Distributed and Parallel Databases manuscript No.
(will be inserted by the editor)

Workload-Aware Wavelet Synopses for Sliding Window Aggregates

Ioannis Mytilinis · Dimitrios Tsoumakos · Nectarios Koziris

Received: date / Accepted: date

Abstract 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 online fashion. For the efficient construction of such synopses, we propose wavelet-based algorithms that provide deterministic guarantees and produce near exact results for a variety of data distributions. Furthermore, we show how accuracy can be further improved when workload information is known. For this purpose, we propose a workload-aware streaming system that trade-offs accuracy with synopsis' construction throughput. The conducted experiments indicate that with only a 15% penalty in throughput, the proposed system produces fairly accurate results even for the most adversarial distributions.

Keywords Stream Processing · Sliding-Window · Approximate Query Processing · Wavelets · Workload-Aware Synopses · Range Queries

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

I. Mytilinis
National Technical University of Athens
E-mail: gmytil@cslab.ece.ntua.gr

D. Tsoumakos
Ionian University
E-mail: dtsouma@ionio.gr

N. Koziris
National Technical University of Athens
E-mail: nkoziris@cslab.ece.ntua.gr

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 [Carney et al. (2002); Madden and Franklin (2002); Yao et al. (2003)], datacenter monitoring [Ganguly et al. (2007)], financial data trackers [Zhu and Shasha (2002)], and real-time analysis of various transaction logs [Cortes et al. (2000)].

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 [Cohen and Strauss (2003)]. The *sliding-window* model [Datar et al. (2002)] 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 [Datar et al. (2002); Gibbons and Tirthapura (2002); Qiao et al. (2003); Xu et al. (2008).]

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 [Stollnitz et al. (1996)] provides a very effective data reduction tool, with applications in data mining [Li et al. (2002)], selectivity estimation [Matias et al. (1998)], approximate and aggregate query processing of massive relational tables [Vitter and Wang (1999); Chakrabarti et al. (2001)] and data streams [Gilbert et al. (2001); Cormode et al. (2006)]. In simple terms, a wavelet synopsis is extracted by applying the wavelet decomposition on an input collection 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 [Chakrabarti et al. (2001)].

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 such as COUNT, SUM and AVG, since more complex queries in sliding-windows [Papapetrou et al. (2012)] usually need to compute such basic aggregates under the hood. In order to provide theoretical guarantees, traditional techniques for the problem, like exponential histograms [Datar et al. (2002)] and deterministic waves [Gibbons and Tirthapura (2002)] 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 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 such as 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 *SW2G*: 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 discuss the required modifications in the proposed algorithm in order to support out-of-order arrivals.
- We introduce a workload-aware streaming system, which exploits available query workload information to significantly boost accuracy in adversarial distributions, where SW2G fails to provide precise results.
- 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 provides formal definitions and theoretical background on the problem. Section 3 presents a literature review for the applications of wavelets in approximate query processing and for sliding-window techniques. In Section 4, we present the SW2G algorithm for maintaining the synopsis in real-time and in Section 5 we show how basic statistics are computed. Section 6 presents an extension for distributed environments and Section 7 discusses out-of-order arrivals. In Section 8, we propose a workload-aware system that improves on accuracy when workload information exists. Section 9 demonstrates the experimental evaluation of our work and Section 10 concludes the paper.

2 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.

2.1 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 sliding-window data stream, where data elements are expected to arrive in the stream-order. Such a stream is formally defined in Definition 1. Henceforth, we are going to simply use the term *stream* in order to describe such a data sequence.

Definition 1 (Ordered Stream) An ordered data stream is an infinite sequence of tuples in the form: $S = \{(t_1, v_1), (t_2, v_2), ...\}, t_1 \leq t_2 \leq ...$, where t_i denotes the index of tuple i and v_i its value.

The proposed algorithms support both count- and time-based streams. In the case of time-based streams, the tuple index corresponds to the arrival time, while in the count-based scenario it corresponds to the position in the stream. Since count-based streams constitute a special case of time-based ones, we abusively use the notion of time in both cases.

Both the sliding-window point and range queries we tackle are defined in Definition 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 within the window.

Definition 2 Let S be a stream, t the current time and W the window size.

- A **sliding-window point query** $P(t_q)$ 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** $AGG(t_q)$ on S returns an estimation for an aggregate $AGG \in \{SUM, COUNT, AVG\}$ computed over the time range: $[t_q, t]$, 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 9, we also investigate queries of the form $[s, e]$, where $t - W \leq s \leq e \leq t$.

2.2 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 [Stollnitz et al. (1996)]; it is computationally efficient and

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	$[6]$	$[1]$

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 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.

The *error-tree*, introduced in [Matias et al. (1998)], 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 $(i = 0, ..., 7)$ is associated with a wavelet coefficient value, and each leaf d_i $(i = 0, ..., 7)$ 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 leaves 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

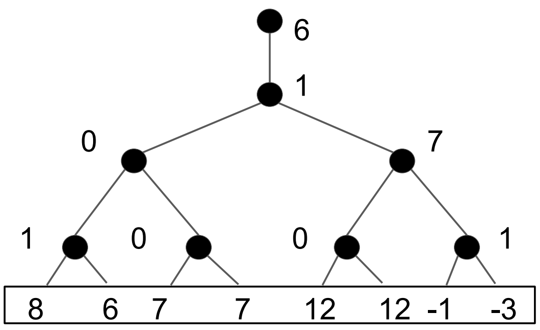


Fig. 1 An error tree that illustrates the hierarchical structure of the Haar wavelet decomposition

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:

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

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.

3 Related Work

Wavelets. In the seminal work of Chakrabarti et al. (2001), 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 [Garofalakis and Gibbons (2002); Garofalakis and Kumar (2004); Guha (2005); Karras and Mamoulis (2007); Karras et al. (2007); Muthukrishnan (2005)] have been proposed for this task. In order to alleviate the complexity burden, greedy algorithms [Karras and Mamoulis (2005); Matias and Portman (2003)] have also been proposed.

All approaches discussed thus far refer to batch jobs, where algorithms are applied to static data. Gilbert et al. (2001, 2003) 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. Cormode et al. (2006) present a similar sketching technique, that improves on updates' efficiency. Streaming techniques have also been proposed for the optimization of the L_∞ norm. Guha and Harb (2005, 2008) 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 proposed by Karras and Mamoulis (2005). However, the algorithm of Karras et al. does not support sliding-window queries.

Workload-aware wavelet synopses have also been widely studied for both point and range queries [Muthukrishnan (2005); Matias and Portman (2003); Guha et al. (2008)]. Nevertheless, to the best of our knowledge, there is no other wavelet-based work that targets workload-aware techniques under the sliding-window model.

The only wavelet-based algorithms that exist in the literature and consider sliding-window streams are the ones of Liu et al. (2010) and Mytilinis et al. (2019). Liu et al. (2010) mainly cover point queries and do not take into account range queries such as COUNT and SUM, which are the most basic and common queries in sliding-window streams. This paper builds upon the work of Mytilinis et al. (2019). However, in our previous work, we do not consider out-of-order arrivals and also we do not handle the workload-aware case of the problem.

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 sublinear space and time (typically, poly-logarithmic) in the window size W. Exponential histograms [Datar et al. (2002)] are a state-of-the-art deterministic technique for maintaining ϵ-approximate counts and sums over sliding windows, using $O\left(\frac{1}{\epsilon}log^2W\right)$ space. Deterministic waves [Gibbons and Tirthapura

(2002)] 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, Gibbons and Tirthapura (2002) also present 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. Xu et al. (2008) 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 (2007) propose a deterministic structure for handling out-of-order arrivals in sliding windows. Similar to other deterministic structures, this structure does not allow composition and focuses only on basic counts and sums. Finally, Chan et al. (2012) 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 [Babcock et al. (2003)], heavy hitters, inner products and self-joins [Papapetrou et al. (2012); Shah et al. (2017); Rivetti et al. (2015)]. 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.

4 Dynamic Synopsis Maintenance

In this Section, we present an efficient algorithm for the computation and online maintenance of wavelet synopses. 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 such as Arduinos[1], that are often met in IoT scenarios, possess memory in the order of KB. Thus, a space budget B should be defined and cap the number of retained wavelet coefficients. The theoretical analysis we provide is inline with previous research and synopses poly-logarithmic in the window size are considered.

[1] https://www.arduino.cc/

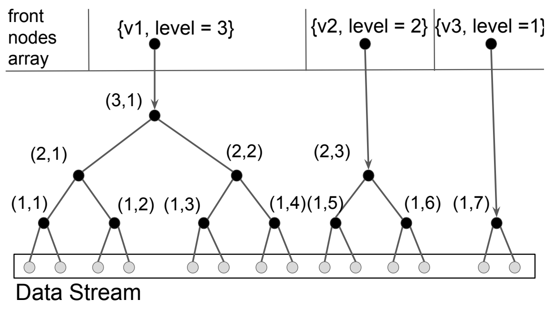


Fig. 2 Error-tree for streaming data.

4.1 Streaming Error-Tree

Similarly to previous works, we operate on the streaming version of an error-tree [Liu et al. (2010); Karras and Mamoulis (2005)]. 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. Hence, 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 heights: 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, the *front nodes array* structure is used. For each sub-tree, that we want to track, we create a *fnode* (i.e., a new element of the front nodes array) 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: $2 \cdot o_j - 1 < (2 \cdot o_i - 1) \cdot 2^{l_i - l_j}$.

This work exploits the sliding-window and proposes 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 $o'_i = \left[(o_i - 1)\, mod \frac{W}{2^{l_i}} + 1\right]$ that uses $log\frac{W}{2^{l_i}}$ 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.

4.2 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; $d_1$, $d_2$ = read($S$);
4      $wSynopsis$.deleteExpired($currentTime$, $W$);
5      $wSynopsis$.insert($currTime$,$W$,$d_1$, $d_2$);
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 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.

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.

Algorithm 2: Insert

```
input: Number of arrived items $N$, window size $W$, item $d_1$,item $d_2$
f = fnode with lowest level; tmp = null; l = 0
maxLevel = $log\left(\frac{W}{logW}\right)$
while $N > 0$ and $N$ mod 2 = 0 do
    N = N / 2; l = l + 1
    if $l > maxLevel$ then break
    if $tmp = null$ then
        avg = ($d_1$ + $d_2$) / 2; v = ($d_1$ - $d_2$) / 2
        minCf = maxCf = v
    else
        avg = (avg + tmp) / 2; v = tmp - avg
        minCf = $min$ (prevFnode.minCf, tmpMin, v)
        maxCf = $max$ (prevFnode.maxCf, tmpMax, v)
    $c_i$ = new WaveletCoef($l_i$ = l, $o_i$ = N, value = v)
    $c_i$.maxCoefInSubtree = maxCf
    $c_i$.minCoefInSubtree = minCf
    if $c_i \neq 0$ then put $c_i$ in min-heap
    delete fnode below f
    if no fnode in level $l$ then
        f = new Fnode(level = $l$, value = avg)
        f.minCf = minCf; f.maxCf = maxCf
        if $l < maxLevel$ then frontNodesArray.add(f)
        else topLevelFnodes.add(f)
    else
        tmp = f.value
        tmpMin = f.minCf; tmpMax = f.maxCf;
    if f.pointer = null then f.pointer = $c_i$
    f = fnode at next level
```

According to the proposed 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 more accurate computation of range queries.

The 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 the query answering mechanism is described.

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 $\lceil log\left(\frac{W}{logW}\right)\rceil$. 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 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 $\lceil log\left(\frac{W}{logW}\right)\rceil$ levels, results in storing at most $O\left(logW\right)$ 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 $O\left(logW\right)$ 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 $O\left(logW\right)$ fnodes is the first power of 2 that is larger than $\frac{W}{c \cdot logW}$ and has $M = \lceil log\left(\frac{W}{c \cdot logW}\right)\rceil$ levels. However, the construction process of a wavelet tree is such that we may have more than $\lceil\frac{W}{k}\rceil$ sub-trees in the window. 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 $M-1$ sub-trees of levels $l = 1, .., M-1$. This way, there are at most $\lceil\frac{W}{k}\rceil - 1 + M - 1 = \lceil\frac{W}{k}\rceil + \lceil log\left(\frac{W}{c \cdot logW}\right)\rceil - 2$ sub-trees and thus fnodes in the window. As we want to save space, we set $c = 1$ and in the worst case we have $logW + log\left(\frac{W}{logW}\right) = O(logW)$ fnodes. □

The cost for inserting new elements in the wavelet synopsis is given by Lemma 2.

Lemma 2 (Insertion Time) *Considering a synopsis size of $B = O\left(logW\right)$, an arriving pair of data items leads to a worst case insertion time of $O\left(log\frac{W}{logW} \cdot loglogW\right)$ and $\Theta\left(log\frac{W}{logW}\right)$ 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 $\lceil log\left(\frac{W}{logW}\right)\rceil$, it follows that this is also the maximum number of operations that an insert-update can cause. Moreover, since the synopsis should occupy only poly-logarithmic space, we assume a min-heap of size $B = O\left(logW\right)$. Thus, the worst-case insertion in the heap is $O\left(loglogW\right)$. 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 $\Theta\left(1\right)$ 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:

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

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

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

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 for the sub-trees there is the constraint that the maximum size N is the first power of 2 that is greater than $\frac{W}{logW}$, the complexity becomes: $\Theta\left(log\frac{W}{logW} - \frac{1}{N}\right) = \Theta\left(log\frac{W}{logW}\right)$. □

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 [Karras and Mamoulis (2005)]. 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 this work, the L_∞ norm is used and the min-heap is implemented as a binary heap. Lemma 3 gives the cost of deletions either due to expiration or budget excess.

Lemma 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. □

4.3 Error Guarantees

Regardless of which error-metric is optimized, the constructed synopsis should be able to provide queries with deterministic guarantees. As shown in [Karras and Mamoulis (2005)], providing guarantees for point queries demands each node to maintain the maximum and minimum signed errors of its left and right sub-trees.

This work also provides deterministic guarantees for range queries. As mentioned in Section 2, 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 it is explained 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 $P(t_q)$ are answered as explained in Section 2, i.e., $P(t_q) = \Sigma_{c_j \in path_q} \delta_{qj} \cdot c_j + f.value$, 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 depicts a range query $AGG(t_q)$. The range of interest $[t_q, t_{now}]$ 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.

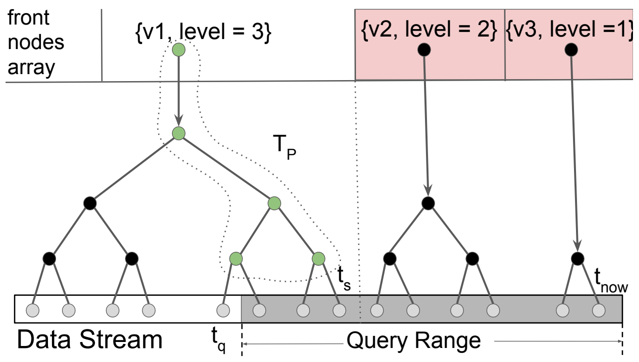


Fig. 3 Range query answering

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 4 holds.

Lemma 4 *Let $avg\left(\cdot\right)$ and $n\left(\cdot\right)$ 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:*

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

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 $\{T_i, ..., T_j\}$ 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: $SUM_{[t_q,t_s]} = \sum_{c_j \in path_{t_q}} c_j x_j + f_p.value \cdot (t_s - t_q + 1)$ 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 and note the time when the update takes place (line 11). When the stream is inactive and no data arrives, we keep the system idle. The next time a tuple arrives after an inactivity period, we insert $t - lastTimeActive - 1$ zero values to both BW and VW (line 6). 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 the proposed approach. Exponential histograms and waves do the same in order to support both COUNT and SUM queries.

Algorithm 3: BW-VW updates

```
1  Initialize $BW, VW$;
2  $lastTimeActive = 0$;
3  for every time tick $t$ do
4      $(t, v)$ = listenToStream();
5      if $(t, v) \neq null$ then
6          for $t^*$ in $[lastTimeActive + 1, t)$ do
7              $BW.insert((t^*, 0))$;
8              $VW.insert((t^*, 0))$;
9          $BW.insert((t, 1))$;
10         $VW.insert((t, v))$;
11         $lastTimeActive = t$;
```

At this point, we need to discuss the performance of the update mechanism. Inserting a single data item into the structure is really efficient (sub-millisecond latency). However, as we must maintain the mapping between the time- and wavelet-domains, we also need to pad the stream with zero values. Inserting a zero-value at a time would signify low latencies but would keep the CPU always busy. On the other hand, a batch insertion of zeroes can lead to a high update cost, if the stream remains inactive for a large period of time. We advocate that by following the batch-update style, we decrease CPU utilization without severely harming update latency.

The design of our algorithm targets streams where data points are expected to arrive at each time tick. Assuming that the user knows some rough characteristics of the underlying stream, she can define the duration of a time tick based on the expected arrival rate. For example, if we expect items to arrive once every minute, we are not going to define time ticks in the millisecond-level, as this would always lead to batch updates. Therefore, the arrival of a data item can occasionally trigger a batch update but we do not expect it to happen often.

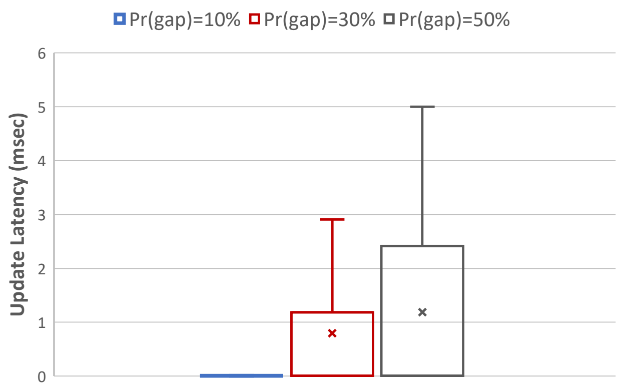


Fig. 4 Update latency for various frequencies of inactivity gaps.

In order to validate this claim, we conduct the following experiment: we track the SUM of a data stream over a window of $1M$ data points. Each data item arrives at the next time tick with probability $1 - p$, while an inactivity gap appears with probability p. The length of an inactivity gap is uniformly distributed in $[10 - 10K]$ time ticks. We consider three distinct cases: (i) $p = 10\%$, (ii) $p = 30\%$ and (iii) $p = 50\%$. The results are shown in Figure 4. Inactivity periods affect indeed the performance of the system but even in the worst case, where gaps appear really often ($p = 50\%$), the average latency (indicated by the x marker) is only near 1 msec.

We now discuss how this structure is used for query processing. 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.

Having described the query answering mechanism of the proposed algorithm, we now discuss the impact of limiting the maximum level of a sub-tree. We saw 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 *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: $\frac{|A|}{|E|} \geq \frac{1}{2logW}$.*

Proof. We distinguish two cases depending on whether A overlaps with a sub-tree of height $\lceil log\frac{W}{logW}\rceil$ or not. Let us initially assume that A overlaps with a sub-tree of size 2^k, with $k < \lceil log\frac{W}{logW}\rceil$. The maximum length of the range we need to approximate is $|A| = 2^{k-1}$. By the wavelet construction, it is guaranteed that there can be up to $k-1$ trees in E of sizes $2, 4, .., 2^{k-1}$ and thus $|E| \leq \sum_{i=2}^{k-1} 2^i = 2^k$. It follows: $\frac{|A|}{|E|} \geq \frac{1}{2}$. We now consider the case where A overlaps with a sub-tree of size M, where M is the first power of 2 which is greater than $\frac{W}{logW}$. In that case, it holds that $|E| \leq W$ and $|A| = \frac{M}{2}$, and so we have $\frac{|A|}{|E|} \geq \frac{M}{2W} \geq \frac{\frac{W}{logW}}{2W} = \frac{1}{2logW}$. □

Lemma 5 implies that for range queries of length near to W, the proposed method may have to approximate only the $\frac{1}{2logW}$ 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 queried range. This is a direct consequence of limiting the maximum level a sub-tree can have. According to the proof, the corresponding ratio in classic wavelets is $\frac{1}{2}$ in the best case.

As factor $\frac{1}{2logW}$ bounds the range we have to approximate 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, the experiments of Section 9 show that the proposed approach is very robust and that for queries of length W high quality results are achieved for all examined datasets, both real and synthetic.

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 this work does, 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, our method performs better when applied to positive numbers, in Section 9, it is shown that it can also be efficiently applied to streams of arbitrary numerical data.

6 Distributed Wavelets For Streams

This paper also addresses 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 the described 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., $AGG\left(F\left(S_1, .., S_N\right)\right)$, while minimizing communication; collecting all streaming data is too costly in many real use-cases. Therefore, similarly to Gilbert et al. (2007), 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 $F\left(S_1, .., S_N\right)$. Thus, $WS\left(F\left(S_1, .., S_N\right)\right) = F\left(WS_1, .., WS_N\right)$ and we can approximate the query $AGG\left(F\left(S_1, .., S_N\right)\right)$ as $AGG\left(F\left(WS_1, .., WS_N\right)\right)$.

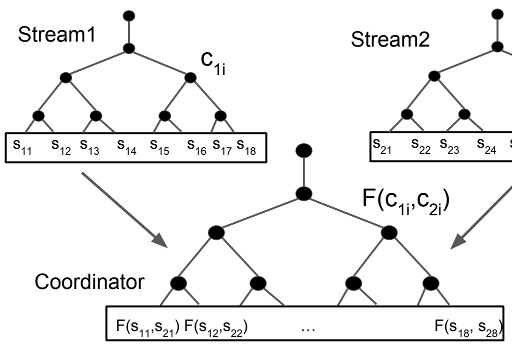

Stream1
c_{1i}
Stream2
c_{2i}
s_{11} s_{12} s_{13} s_{14} s_{15} s_{16} s_{17} s_{18}
s_{21} s_{22} s_{23} s_{24} s_{25} s_{26} s_{27} s_{28}
$F(c_{1i}, c_{2i})$
Coordinator
$F(s_{11}, s_{21})$ $F(s_{12}, s_{22})$ ... $F(s_{18}, s_{28})$

Fig. 5 Composition of individual wavelet synopses.

Figure 5 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 $F\left(s_{1i}, s_{2i}\right)$. 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 $cm_i = F\left(c_{1i}, c_{2i}\right)$, where cm_i is the corresponding coefficient in the error-tree of the coordinator. Thus, it suffices to aggregate the coefficients by index and compute the 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 *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:

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

□

7 Out-of-Order Arrivals

While so far we have considered time-based streams where items arrive in order, this is not a real restriction of the algorithm. In favor of completeness, in this Section it is described how the scheme can be generalized to handle out-of-order arrivals. In Algorithm 3, we saw that in case arrivals are in order but a discontinuity in time exists, i.e., the next arrived value has a timestamp $t = t_{now} + k, k > 1$, we pad the stream with zero-values. This way it is ensured that the wavelet transform is performed over a continuous time domain and the error-tree contains a path for each possible time t.

It is now described how we handle the case where a tuple (t_p, v) with $t_p < t_{now}$ arrives. The only restriction we have is: $t_p \geq t_{now} - W$, i.e., the tuple should lie within the active window. We first have to find the sub-tree where this tuple belongs. This can be accomplished by a linear scan over the fnodes. It is reminded that each fnode f maintains the start point $f.start$ of its coverage in time as well as its level $f.level$. Thus, the range it spans in time is $[f.start, f.start + 2^{f.level}]$. Then, for the $f.level$ levels of the sub-tree, we compute the contribution of value v to the wavelet nodes in $path_{t_p}$. The contribution of v to a wavelet node c_i with index (l_i, o_i) is $\delta_{ij} \frac{v}{2^{l_i}}$, where $\delta_{ij} = 1$ if $t_p \in \text{leftleaves}_{c_i}$ and -1 otherwise. Each of the newly computed coefficients $((l_i, o_i), \delta_{ij} \frac{v}{2^{l_i}})$ has to be inserted into the synopsis. If a coefficient $((l_i, o_i), v_{old})$ already exists for the index (l_i, o_i), then we just update its value and the quantities that help us provide error guarantees (e.g., MA-value). If the node that corresponds to index (l_i, o_i) has been deleted, the newly computed node is directly inserted into the synopsis. Nevertheless, in the latter case, the new coefficient misses some information (maximum/minimum errors and coefficients in sub-tree) for providing error-guarantees. For dealing with this issue, we can find its first available ancestor c_j in the error-tree and inherit that information from there. However, as the ancestor c_j covers a larger part of the time domain than c_i does, the max/min values it maintains are derived not only from the sub-tree rooted at c_i but from other sub-trees too. Hence, the error-guarantees will still hold but are expected to be loosened compared to the in-order case.

8 Workload Aware Synopses

Lemma 5 provides an intuition on why the proposed scheme works well in a variety of cases but also indicates some of its weaknesses: it does not provide theoretical guarantees and it is not expected to present a good behavior when the query range is significantly smaller than W. In this Section, we are going to demonstrate how we can boost performance in these cases too, assuming we have knowledge of the workload.

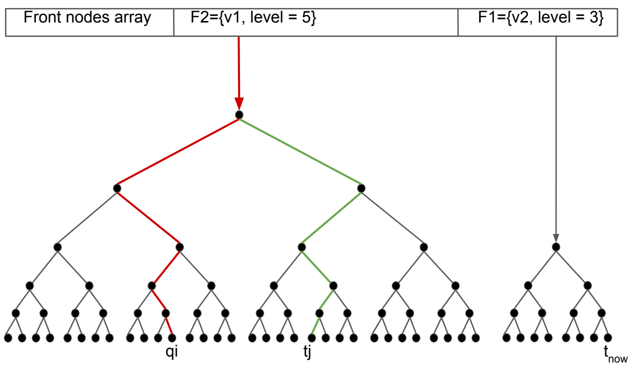


Fig. 6 Example demonstrating the pitfalls in workload-aware sliding-window synopses: If q_i is a query of interest, eventually all coefficients in paths $t_j > t_{now} - q_i$ will be requested. Hence, we have to delete coefficients that we know they will be important in the future.

We consider workload to be a set of fixed queries in the form $Q = \{q_1, q_2, ..., q_k\}$ which are known a priori and can be asked at any time. Each q_i represents a range query $[t_{now} - q_i, t_{now}]$ and thus it should be $0 \leq q_i < W$. The problem of constructing an optimal wavelet synopsis with respect to a set of range queries has been extensively studied [Guha et al. (2008)]. Guha et al. propose both DP and heuristic algorithms not only for prefix queries[2], which is our case, but also for the more general case of hierarchical range queries. However, they examine the static version of the problem where data is fixed and does not change over time. The real-time requirements we have, and the sliding-window model render the approach of Guha et al. (2008) inapplicable to our case.

In Section 4 we observed that in order to compute a range sum over $[t_j, t_{now}]$ we only need the coefficients $c_i \in path_{t_j}$. Then, the answer is derived by the fnodes of the sub-trees that are fully contained in the query and the term $\sum_{c_i \in path_{t_j}} c_i x_i$, where t_j belongs to the last sub-tree, that partially

[2] In prefix range queries, the start (or end) of a range is always the same for all queries of the workload

overlaps the query range. If we knew the coefficients $c_i \in path_{q_j}$ for all $q_j \in Q$ then the provided answers would always be exact.

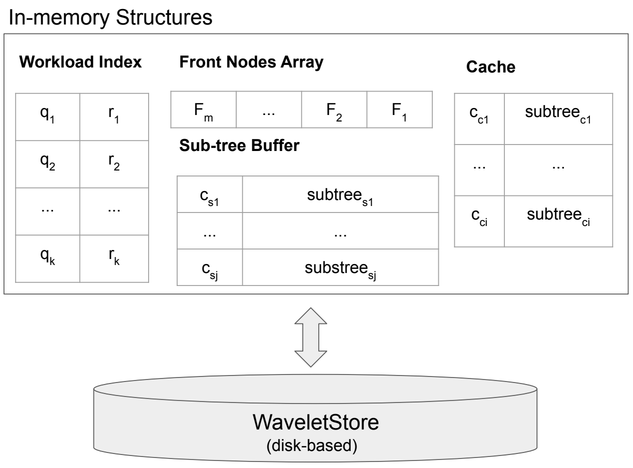


Fig. 7 Architecture of the proposed system for workload-aware range queries in sliding-window streams.

For understanding the extra difficulties the streaming case introduces, we consider the example of Figure 6. Let us assume a budget of $B = 5$ coefficients and a workload $Q = \{q_i\}$. If we keep in the synopsis the whole $path_{q_i}$, then at $t_{now} = t_1$ we will be able to provide an exact answer. Nevertheless, in order to achieve this, we have discarded all the other coefficients in the active window. Hence, since we always care for q_i, in a later time (e.g., $t_{now} = t_1 + 9$), the coefficients of $path_{t_j}$ (annotated with green color) will be of interest but they will have already been deleted.

In order to overcome this obstacle, this work introduces a system design that violates the “one-pass” over the data feature but offers very interesting trade-offs between accuracy and real-time responsiveness. Besides the limited memory, many IoT devices are also equipped with a secondary storage (e.g., SD card[3]) with larger capacity but which is more “expensive” to access. Based on this observation, the system of Figure 7 is proposed. According to the presented design, we do not keep in-memory the whole wavelet synopsis as before, but only the front nodes array and some helping structures that are going to be explained. Moreover, there is an analysis that shows that the helping structures we maintain do not incur a memory overhead larger than $O(logW)$ and thus, the memory constraints still hold.

The main idea of the system is the following: as data items arrive, the Haar wavelet transform is dynamically computed. However, as soon as a new coefficient is created, it is persisted into the *WaveletStore*: a disk-based

[3] https://www.arduino.cc/en/reference/SD

storage device. This way we can retrieve in the future coefficients that have been discarded. Please also note that as *write* operations are performed asynchronously, the ingestion of data into the disk does not delay the construction of the synopsis. In order to answer a query, we perform a lookup in the *Workload Index*. This is a structure that contains the materialized results for the queries of interest. For having fresh data in the Workload Index, we need to continuously update it. An update consists of computing the answer for each query $q_i \in Q$. As usually, the computation of a query q_i consists of two parts: (i) one fully contained in range and (ii) a sub-tree that partially overlaps with it. For the part of the query that is fully contained in the range, we derive the answer by using the *front nodes array*. But for the last sub-tree, we can now retrieve the coefficients $c_j \in path_{q_i}$ from the WaveletStore. A naive solution would require $O\left(\lceil log\frac{W}{logW}\rceil\right)$ *GET* operations, i.e., as many as the coefficients in a maximal path. Nevertheless, such an approach would result in excessive accesses of the considerably more expensive secondary storage and would defeat the purpose of the fast, in-memory approximate query processing.

In the following, we present how we can limit disk accesses and create a fast system that can accurately answer workload-aware range queries under the sliding-window model.

8.1 Disk Access Patterns

The data organization on disk plays a crucial role on the system's performance. In this Section, we discuss how data should be stored and retrieved in order to obtain better response times compared to the naive solution where a logarithmic number of *GET* requests is required for each query. In the following discussion, we assume the WaveletStore to be any disk-based lightweight key-value store.

8.1.1 Path-based Organization

The first approach for limiting the number of issued *GET* requests per query is to store in a single value all the coefficients that have been created at a specific time. Let us denote $P\left(t\right)$ the set of coefficients that are created at t. Thus, at each time t, we persist a key-value of the form:

$$(key, value) = (t, P\left(t\right))$$

As all $|P(t)|$ coefficients must have been created before they are persisted on disk, a buffer of size $|P(t)|$ should exist. According to Section 4, the arrival of two data items can trigger the creation of up to $O\left(\lceil log\frac{W}{logW}\rceil\right)$ new coefficients, and thus the memory overhead of this approach is $O\left(\lceil log\frac{W}{logW}\rceil\right)$.

For the query answering, consider the example of Figure 8. The construction process of the error-tree implies that all coefficients that are surrounded

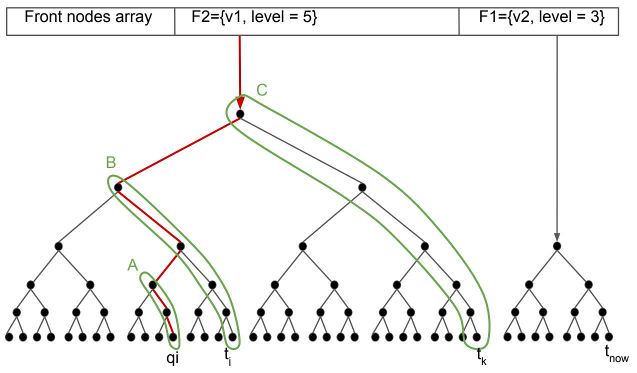


Fig. 8 Example of the path-based data organization.

by curve A have been created at $t_{now} - q_i$, and thus are stored in a single key-value $(t_{now} - q_i, P(t_{now} - q_i))$. Similarly, the coefficients surrounded by B have been created at t_j and by C at t_k. For computing q_i, we need to fetch from disk $P(t_{now} - q_i), P(t_j), P(t_k)$ and filter in memory the coefficients that belong to $path_{q_i}$. This way, for the example of Figure 8, we perform 3 *GET* requests instead of the 5 that the naive approach requires. Lemma 7 places the bounds for the improvements this approach can bring.

Lemma 7 *For reconstructing the exact answer, the path-based organization needs* 1 GET *in the best case and has the same behavior as the naive approach in the worst case.*

Proof. The worst case is observed when the start of the query range is near to the leftmost path of a sub-tree. In that case, each coefficient of the path has been created at a different time and thus, $O\left(\lceil log\frac{W}{logW}\rceil\right)$ *GETs* are required. The best case is observed, if we query the rightmost path of a sub-tree. The rightmost path is wholly created at a single time moment and can be fetched with a single request. □

8.1.2 Subtree-based Organization

Similarly to Section 2, the subtree-based organization partitions the error-tree into sub-trees of fixed size s[4]. For persisting a sub-tree into the WaveletStore, we use as key the index of its root coefficient and as a value the sub-tree itself. Hence, we have key-values of the form:

$$(key, value) = ((r(S).level, r(S).order), S)$$

where S denotes a sub-tree and $r(S)$ its root coefficient. This approach is more time-efficient as it achieves better locality and retrieves more "useful" coefficients with a single *GET* request. However, this is accomplished at the cost

[4] The size of a partition is of the form $s = 2^k - 1, k > 0$

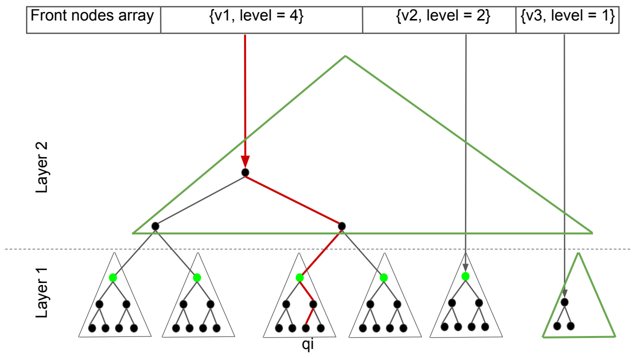


Fig. 9 Example of the subtree-based data organization.

of a higher memory overhead. Figure 9 depicts an example where partitions of size 7 are annotated. We observe that this partitioning scheme divides the error-tree into layers and at any time there can be at most one semi-completed partition at each layer. Partitions that are not yet fully completed should be buffered in memory. Buffering sub-trees until they are flushed to disk is responsibility of the *Sub-tree Buffer* component which is illustrated in Figure 7.

Since we can have at most one semi-completed partition at each sub-tree layer, the memory overhead the *Sub-tree Buffer* incurs is:

$$O(s \cdot \#Layers) = O(s \cdot \lceil log\frac{W}{logW}\rceil / \lfloor log(s)\rfloor)$$

A question that naturally arises is what is a proper value for s. The conducted experiments in Section 9 indicate that the larger the value of s the higher is the memory consumption and the better is the query response time. Moreover, the experiments suggest that the optimal s-value is dependent on the window size W. Setting s equal to a sub-linear function of the window size, such as $logW$, leads to log^2W space complexity and thus, the constraint for poly-logarithmic memory is not violated.

For answering queries under this model, we traverse $path_{q_i}$ in a top-down manner and fetch from disk the sub-trees that intersect with the path. As some sub-trees of interest may reside in the *Sub-tree Buffer* and have not been persisted on disk yet, we also check if the query-path intersects with any of the sub-trees contained in memory.

The improvements on disk-accesses that can be achieved with the subtree-based approach are presented in Lemma 8.

Lemma 8 *The subtree-based organization can reconstruct the exact answer without any disk access in the best case and with at most* $\lceil log(\frac{W}{logW})\rceil / \lfloor log(s)\rfloor$ GET *requests in the worst case.*

Proof. The best case occurs when the query asks for one of the s rightmost paths of the sub-tree. In that case, it can be answered solely based on the

Sub-tree Buffer and no disk access is needed. In the worst case, the query asks for a path of a maximum height sub-tree ($\lceil log\frac{W}{logW}\rceil$) that does not have any overlap with the *Sub-tree Buffer*. Thus, there are $\lceil log(\frac{W}{logW})\rceil/\lfloor log(s)\rfloor$ partitions/sub-trees that need to be fetched from the secondary storage. □

8.2 Maximizing Throughput

Selecting a good data placement in the secondary storage helps improving performance but there are still too many disk accesses that need to be made. The experiments of Section 9 show that even with the subtree-based organization the throughput of maintaining the wavelet structure is 8× lower than the one achieved by the algorithm of Section 4 that works completely in-memory. For speeding-up construction, two key-ideas are used: (i) AQP and (ii) caching.

8.2.1 AQP

So far, the presented algorithms traverse the whole $path_{q_i}$ and compute an exact answer. However, this is too costly as very frequent disk accesses take place. Thus, it is suggested to fetch only the g topmost partitions that intersect with $path_{q_i}$. Retrieving from disk only a part of the path ($g \cdot s$ coefficients) leads to an approximate answer but favors performance. Intuitively, loading the topmost coefficients of a path yields better quality results, since these coefficients contribute to a larger part of the query range. Error guarantees are provided in exactly the same way as described in Section 4.3. The evaluation of Section 9 shows that there are very interesting speed-accuracy trade-offs to explore by experimenting with different g values.

8.2.2 Caching

For further boosting the synopsis' construction throughput, a small cache is also used, as shown in Figure 7. Similarly to *Sub-tree Buffer*, the cache is allowed to contain at most a logarithmic number of sub-trees/partitions.

By observing Figure 9, we notice that each partition that intersects with $path_{q_i}$ for a given query q_i, is going to be present in many consecutive *GET* requests. That means that each partition will be fetched from disk multiple times. Retrieving over and over the same data incurs a significant overhead that we can mitigate with caching. Having available memory space for c partitions, the idea is to cache the c partitions that are requested by the workload and will be "active" for the longest period of time. The time a partition stays active depends on its layer; a partition of size s in layer L has a time coverage of length: $(s + 1)^L$. In the simple example of Figure 9, by considering a cache capable of storing a single partition, we can avoid 63 *GET* requests. For many distributions, retrieving from disk even only the topmost partition for a query can lead to very accurate results. This fact in combination with the caching mechanism create a fast and accurate system where the secondary storage is merely accessed for retrieving data values.

9 Experimental Evaluation

In this Section, the experimental evaluation of the proposed streaming algorithms is presented. Algorithms are compared in terms of accuracy and memory consumption. As accuracy we measure the real observed relative error, i.e.,

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

For the disk-based approach of Section 8, the goal is to explore the speed-accuracy trade-off that the secondary storage incurs. Thus, only for this case, synopsis construction throughput experiments are also considered.

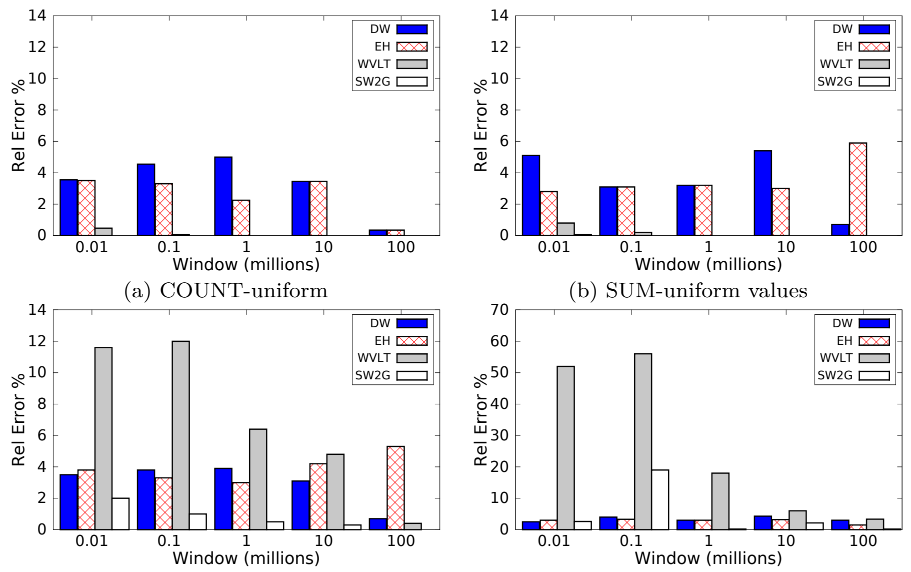
Algorithms. Henceforth, SW2G (Sliding Window Wavelets with Guarantees) denotes the in-memory, approximate algorithm that is presented in Section 4. SW2G is compared to the following techniques: (i) Exponential Histograms (*EH*) [Datar et al. (2002)], (ii) Deterministic Waves (*DW*) [Gibbons and Tirthapura (2002)] and (iii) the classic wavelet structure (*WVLT*) as discussed by Liu et al. (2010) 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 [Datar et al. (2002)] 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 the proposed method of this work are computed while constructing the synopsis and are not theoretical, we demonstrate the results of the proposed approach even for the case of arbitrary data values.

All single-stream algorithms are implemented in Java 8, except for the exponential histograms where the Scala implementation of Algebird (2019) is used. For the distributed algorithms, the Apache Flink 1.6 stream processing framework is employed. The Flink implementation for distributed exponential histograms is based on the Java code of Papapetrou et al. (2012). For the workload-aware case, where a disk-based secondary storage is required, a port of the LevelDB[5] key-value store in Java has been used.

Queries. The considered workloads are mainly range queries (COUNT, SUM, AVG) in the form described in Section 2. 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 the proposed approach, in Section 9.5, aggregates over arbitrary ranges are also considered as well as point queries.

Datasets. For the assessment of the proposed algorithms, both synthetic and real data is used. Synthetic data is used for experimenting with various data distributions. The generated data values lie in the range $[0 - 1000]$

[5] https://github.com/dain/leveldb



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

Fig. 10 Relative error in streams of positive integers (query length = W).

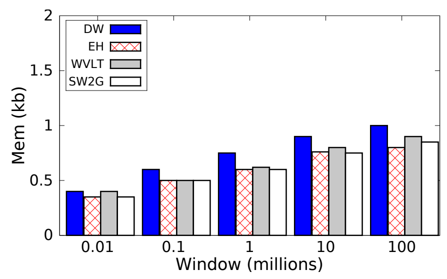
and follow a uniform, normal or highly biased ($s = 2$) zipf distribution. As real data, we use the sensor measurements provided by NOAA (2019). From the various attributes contained in NOAA, the temperature (noaaTemp) and wind-speed (noaaSpeed) time-series are selected. NOAA time-series consist of both 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, a cluster of 4 machines with the same processing and memory capabilities is used.

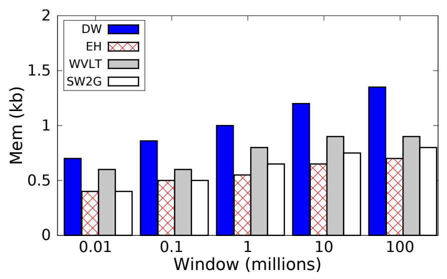
9.1 Positive Integers

In the first experiment, SW2G is evaluated over a single stream of positive integers. As this is the only case where EH and DW can be applied, a direct comparison among the various methods can be performed.

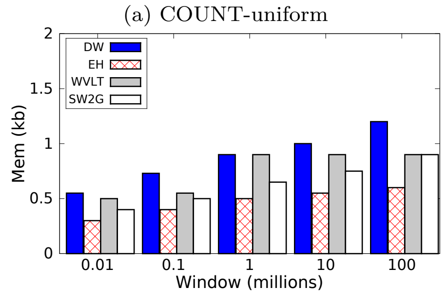
Figure 10 presents accuracy results for various data distributions and window sizes. We consider streams of 400 millions data points and window sizes in the range of $[10k, 100M]$. At random times, we query each structure for the COUNT, SUM or AVG of the stream elements over the last W time units. In the case of the noaaTemp dataset, a more complex query is computed: 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, algorithms are tuned to use approximately the same amount of memory. In



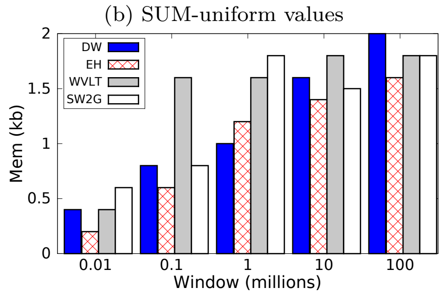
(a) COUNT-uniform



(b) SUM-uniform values



(c) SUM-zipfian values



(d) AVG-noaaTemp

Fig. 11 Memory consumption in streams of positive integers (query length = W).

EH and DW, the tuning knob of memory consumption is the guaranteed error ϵ and for the wavelet-based techniques, the available budget B.

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.

Please recall 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 11 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.

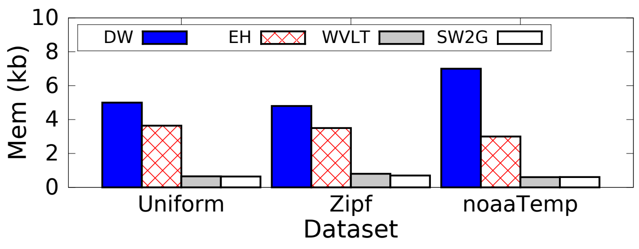


Fig. 12 Memory for $\epsilon = 0.01$

Figure 12 illustrates 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 $10M$.

9.2 Out-of-order Arrivals

Here, we assess the impact that out-of-order tuples have on accuracy. We consider again streams of positive integers and evaluate accuracy for a variety of data-value distributions. Specifically, we track the SUM-statistic and measure the relative error when there is a varying percentage of out-of-order tuples. As in reality late arrivals occur due to network glitches or delays, we do not expect them to be too frequent. In our experiment, we consider three degrees of *lateness*: (i) 1%, (ii) 10% and (iii) 30% of tuples are expected to appear out-of-order.

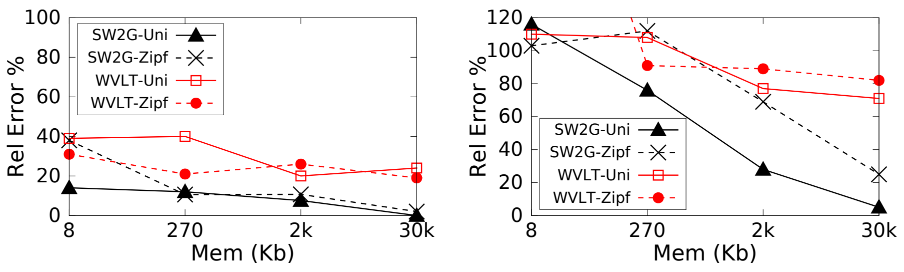
Results are presented in Table 2. As we can see, our structure efficiently handles out-of-order arrivals and the temporal ordering of the stream does not seem to have a significant impact on the quality of results.

Table 2 Relative error(%) for SUM queries with a varying degree of out-of-order tuples

Data-Value Distribution	**Percentage of late tuples**		
	1%	**10%**	**30%**
Uniform	2.5	2.3	2.4
Zipf($s = 2$)	2.1	3.6	3.4
Normal($\mu = 100, \sigma = 20$)	1.6	2.2	2.5

9.3 Streams of Generic Numerical Data

In the case of positive numbers, we demonstrated that the proposed approach outperforms existing techniques. In this Section, the applicability and efficiency of SW2G are examined in more general cases, where the stream also includes negative values. We experiment with SUM queries in real and synthetic data. Uniform and zipf synthetic distributions are used, 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 $= W/8$

Fig. 13 Relative error in streams of arbitrary numerical data.

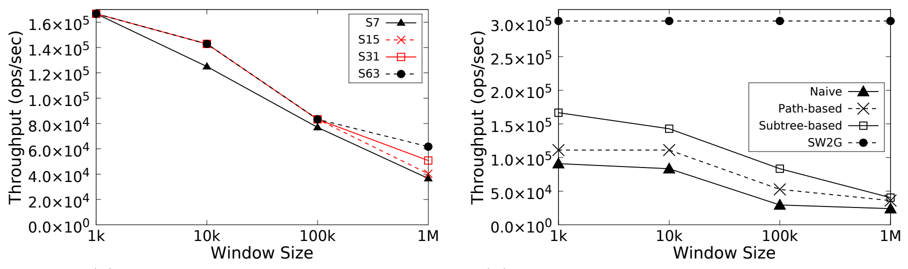
First, queries of length W and $W/8$ are computed on the noaaTemp and noaaSpeed datasets. As the value distribution of the NOAA datasets does not present a great variance, it can be easily approximated by wavelets. As such, both SW2G and WVLT achieved relative errors less than 1% in both workloads.

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

Figure 13 illustrates relative error with respect to the consumed amount of memory. We observe that for both distributions and query lengths, SW2G converges better than WVLT as memory increases. 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.

9.4 Evaluating Workload Aware Synopses

In this Section we evaluate the described system of Section 8 and explore the trade-offs it can achieve between construction throughput and accuracy. For all the experiments of this Section, we consider a workload of a single query, i.e., $Q = \{q_i\}$. The query q_i is randomly selected in the range $[1, W]$. As it is shown, even a single query is enough to showcase the implications of the secondary storage as well as the worst- and best-case performance of the proposed system.



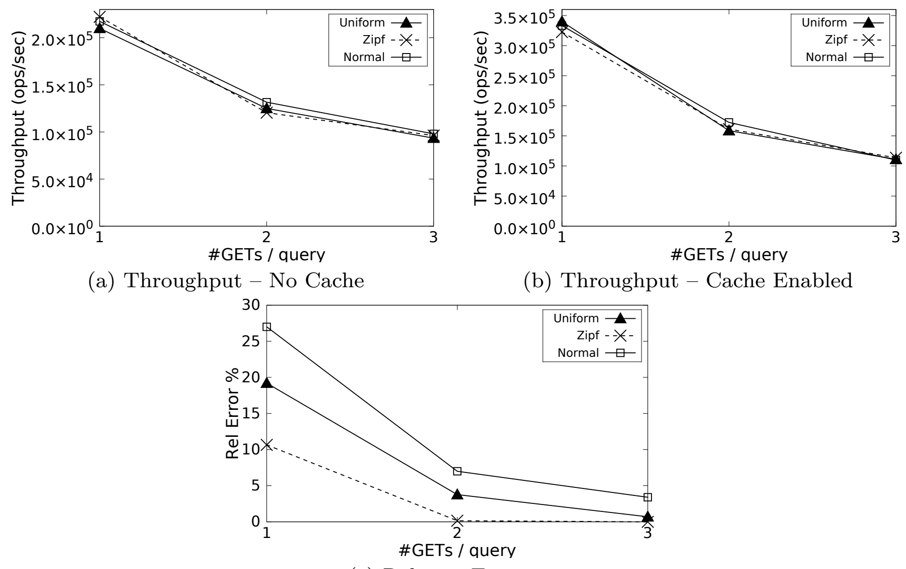
(a) Impact of partition size (b) Compare various disk access patterns

Fig. 14 Experiments on disk placement parameters.

9.4.1 Disk Organization Parameters

First, we assess the proposed methods for organizing data in the secondary storage. As here we want to investigate the pure impact of disk, for the experiments of this Section, we compute an exact answer by fetching the whole $path_{q_i}$ and the caching mechanism is turned off.

Figure 14-(a) illustrates the throughput of constructing a synopsis when the subtree-based organization is used and for various partition sizes. In the Figure, we denote with S_k a partition that contains k wavelet nodes. The results suggest that in general larger partitions achieve better throughput. We observe that the smallest partition S_7 is always outperformed and in the case of a window $W = 1M$, performance results strictly follow the order of partition sizes; the run for the largest partition S_{63} is the fastest one, the run for S_{31} comes second, etc. However, we also notice that performance-wise, the optimal partition size is dependent on the window. For windows smaller than $W = 1M$, further increasing the partition size does not have an impact on throughput. For the remainder of this Section, partitions of 15 wavelet coefficients are considered. Larger partitions may achieve better running-time results but consume more memory. As we want the *Sub-tree Buffer* to be of poly-logarithmic space in the window size, partitions have been selected in a way to achieve a good trade-off between running-time and memory consumption.



(a) Throughput – No Cache (b) Throughput – Cache Enabled

(c) Relative Error

Fig. 15 Impact of # GETs/query on throughput and relative error.

Figure 14-(a) presents a comparative analysis among the various data organizations on disk. The subtree-based organization is 1.5× as fast as the path-based and 2× as fast as the naive one for all evaluated window sizes. Nevertheless, we notice that as the window size increases, throughput drops. When the secondary storage is involved, computing the exact answer for a window $W = 1M$ is 7.5× more expensive, with respect to throughput, than using the in-memory SW2G.

9.4.2 Exploring the Time-Accuracy Trade-off

In Figure 13, we noticed that under some circumstances SW2G does not behave well. More specifically, when the stream contains negative numbers, the budget space is small and the query range is much smaller than W, the relative error increases considerably.

In this Section, it is demonstrated how the proposed system comes to the rescue when workload information exists. Based on the above observations, first, we create adversarial conditions for SW2G and test its running-time and accuracy for various distributions. The results for SW2G are shown in Table 3. Figure15 shows the corresponding results when the proposed workload-aware system is employed.

For all examined distributions, Figure 15-(c) shows that even a single *GET* query to the secondary storage can ensure a relative error lower than 30%; that is a 90% improvement compared to SW2G. When two *GETs* are issued per query, the corresponding error drops to lower than 10%, while three *GETs* provide an almost exact result.

Table 3 Running-time and accuracy performance of SW2G for various distributions

Metric	**Uniform**	**Zipf**	**Normal**
Throughput (ops/sec)	400K	385K	345K
Relative Error %	60	390	159

Figure 15-(a) presents the corresponding throughput results when caches are disabled. The first *GET* request to the disk has a cost of 40% performance degradation compared to SW2G. However, the situation is much better when caches are enabled. In that case, the cost in throughput is less than 15%. Thus, for all distributions, the proposed system can achieve an error lower than 30% with minimal performance overheads.

9.5 General Range and Point Queries

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

Table 4 shows the results for a workload of random AVG queries where the limits of the queried ranges are selected at random. For this experiment, a $200Kb$-sized synopsis is used. Moreover, all datasets contain both positive and negative values.

Table 4 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

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

Figure 16 demonstrates the results for point queries. The applied workload 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 cases. Thus, while optimizing for range queries, the performance of the proposed algorithm in point queries is not compromised. As noticed in Figure 16a, 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.

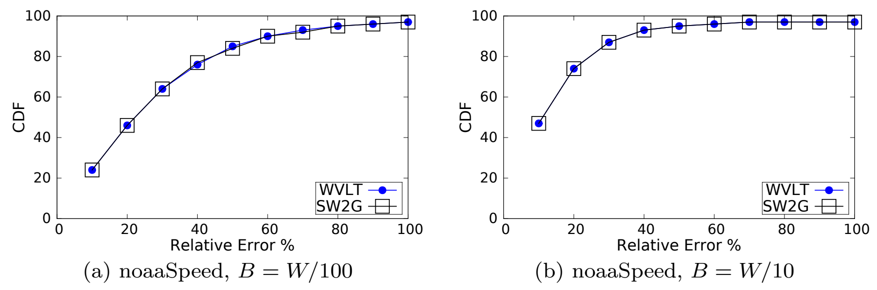


Fig. 16 CDF of relative error in point queries.

9.6 Distributed Streams

This Section examines 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. SW2G is compared with the distributed version of EH which is described in Papapetrou et al. (2012). For distributed exponential histograms we set an error of $\epsilon = 0.1$ both for the coordinator and all remote streams.

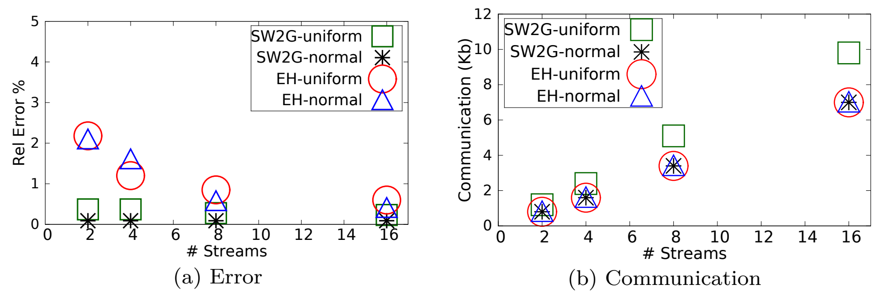


Fig. 17 Relative error and communication cost in distributed streams.

Figure 17 shows the real relative error and the communication cost for synthetic data of uniform and normal distributions. Results are presented 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 Papapetrou et al. (2012) 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.

10 Conclusion

In this paper 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 such as 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 propose a system that can take advantage of secondary-storage and provide interesting trade-offs between accuracy and running-time performance. The experimental evaluation indicates that with a minimal penalty in performance, we can obtain near accurate results for a variety of data distributions and query workloads.

References

Algebird (2019) Abstract algebra for scala. https://twitter.github.io/algebird/

Babcock B, Datar M, Motwani R, O'Callaghan L (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, pp 234–243

Busch C, Tirthapura S (2007) A deterministic algorithm for summarizing asynchronous streams over a sliding window. In: Annual Symposium on Theoretical Aspects of Computer Science, Springer, pp 465–476

Carney D, Çetintemel U, Cherniack M, Convey C, Lee S, Seidman G, Stonebraker M, Tatbul N, Zdonik S (2002) Monitoring streams: a new class of data management applications. In: Proceedings of the 28th international conference on Very Large Data Bases, VLDB Endowment, pp 215–226

Chakrabarti K, Garofalakis M, Rastogi R, Shim K (2001) Approximate query processing using wavelets. The VLDB Journal—The International Journal on Very Large Data Bases 10(2-3):199–223

Chan HL, Lam TW, Lee LK, Ting HF (2012) Continuous monitoring of distributed data streams over a time-based sliding window. Algorithmica 62(3-4):1088–1111

Cohen E, Strauss M (2003) Maintaining time-decaying stream aggregates. In: Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, ACM, pp 223–233

Cormode G, Garofalakis M, Sacharidis D (2006) Fast approximate wavelet tracking on streams. In: International Conference on Extending Database Technology, Springer, pp 4–22

Cortes C, Fisher K, Pregibon D, Rogers A (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, pp 9–17

Datar M, Gionis A, Indyk P, Motwani R (2002) Maintaining stream statistics over sliding windows. SIAM journal on computing 31(6):1794–1813

Ganguly S, Garofalakis M, Rastogi R, Sabnani K (2007) Streaming algorithms for robust, real-time detection of ddos attacks. In: Distributed Computing Systems, 2007. ICDCS'07. 27th International Conference on, IEEE, pp 4–4

Garofalakis M, Gibbons PB (2002) Wavelet synopses with error guarantees. In: Proceedings of the 2002 ACM SIGMOD international conference on Management of data, ACM, pp 476–487

Garofalakis M, Kumar A (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, pp 166–176

Gibbons PB, Tirthapura S (2002) Distributed streams algorithms for sliding windows. In: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures, ACM, pp 63–72

Gilbert AC, Kotidis Y, Muthukrishnan S, Strauss M (2001) Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In: Vldb, vol 1, pp 79–88

Gilbert AC, Kotidis Y, Muthukrishnan S, Strauss MJ (2003) One-pass wavelet decompositions of data streams. IEEE Transactions on Knowledge & Data Engineering (3):541–554

Gilbert AC, Kotidis I, Muthukrishnan S, Strauss MJ (2007) Method and apparatus for using wavelets to produce data summaries. US Patent 7,296,014

Guha S (2005) Space efficiency in synopsis construction algorithms. In: Proceedings of the 31st international conference on Very large data bases, VLDB Endowment, pp 409–420

Guha S, Harb B (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, pp 88–97

Guha S, Harb B (2008) Approximation algorithms for wavelet transform coding of data streams. IEEE Transactions on Information Theory 54(2):811–830

Guha S, Park H, Shim K (2008) Wavelet synopsis for hierarchical range queries with workloads. The VLDB Journal—The International Journal on Very Large Data Bases 17(5):1079–1099

Karras P, Mamoulis N (2005) One-pass wavelet synopses for maximum-error metrics. In: Proceedings of the 31st international conference on Very large data bases, VLDB Endowment, pp 421–432

Karras P, Mamoulis N (2007) The haar+ tree: a refined synopsis data structure. In: Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on, IEEE, pp 436–445

Karras P, Sacharidis D, Mamoulis N (2007) Exploiting duality in summarization with deterministic guarantees. In: Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, pp 380–389

Li T, Li Q, Zhu S, Ogihara M (2002) A survey on wavelet applications in data mining. ACM SIGKDD Explorations Newsletter 4(2):49–68

Liu KH, Teng WG, Chen MS (2010) Dynamic wavelet synopses management over sliding windows in sensor networks. IEEE Transactions on Knowledge and Data Engineering 22(2):193–206

Madden S, Franklin MJ (2002) Fjording the stream: An architecture for queries over streaming sensor data. In: Data Engineering, 2002. Proceedings. 18th International Conference on, IEEE, pp 555–566

Matias Y, Portman L (2003) Workload-based wavelet synopses. Tech. rep., Technical report, Department of Computer Science, Tel Aviv University

Matias Y, Vitter JS, Wang M (1998) Wavelet-based histograms for selectivity estimation. In: ACM SIGMoD Record, ACM, vol 27, pp 448–459

Muthukrishnan S (2005) Subquadratic algorithms for workload-aware haar wavelet synopses. In: International Conference on Foundations of Software Technology and Theoretical Computer Science, Springer, pp 285–296

Mytilinis I, Tsoumakos D, Koziris N (2019) Maintaining wavelet synopses for sliding-window aggregates. In: Proceedings of the 31st International Conference on Scientific and Statistical Database Management, ACM, pp 73–84

NOAA (2019) National oceanic and atmospheric administration. https://www1.ncdc.noaa.gov/pub/data/noaa/

Papapetrou O, Garofalakis M, Deligiannakis A (2012) Sketch-based querying of distributed sliding-window data streams. Proceedings of the VLDB Endowment 5(10):992–1003

Qiao L, Agrawal D, El Abbadi A (2003) Supporting sliding window queries for continuous data streams. In: Scientific and Statistical Database Management, 2003. 15th International Conference on, IEEE, pp 85–94

Rivetti N, Busnel Y, Mostefaoui A (2015) Efficiently summarizing distributed data streams over sliding windows. PhD thesis, LINA-University of Nantes; Centre de Recherche en Économie et Statistique; Inria Rennes Bretagne Atlantique

Shah Z, Mahmood AN, Tari Z, Zomaya AY (2017) A technique for efficient query estimation over distributed data streams. IEEE Transactions on Parallel & Distributed Systems

(10):2770–2783

Stollnitz EJ, DeRose TD, Salesin DH (1996) Wavelets for computer graphics: theory and applications. Morgan Kaufmann

Vitter JS, Wang M (1999) Approximate computation of multidimensional aggregates of sparse data using wavelets. In: Acm Sigmod Record, ACM, vol 28, pp 193–204

Xu B, Tirthapura S, Busch C (2008) Sketching asynchronous data streams over sliding windows. Distributed Computing 20(5):359–374

Yao Y, Gehrke J, et al. (2003) Query processing in sensor networks. In: Cidr, pp 233–244

Zhu Y, Shasha D (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, pp 358–369