window sliding or a manual coefficient removal in order to respect the budget constraint. We observe that in the permanent state of the algorithm (more than B data items have already arrived) the synopsis size increases by at most two elements with every new arrival. Thus, there are at most two deletions that we need to make. As the *deleteExpired* function can delete at most one coefficient, the *discardNext* function is called at most twice. The manual removal of a coefficient results in the extraction of the minimum element of a binary heap. Considering $B = O(logW)$, this operation has a worst-case complexity $O(loglogW)$ and average time $\Theta(1)$. As for identifying an

expired coefficient we need to scan the whole synopsis, a $O(logW)$ operation is needed for both the worst and average case. □

Error Guarantees. Regardless of which error-metric is optimized, the constructed synopsis should be able to provide queries with deterministic guarantees. As shown in [22], providing guarantees for point queries demands each node to maintain the maximum and minimum signed errors of its left and right sub-trees.

In this work, we also provide deterministic guarantees for range queries. As mentioned in Section 3, the value of a SUM query over a range $[t_1, t_2]$ can be exactly reconstructed, by only using the coefficients $c_j \in path_{[t_1,t_2]}$, according to Equation 1. Here, we observe that under the sliding-window model, the sum can be computed solely based on the coefficients $c_j \in path_{t_1}$, i.e., the ones that belong to the left path of the queried interval. As we explain in detail in Section 5, in the sliding-window model, we expect some sub-trees to be fully-contained in the query-range and one last sub-tree to partially overlap with it. Let us consider that $[t_1, t_2]$ is the range of overlap with the last sub-tree. Thus, by definition, $path_{t_2}$ is the rightmost path of a full binary tree. As such, every coefficient c_j in $path_{t_2} \backslash path_{t_1}$ is expected to have $x_j = 0$ and does not contribute to the sum, either it is contained in the synopsis or not. Thus, $SUM_{[t_1,t_2]} = \sum_{c_j \in path_{t_1}} c_j x_j$.

For providing error guarantees, we need to bound this sum. No matter if we have deleted a coefficient c_j or not, the x_j value is always known since it only depends on the coefficient's position in the error-tree and the query range. So, if we had some bounds for the deleted (and thus, unknown) coefficients c_j, such that $l_j \leq c_j \leq h_j$, it would hold:

- $x_j \geq 0 \Rightarrow l_j x_j \leq c_j x_j \leq h_j x_j$
- $x_j < 0 \Rightarrow h_j x_j \leq c_j x_j \leq l_j x_j$

By summing up these inequalities for all deleted coefficients c_j, we obtain deterministic guarantees for the $SUM_{[t_1,t_2]}$. The idea for bounding c_j values is to keep track of the minimum and maximum coefficients in each sub-tree. In Algorithm 2, we annotate with blue color all required modifications for tracking minimum/maximum coefficients in each sub-tree.

5 QUERY ANSWERING

Point queries $P\left(t_q\right)$ are answered as explained in Section 3, i.e., $P\left(t_q\right) = \Sigma_{c_j \in path_q} \delta_{qj} \cdot c_j + f.value$, where f is the corresponding fnode of the sub-tree where t_q belongs. We are now going to focus on the query answering mechanism for range queries.

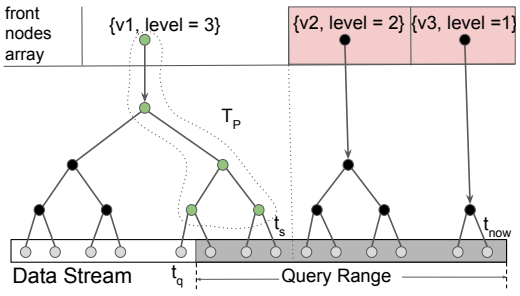


Figure 3: Range query answering

Figure 3 depicts a range query $AGG\left(t_q\right)$. The range of interest $\left[t_q, t_{now}\right]$ is highlighted with grey color. We observe that there are sub-trees which are fully-contained in the range and a last sub-tree T_p that partially overlaps with it. Let us denote t_s the moment in time that separates T_p with the leftmost fully-contained sub-tree.

For the part of the query that corresponds to fully-contained sub-trees we can provide an exact answer. Thus, $AGG\left(t_q\right) = AGG_{approx} \oplus AGG_{exact} = AGG_{[t_q,t_s]} \oplus AGG_{t>t_s}$, where $\oplus$ is a function that combines partial aggregates. This function is a simple addition for the case of COUNT and SUM queries, while for AVG Lemma 5.1 holds.

Lemma 5.1. *Let $avg(\cdot)$ and $n(\cdot)$ denote the averaging and counting functions respectively. The average value of region $X = \bigcup x_i$, $i = 1, 2, .., k$ with $x_i \cap x_j = \emptyset$ can be computed as:*

$$AVG(X) = \oplus\left(avg(x_1), ..., avg(x_k)\right) = \sum \frac{n(x_i) \cdot avg(x_i)}{n(X)}$$

We first show how to compute the exact part of the aggregate and then discuss how to approximate the range that intersects with the last sub-tree T_p. Recall that each fnode f_i keeps information about the level of its sub-tree T_i and the average value of the corresponding data elements. Thus, an aggregate of T_i can be computed solely based on f_i. Considering that a data item arrives at each time unit, a COUNT query can be computed as $2^{f_i.level}$, the answer to an AVG query is $f_i.value$ and the SUM can be derived by $f_i.value \cdot 2^{f_i.level}$. So, $AGG_{t>t_s} = \oplus\left(AGG_{T_i}, ..., AGG_{T_j}\right)$, where $\left\{T_i, ..., T_j\right\}$ are all the sub-trees that are fully-contained in the range query $AGG\left(t_q\right)$.

For approximating $AGG_{[t_q,t_s]}$ we use the wavelet coefficients that lie in $path_{t_q}$. We remind that for coefficients c_j in $path_{t_s} \backslash path_{t_q}$ we expect $x_j = 0$. As there is exactly one item that arrives at each time unit, we know that there are $t_s - t_q + 1$ items in the range. A SUM query can be approximated as: $SUM_{[t_q,t_s]} = \sum_{c_j \in path_{t_q}} c_j x_j + f_p.value \cdot \left(t_s - t_q + 1\right)$ and an AVG query can then be easily answered as: $\frac{SUM_{[t_q,t_s]}}{(t_s - t_q + 1)}$. Guarantees for the approximate $AGG_{[t_q,t_s]}$ are provided as follows: we traverse $path_{t_q}$ in a bottom-up fashion. For each position j of the error-tree, we check if coefficient c_j exists in the synopsis. If it does, we compute its contribution $c_j x_j$. If it does not, we buffer the x_j value that corresponds to the missing coefficient until we find the next coefficient that exists in the synopsis. Then, we use the minimum and maximum coefficients stored in this node, in order to bound the contribution of the missing coefficients.

Thus far, we have assumed that an item arrives at each time unit. However, in reality, streams may be bursty and arrival rates do not follow a regular pattern. In order to handle the general case and be able to answer all COUNT, SUM and AVG queries, we maintain two distinct wavelet structures. The first one keeps track of a bit-stream $\{(t, b), b \in \{0, 1\}\}$ that indicates whether a tuple has appeared at time t. The second one approximates the value distribution of the actual input stream. Let BW denote the wavelet synopsis of the bit-stream and VW the synopsis of the value-stream. The procedure for updating BW, VW is presented in Algorithm 3. Every time t a data item (t, v) appears, we insert it in VW exactly as explained in Section 4. Moreover, we insert the tuple $(t, 1)$ in BW. On the other

hand, if the stream is idle at t and no data arrives, we insert the tuple: $(t,0)$ to both BW and VW. This mechanism ensures that a direct mapping between the time and wavelet domains always exists. Let us also note that keeping two structures does not constitute a deficiency of our approach. Exponential histograms and waves do the same in order to support both COUNT and SUM queries.

Algorithm 3: BW-VW updates

```
Initialize BW, VW;
for every time unit t do
    (t, v) = listenToStream();
    if (t, v) ≠ null then
        BW.insert((t, 1)); VW.insert((t, v));
    else BW.insert((t, 0)); VW.insert((t, 0)) ;
```

Answering COUNT queries on the stream is translated into SUM queries on the BW structure. For instance, if we need to know the number of measurements that a sensor produced between times t_1 and t_2, we have to add the 1-bits that exist in the corresponding time range. SUM queries on the input stream are answered by the VW structure. Since in the absence of arrived data we insert zero-values to VW, we do not affect the result of additive operations. For AVG and point queries, we have to "touch" both structures. For an AVG query, we compute the sum from VW, the count from BW and divide the results. For point queries, we issue the same query to both structures; BW checks the existence of an item at time t and VW approximates its value.

At this point, we discuss the choice of limiting the maximum level of a sub-tree. From the described query answering mechanism, we see that an error is introduced only due to the range $[t_q, t_s]$. Intuitively, the higher is the T_P wavelet sub-tree, the larger this range can be. By keeping sub-trees short, we increase the possibility to have more sub-trees fully-contained in the query-range and thus, increase the exact part of the answer $AGG_{t>t_s}$. The following Lemma shows how the maximum level we allow for sub-trees affects the relation between the $[t_q, t_s]$ and $[t_{s+1}, t_{now}]$ ranges.

Lemma 5.2. *Let Q a range query, $E = [t_{s+1}, t_{now}] \subseteq Q$ the sub-range of Q for which our structure provides an exact result and A the sub-range of Q that we need to approximate. It holds that:* $\frac{1}{2(logW-1)} \leq \frac{|A|}{|E|} \leq \frac{1}{2}$

Proof. We distinguish two cases depending on whether A overlaps with a sub-tree of size $\frac{W}{logW}$ or not. Let us initially assume that A overlaps with a sub-tree of size 2^k smaller than $\frac{W}{logW}$ nodes. The maximum length of the range we need to approximate is 2^{k-1}. By the wavelet construction, it is guaranteed that there are $k-1$ trees in E of sizes $2, 4, .., 2^{k-1}$ and thus $|E| = \sum_{i=2}^{k-1} 2^i = 2^k$. It follows that $\frac{|A|}{|E|} = \frac{2^{k-1}}{2^k} = \frac{1}{2}$. We now consider the case where A overlaps with a sub-tree of size $\frac{W}{logW}$. The maximum length of the range we have to approximate in this case is $\frac{W}{2logW}$. Since there are $1 \leq s \leq logW - 1$ such sub-trees in the active window, the size of E will be: $\frac{W}{logW} \leq |E| \leq \frac{W}{logW}(logW - 1)$ and $\frac{1}{2(logW-1)} \leq \frac{|A|}{|E|} \leq \frac{1}{2}$. □

5.1 Discussion

Lemma 5.2 implies that for range queries of length W, our method has to approximate only a small part of $\frac{1}{2(logW-1)}$ of the query. The larger the window size, the larger the portion of the query we can exactly compute. For windows larger than 1 million items, we have to approximate less than 3% of the submitted query. This is a direct consequence of limiting the maximum level a sub-tree can have. The corresponding ratio in classic wavelets is always $\frac{1}{2}$.

As factor $\frac{1}{2(logW-1)}$ restricts a query range but does not contain information on data values distribution, it favors mostly COUNT queries but no theoretical guarantees can be given for SUM and AVG. However, our experiments in Section 7 show that our approach is very robust and that for queries of length W high quality results are achieved for all examined datasets, both real and synthetic.

According to Lemma 5.2, for aggregate queries of length less than W, the range our scheme needs to approximate is half the query range, as in the classic wavelet structure. In that case, result quality becomes highly dependent on the available budget B and data distribution. Nevertheless, as our method dedicates a larger portion of its budget for keeping fnodes than simple wavelets, we guarantee that at every time moment, we maintain the average value of all sub-trees in the active window. Our experiments show that the proposed algorithm outperforms classic wavelets in range queries in all examined cases.

Other methods, such as exponential histograms (EH), provide theoretical guarantees by tracking query results over time. Instead of approximating the data distribution of the stream, as we do in this work, they approximate the distribution of a query over time. For example, in the case of a SUM query, they maintain a structure that tracks the SUM at different time intervals. The benefit of wavelet-based techniques compared to such approaches is flexibility to handle more generic query types and underlying data distributions. EH-like techniques are restricted to only handle streams of positive integers and answer a single query. While due to Lemma 5.2, our method performs better when applied to positive numbers, in Section 7, we show that it can also be efficiently applied to streams of arbitrary numerical data. Moreover, we show that the same structure can be used to also answer point queries and more general range queries, where the end of the query range is not equal to the current time.

6 DISTRIBUTED WAVELETS FOR STREAMS

We also address the problem of tracking basic sliding-window aggregates over the union of local streams in a large-scale distributed system. By union, we mean a linear combination (e.g., average) of the remote streams. In our setting, the remote sites are not allowed to exchange information with each other but communicate through the network with a centralized coordinator node. Let us consider a linear function F applied on a set of N distributed streams $S_i, i = 1, .., N$. Our goal is to answer COUNT, SUM and AVG queries on F, i.e., $AGG(F(S_1, .., S_N))$, while minimizing communication; collecting all streaming data is too costly to afford in many real use-cases. Therefore, similarly to [16], each remote site computes a wavelet synopsis (WS) on its local stream (S) and it is only the synopses that are sent to the coordinator. This way, the communication cost is reduced.

The coordinator computes the requested aggregate directly in the wavelet domain. As Haar wavelets are linear functions of the original streams and F is also a linear function, if we apply F on the individual synopses WS_i, we are going to get a wavelet synopsis of $F(S_1,..,S_N)$. Thus, $WS(F(S_1,..,S_N)) = F(WS_1,..,WS_N)$ and we can approximate the query $AGG(F(S_1,..,S_N))$ as $AGG(F(WS_1,..,WS_N))$.

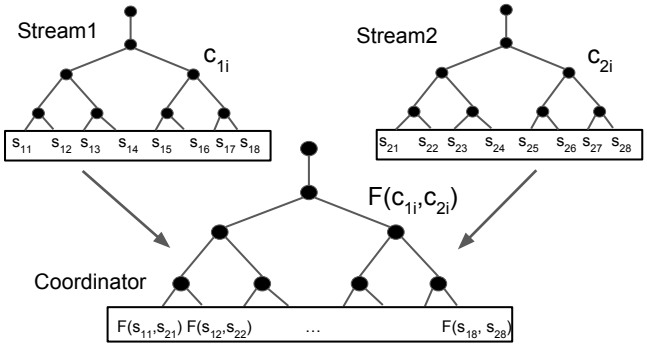


Figure 4: Composition of individual wavelet synopses.

Figure 4 illustrates an example. Sites 1,2 monitor their local streams s_{1i}, s_{2i} and construct the corresponding wavelet synopses. At the coordinator node, we want to track the stream $F(s_{1i}, s_{2i})$. Instead of collecting the s_{1i}, s_{2i} values, applying F on them, computing the wavelet transform and constructing the synopsis, we observe that for each coefficient with index i, it holds that $cm_i = F(c_{1i}, c_{2i})$, where cm_i is the corresponding coefficient in the error-tree of the coordinator. Thus, it suffices to aggregate by index the coefficients and compute F function. The following Lemma shows that the maximum error guarantees in the wavelet synopsis of the coordinator also follow the F function. Therefore, we are able to provide deterministic guarantees to queries on the union of the streams.

Lemma 6.1. *Let $S_1, S_2, .., S_N$ be N streams and $\epsilon_{1k}, \epsilon_{2k}, .., \epsilon_{Nk}$ the corresponding maximum absolute errors for the reconstruction of the data value at $t = k$. The corresponding error in the stream $F(S_1,..,S_N)$, where F is a linear function, is $F(\epsilon_{1k}, \epsilon_{2k}, .., \epsilon_{Nk})$.*

Proof. Since the reconstruction error of stream S_i for $t = k$ is ϵ_{ik}, it holds: $|\sum \delta_{kj} c_{ij} - d_{ik}| \leq \epsilon_{ik}$, where c_{ij} are the wavelet coefficients of S_i that have been retained in the synopsis. Let $F = a_1 x_1 + .. + a_N x_N$. By applying F on the above inequalities we get: $-a_i \epsilon_{ik} \leq \sum \delta_{kj} c_{ij} a_i - a_i d_{ik} \leq a_i \epsilon_{ik}$. Summing up for all streams yields:

$$|\sum \delta_{kj} F\left(c_{1j},..,c_{Nj}\right) - F(d_{1k},..,d_{2k})| \leq F(\epsilon_{1k},..,\epsilon_{Nk})$$

□

7 EXPERIMENTAL EVALUATION

In this Section, we present the experimental evaluation of our work. We compare our wavelet-based approach against other state-of-the-art techniques for range queries in sliding windows. Algorithms are compared in terms of accuracy and memory consumption. As accuracy we measure the real observed relative error, i.e.,

$$\text{Real Error} = \frac{|\text{precise answer} - \text{approximate answer}|}{\text{precise answer}} \cdot 100\%$$

Algorithms. The techniques we compare to our work are: (i) Exponential Histograms (*EH*) [11], (ii) Deterministic Waves (*DW*) [15] and (iii) the classic wavelet structure (*WVLT*) as discussed in [26] for sliding-windows. EH and DW are deterministic structures that provide theoretically ϵ-approximate results in COUNT and SUM queries for positive integers. However, it is proven [11] that for general SUM queries that also include negative numbers, providing theoretical guarantees requires $\Omega(W)$ bits and these methods cease to work. As the guarantees of our method are computed while constructing the synopsis and are not theoretical, we demonstrate that our approach provides near optimal results even for the case of arbitrary data values. Henceforth, *SW2G* (Sliding Window Wavelets with Guarantees) denotes the approach proposed in this work.

We have implemented all single-stream algorithms in Java 8, except for the exponential histograms where we use the Scala implementation of [1]. For the distributed part of our work, we use the Apache Flink 1.6 stream processing framework. Our Flink implementation for distributed exponential histograms is based on the Java code of [31].

Queries. The workloads we consider are mainly range queries (COUNT, SUM, AVG) in the form we describe in Section 3. This is the most common query type in the sliding-window context. Moreover, the performance of wavelets in sliding-window aggregates has not been studied before. In order to demonstrate the generality of our approach, in Section 7.3, we also consider aggregates over arbitrary ranges within the active window as well as point queries. In all experiments, we apply workloads of 50 queries each and report the average accuracy and memory utilization in each workload.

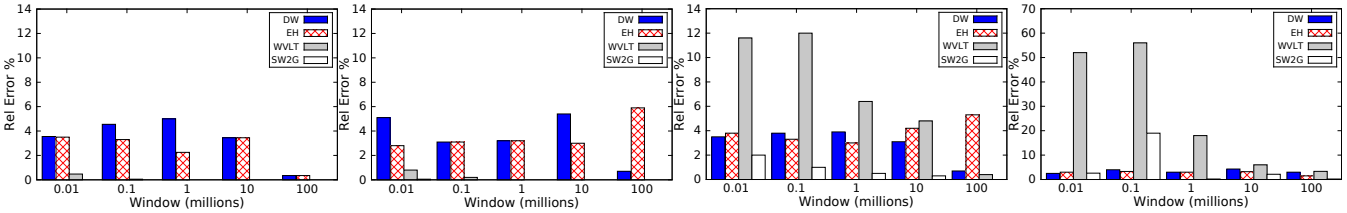
Datasets. For the assessment of the proposed algorithms, we use both synthetic and real data. Synthetic data is used for experimenting with various data distributions. The generated data values lie in the range $[0 - 1000]$ and follow a uniform, normal or highly biased ($s = 2$) zipf distribution. As real data, we use the sensor measurements provided by NOAA [2]. For our experiments we use temperature (noaaTemp) and wind-speed (noaaSpeed) time-series. NOAA time-series consist of both positive and negative numerical data.

Platform. All single-stream algorithms are executed on top of a server with 8 Intel(R) Xeon(R) CPU E5405 @ 2.00GHz processors and 8 GB of main memory. For the experiments on distributed streams, we use a cluster of 4 machines with the same processing and memory capabilities.

7.1 Positive Integers

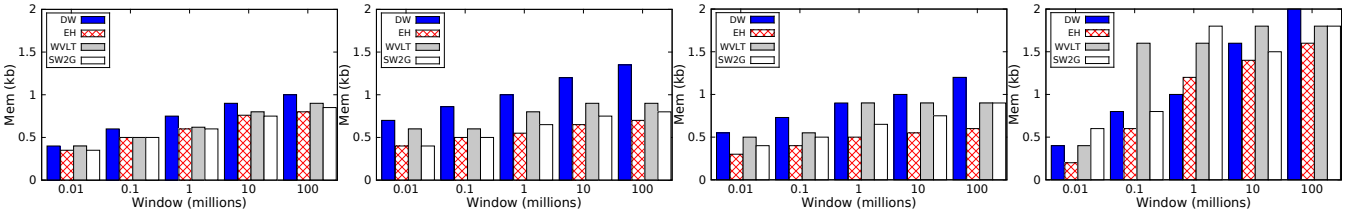
In our first experiment, we evaluate the proposed method over a single stream of positive integers. As this is the only case where EH and DW can be applied, we can directly compare them with our approach.

In Figure 5, we present accuracy results for various data distributions and window sizes. We consider streams of 400 millions data points and window sizes in the range of $[10k, 100M]$. At random time moments, we query each structure for the COUNT, SUM or AVG of the stream elements in the last W time units. In the case of



(a) COUNT-uniform (b) SUM-uniform values (c) SUM-zipfian values (d) AVG-noaaTemp

Figure 5: Relative error in streams of positive integers (query length = W).



(a) COUNT-uniform (b) SUM-uniform values (c) SUM-zipfian values (d) AVG-noaaTemp

Figure 6: Memory consumption in streams of positive integers (query length = W).

the noaaTemp dataset, we compute a more complex query, where we filter the stream *on the fly* and compute the average temperature only for tuples having a temperature larger than $86F$. In favor of a fair comparison, we tune each algorithm to use approximately the same amount of memory. In EH and DW, the tuning knob of memory consumption is the guaranteed error ϵ and for the wavelet-based techniques, the available budget B. We set $\epsilon = 0.1$ for EH and $\epsilon = 0.2$ for DW.

EH and DW respect the theoretical guarantees and both achieve an average error near 4% for all datasets. The vanilla wavelet method, while performing well in uniform distributions, it presents considerably large errors for the other two datasets. Particularly for noaaTemp, as WVLT can reach up to a 60% relative error, it cannot provide an acceptable solution to the problem. By being near precise in all demonstrated cases, SW2G appears to be the best alternative for approximating the examined datasets. As Lemma 5.2 indicates, larger windows favor our method since due to the factor $\frac{1}{2(logW-1)}$, we have to approximate only a tiny part of the active window. Moreover, in all four datasets, the guarantees SW2G provides are tighter than 10%.

We remind that in sliding window range queries, an error is introduced only due to the overlap of the query range with the last bucket of the active window. Techniques like EH and DW control the size of the last bucket in a way that provides theoretical guarantees. By putting a constraint on the maximum level of a sub-tree, SW2G also controls the size of the last bucket. WVLT is not designed with range queries in mind; the whole window can be covered by a single tree of size W. Thus, WVLT presents an unstable behavior where quality highly depends on the current state and structure of the error-tree.

The overlap with the last bucket is also the cause for the high quality results of SW2G compared to EH and DW. Both these techniques assume that half of the last bucket's items lie in the range of interest. On the other hand, wavelet-based techniques can more accurately approximate the number of items that should be considered. By combining the powerful wavelet structure and the idea of limiting the maximum size of an error-tree, SW2G manages to present the best results in all cases.

Figure 6 illustrates the corresponding memory consumption. We observe that as window size increases, we need to consume more memory in order to preserve error guarantees. We see that DW is the most expensive among the evaluated methods. Moreover, we observe that COUNT queries use slightly less memory than SUM ones and AVG queries need the largest amount of memory since we have to maintain two structures for each algorithm: one that keeps track of counts and one for sums. However, in all cases, memory consumption is negligible. In the case of $W = 100M$, the footprint of the exact solution is 400 MB, while all approximation techniques need only around a single kilobyte. Especially in the case of SW2G, 1 Kb is enough for achieving a relative error lower than 1% in all demonstrated cases.

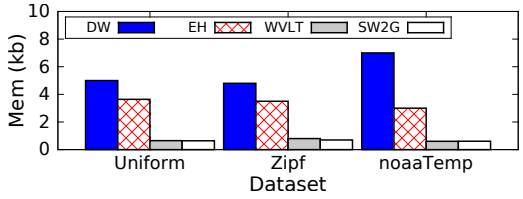


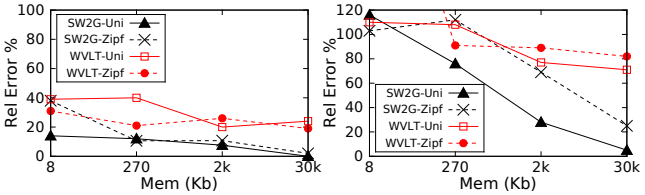
Figure 7: Memory for $\epsilon = 0.01$

In Figure 7, we present the memory EH and DW need in order to achieve the same performance as SW2G when $W = 10M$. For this purpose, we set $\epsilon = 0.01$ for both EH and DW and issue SUM queries to all datasets. In the case of noaaTemp, we notice that DW needs 7× and EH 4× the memory that SW2G requires.

As window size does not affect accuracy, in all subsequent experiments we set W to $10M$.

7.2 Streams of generic numerical data

In the case of positive numbers, we demonstrated that our approach outperforms existing techniques. In this Section we examine the applicability and efficiency of SW2G in more general cases, where the stream also includes negative values. We experiment with SUM queries in real and synthetic data. As synthetic distributions we use uniform and zipf, where each data point d_i is drawn from range $[0, 1000]$ and is converted to the corresponding negative value $-d_i$ with a probability of $\frac{1}{2}$. Since EH and DW do not work for negative numbers, they are not considered for these experiments.



(a) Query length = W (b) Query length =$W/8$

Figure 8: Relative error in streams of arbitrary numerical data.

First, we compute queries of length W and $W/8$ on the noaaTemp and noaaSpeed datasets. As the value distribution of the NOAA datasets does not present a great variance, it can be easily approximated by wavelets. As such, both SW2G and WVLT achieved relative errors less than 1% in both workloads.

In order to stress wavelet algorithms, we use the synthetic distributions we described. As each subsequent data point can vary from −1000 to 1000 large discontinuities appear and the distribution becomes hard to approximate.

Figure 8 illustrates relative error with respect to the amount of memory we use for the synopsis. We observe that for both distributions and query lengths, SW2G converges better than WVLT as we increase memory. In the case where the query is applied over the whole window, a budget size of $W/10$ is enough to achieve an error less than 10% both for the uniform and the zipfian data.

7.3 General range and point queries

We also evaluate SW2G and WVLT in other query types such as aggregates over arbitrary ranges and point queries.

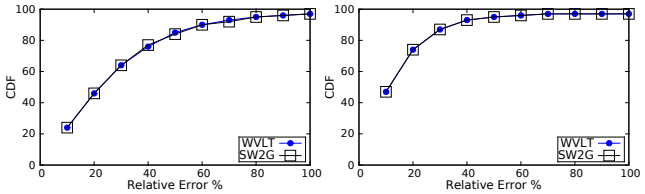
Table 2 shows the results for a workload of random AVG queries where the limits of the queried ranges are selected at random. For this experiment, we use a $200kb$ synopsis. Moreover, all datasets contain both positive and negative values.

Depending on the data distribution, the performance of both algorithms varies. However, SW2G consistently outperforms WVLT, demonstrating this way our contribution to the wavelet structure for tackling range queries.

Figure 9 demonstrates the results for point queries. The workload we apply in this case is the following: Every W time units, we ask for the value of every item in the range $[t-W, t]$, where t is the current time. Both algorithms achieve the same accuracy in all examined

Table 2: Relative error for AVG queries with random ranges

Dataset	SW2G	WVLT	% Gain
uniform	27	126	78
zipf	53	61	13
noaaTemp	0.12	1.04	88
noaaSpeed	0.75	5.4	86



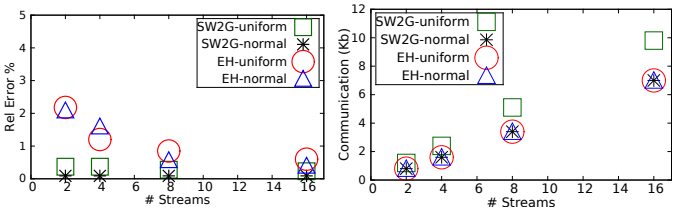
(a) noaaSpeed, $B = W/100$ (b) noaaSpeed, $B = W/10$

Figure 9: CDF of relative error in point queries.

cases. Thus, while optimizing for range queries, the performance of our algorithm in point queries is not compromised. As noticed in Figure 9a, the distribution of the noaaSpeed dataset needs more space than $W/100$ in order to be accurately represented. However, error drops as space budget is increased. Having available $W/10$ of memory leads to an error less than 20% for the 70% of the workload.

7.4 Distributed Streams

We examine the behavior of SW2G in a distributed environment of multiple streams. In this scenario, we track range queries in the average of the streams. Each stream maintains a local synopsis; a coordinator node collects wavelet coefficients from all streams and composes a global synopsis which is used to answer queries. We compare SW2G with the distributed version of EH which is described in [31]. For distributed exponential histograms we set an error of $\epsilon = 0.1$ both for the coordinator and all remote streams.



(a) Error (b) Communication

Figure 10: Relative error and communication cost in distributed streams.

Figure 10 shows the real relative error and the communication cost for synthetic data of uniform and normal distributions. We present results for 2 up to 16 streams. For each setup we plot the average error of the issued workload and the total bytes sent over the network each time the streams emit their local synopses. Although EH are configured with $\epsilon = 0.1$ and according to [31] are expected to have an error up to $2\epsilon + \epsilon^2 = 21\%$, they present a maximum error of only 2%. SW2G performs even better and is almost exact in all cases. Furthermore, the guarantees it provides do not exceed 9%. As expected, communication increases linearly to the number of

streams for both techniques. Moreover, EH present slightly better communication cost than SW2G for the uniform distribution.

8 CONCLUSION

In this work we investigate the usability of wavelets for approximating streams under the sliding-window model. As wavelets have been extensively studied for point queries, we design algorithms carefully optimized for the case of range queries. Traditional techniques like exponential histograms and deterministic waves are restricted to track a single type of query and only work with streams of positive numbers. With the use of wavelets we opt for a more generic solution where the same structure can be used to answer both aggregates and point queries over arbitrary ranges and data. Moreover, we show that our algorithm can be also applied to distributed environments where multiple streams produce information and the computation of an aggregate is requested over their union. Our experiments show that by yielding near precise results, the proposed method outperforms other state-of-the-art techniques for computing basic aggregates, while it preserves the quality of classic wavelet algorithms for point queries.

9 ACKNOWLEDGMENT

This work has been supported by the European Commission in terms of the H2020 E2Data Project (780245).

REFERENCES

[1] [n. d.]. Abstract Algebra for Scala. https://twitter.github.io/algebird/.
[2] [n. d.]. National Oceanic and Atmospheric Administration. https://www1.ncdc.noaa.gov/pub/data/noaa/.
[3] Brain Babcock, Mayur Datar, Rajeev Motwani, and Liadan O'Callaghan. 2003. Maintaining variance and k-medians over data stream windows. In *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 234–243.
[4] Costas Busch and Srikanta Tirthapura. 2007. A deterministic algorithm for summarizing asynchronous streams over a sliding window. In *Annual Symposium on Theoretical Aspects of Computer Science*. Springer, 465–476.
[5] Don Carney, UÇğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. 2002. Monitoring streams: a new class of data management applications. In *Proceedings of the 28th international conference on Very Large Data Bases*. VLDB Endowment, 215–226.
[6] Kaushik Chakrabarti, Minos Garofalakis, Rajeev Rastogi, and Kyuseok Shim. 2001. Approximate query processing using wavelets. *The VLDB JournalâĂŤThe International Journal on Very Large Data Bases* 10, 2-3 (2001), 199–223.
[7] Ho-Leung Chan, Tak-Wah Lam, Lap-Kei Lee, and Hing-Fung Ting. 2012. Continuous monitoring of distributed data streams over a time-based sliding window. *Algorithmica* 62, 3-4 (2012), 1088–1111.
[8] Edith Cohen and Martin Strauss. 2003. Maintaining time-decaying stream aggregates. In *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 223–233.
[9] Graham Cormode, Minos Garofalakis, and Dimitris Sacharidis. 2006. Fast approximate wavelet tracking on streams. In *International Conference on Extending Database Technology*. Springer, 4–22.
[10] Corinna Cortes, Kathleen Fisher, Daryl Pregibon, and Anne Rogers. 2000. Hancock: a language for extracting signatures from data streams. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 9–17.
[11] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. 2002. Maintaining stream statistics over sliding windows. *SIAM journal on computing* 31, 6 (2002), 1794–1813.
[12] Sumit Ganguly, Minos Garofalakis, Rajeev Rastogi, and Krishan Sabnani. 2007. Streaming algorithms for robust, real-time detection of ddos attacks. In *Distributed Computing Systems, 2007. ICDCS'07. 27th International Conference on*. IEEE, 4–4.
[13] Minos Garofalakis and Phillip B Gibbons. 2002. Wavelet synopses with error guarantees. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. ACM, 476–487.
[14] Minos Garofalakis and Amit Kumar. 2004. Deterministic wavelet thresholding for maximum-error metrics. In *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 166–176.
[15] Phillip B Gibbons and Srikanta Tirthapura. 2002. Distributed streams algorithms for sliding windows. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*. ACM, 63–72.
[16] Anna C Gilbert, Ioannis Kotidis, Shanmugavelayutham Muthukrishnan, and Martin J Strauss. 2007. Method and apparatus for using wavelets to produce data summaries. US Patent 7,296,014.
[17] Anna C Gilbert, Yannis Kotidis, S Muthukrishnan, and Martin Strauss. 2001. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *Vldb*, Vol. 1. 79–88.
[18] Anna C Gilbert, Yannis Kotidis, S Muthukrishnan, and Martin J Strauss. 2003. One-pass wavelet decompositions of data streams. *IEEE Transactions on Knowledge & Data Engineering* 3 (2003), 541–554.
[19] Sudipto Guha. 2005. Space efficiency in synopsis construction algorithms. In *Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment, 409–420.
[20] Sudipto Guha and Boulos Harb. 2005. Wavelet synopsis for data streams: minimizing non-euclidean error. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*. ACM, 88–97.
[21] Sudipto Guha and Boulos Harb. 2008. Approximation algorithms for wavelet transform coding of data streams. *IEEE Transactions on Information Theory* 54, 2 (2008), 811–830.
[22] Panagiotis Karras and Nikos Mamoulis. 2005. One-pass wavelet synopses for maximum-error metrics. In *Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment, 421–432.
[23] Panagiotis Karras and Nikos Mamoulis. 2007. The Haar+ tree: a refined synopsis data structure. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*. IEEE, 436–445.
[24] Panagiotis Karras, Dimitris Sacharidis, and Nikos Mamoulis. 2007. Exploiting duality in summarization with deterministic guarantees. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 380–389.
[25] Tao Li, Qi Li, Shenghuo Zhu, and Mitsunori Ogihara. 2002. A survey on wavelet applications in data mining. *ACM SIGKDD Explorations Newsletter* 4, 2 (2002), 49–68.
[26] Ken-Hao Liu, Wei-Guang Teng, and Ming-Syan Chen. 2010. Dynamic wavelet synopses management over sliding windows in sensor networks. *IEEE Transactions on Knowledge and Data Engineering* 22, 2 (2010), 193–206.
[27] Samuel Madden and Michael J Franklin. 2002. Fjording the stream: An architecture for queries over streaming sensor data. In *Data Engineering, 2002. Proceedings. 18th International Conference on*. IEEE, 555–566.
[28] Yossi Matias and Leon Portman. 2003. *Workload-based wavelet synopses*. Technical Report. Technical report, Department of Computer Science, Tel Aviv University.
[29] Yossi Matias, Jeffrey Scott Vitter, and Min Wang. 1998. Wavelet-based histograms for selectivity estimation. In *ACM SIGMoD Record*, Vol. 27. ACM, 448–459.
[30] S Muthukrishnan. 2005. Subquadratic algorithms for workload-aware haar wavelet synopses. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer, 285–296.
[31] Odysseas Papapetrou, Minos Garofalakis, and Antonios Deligiannakis. 2012. Sketch-based querying of distributed sliding-window data streams. *Proceedings of the VLDB Endowment* 5, 10 (2012), 992–1003.
[32] Lin Qiao, Divyakant Agrawal, and Amr El Abbadi. 2003. Supporting sliding window queries for continuous data streams. In *Scientific and Statistical Database Management, 2003. 15th International Conference on*. IEEE, 85–94.
[33] Nicoló Rivetti, Yann Busnel, and Achour Mostefaoui. 2015. *Efficiently Summarizing Distributed Data Streams over Sliding Windows*. Ph.D. Dissertation. LINA-University of Nantes; Centre de Recherche en Économie et Statistique; Inria Rennes Bretagne Atlantique.
[34] Zubair Shah, Abdun Naser Mahmood, Zahir Tari, and Albert Y Zomaya. 2017. A technique for efficient query estimation over distributed data streams. *IEEE Transactions on Parallel & Distributed Systems* 10 (2017), 2770–2783.
[35] Eric J Stollnitz, Tony D DeRose, and David H Salesin. 1996. *Wavelets for computer graphics: theory and applications*. Morgan Kaufmann.
[36] Jeffrey Scott Vitter and Min Wang. 1999. Approximate computation of multidimensional aggregates of sparse data using wavelets. In *Acm Sigmod Record*, Vol. 28. ACM, 193–204.
[37] Bojian Xu, Srikanta Tirthapura, and Costas Busch. 2008. Sketching asynchronous data streams over sliding windows. *Distributed Computing* 20, 5 (2008), 359–374.
[38] Yong Yao, Johannes Gehrke, et al. 2003. Query Processing in Sensor Networks.. In *Cidr*. 233–244.
[39] Yunyue Zhu and Dennis Shasha. 2002. StatStream: Statistical Monitoring of Thousands of Data Streams in Real Time** Work supported in part by US NSF grants IIS-9988345 and N2010: 0115586.. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier, 358–369.