One-Pass Wavelet Synopses for Maximum-Error Metrics*

Panagiotis Karras   Nikos Mamoulis

Department of Computer Science
University of Hong Kong
Pokfulam Road
Hong Kong
{pkarras,nikos}@cs.hku.hk

Abstract

We study the problem of computing wavelet-based synopses for massive data sets in static and streaming environments. A compact representation of a data set is obtained after a thresholding process is applied on the coefficients of its wavelet decomposition. Existing polynomial-time thresholding schemes that minimize maximum error metrics are disadvantaged by impracticable time and space complexities and are not applicable in a data stream context. This is a cardinal issue, as the problem at hand in its most practically interesting form involves the time-efficient approximation of huge amounts of data, potentially in a streaming environment. In this paper we fill this gap by developing efficient and practicable wavelet thresholding algorithms for maximum-error metrics, for both a static and a streaming case. Our algorithms achieve near-optimal accuracy and superior runtime performance, as our experiments show, under frugal space requirements in both contexts.

1 Introduction

Several database applications require the reduction of vast amounts of data into a more manageable size. Such data reduction is useful in situations where exactness is not valued as high as speed. For example, in order to evaluate a query execution plan, it is imperative to estimate the selectivity of the query components efficiently, while it is not necessary to get precise knowledge about it. In Decision Support Systems (DSS) applications, a user is not primarily interested in the exact (expensive to retrieve) answer to a query, but in a fairly accurate estimation of it, such that it would reveal the basic features of the examined body of data. Moreover, the need for quick and reliable data approximation is prominent in situations where massive data arrives in a stream; in such settings the approximation needs to be also extracted in a single pass over the data.

Wavelet decomposition [1] provides a very effective data reduction tool, with applications in data mining [12], selectivity estimation [13], and approximate and aggregate query processing of massive relational tables [16, 4] and data streams [7]. In simple terms, a *wavelet synopsis* is extracted by applying the wavelet decomposition on an input collection (considered as a sequence of values) and then summarizing it by retaining only a select 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 wavelet synopses [13, 16, 4].

Wavelet thresholding is the problem of determining the coefficients to be retained in the synopsis given an available space budget B. A conventional approach to this problem features a polynomial-time deterministic thresholding scheme that minimizes the overall mean squared error [15]. Still, the synopses produced by this method have some significant drawbacks [6], such as the high variance in the quality of data approximation, the tendency for severe bias in favor of certain regions of the data and the lack of comprehensible error guarantees for individual approximate answers. On the other hand, synopses that minimize *maximum* error metrics on individual data values prove more robust in accurate data reconstruction [5, 6].

Garofalakis and Kumar [6] have proposed optimal, PTIME *deterministic* thresholding algorithms for minimizing absolute or relative maximum error metrics in the data approximation, based on dynamic programming. These solutions improve upon an earlier *probabilistic* coefficient thresholding scheme [5]. In spite of this progress, the cardinal questions of time-performance and space-efficiency in wavelet thresholding algorithms have not received the at-

* Work sponsored by grants HKU 7380/02E and HKU 7149/03E from Hong Kong RGC.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

**Proceedings of the 31st VLDB Conference,
Trondheim, Norway, 2005**

tention that they deserve to-date. Symptomatically, while the latest contribution [6] provides comprehensible error guarantees for individual approximate answers, it imposes large time and space requirements on the algorithm. Such requirements render the solution impracticable for the purpose it is meant to be for, namely the quick and space-efficient summarization of data into manageable general-purpose synopses [4].

In this paper we address the ensuing need for efficient near-linear deterministic algorithms, which extract accurate enough wavelet synopses in practicable time and space. We first propose efficient greedy thresholding algorithms that achieve near-optimal results in terms of maximum-error metrics and far superior performance in terms of running time and space requirements. Then, we effectively extend these techniques to the problem of compressing streaming data. Such data (i) are not immediately available and may not be accessed at anytime, (ii) they are seen only once, and (iii) they may have fast arrival rate, calling for efficient computation techniques. Previous work on extracting synopses from streaming data [7, 9] are also based on the conventional mean squared error minimization, which as discussed is inadequate. On the other hand, our proposed one-pass thresholding methods are suited for max-error metrics. As demonstrated by experiments with real and synthetic data, they achieve competitive results to their static data counterparts.

The remainder of the paper is structured as follows. Section 2 provides some background and presents the fundamental ideas behind the wavelet thresholding methodology and the shortcomings of existing approaches. In section 3 we introduce our new wavelet thresholding algorithms. These are extended to one-pass versions for streaming data in section 4. Experimental results on real-world and synthetic data sets are outlined in Section 5. Section 6 presents some further discussion and potential future research directions. Finally, in Section 7 we outline our conclusions.

2 Background and Related Work

Wavelet analysis is a major mathematical technique that facilitates effective hierarchical decompositions of functions [8, 1, 3]. Functions are represented by means of a coarse overall shape, in addition to progressively narrower hierarchical levels of detail that influence it at various scales. Thus, wavelets can successfully approximate sharp discontinuities. In this section, we provide the basic background to the wavelet thresholding method and discuss the relation of our work to past research.

2.1 Haar Wavelets

Haar wavelets constitute conceptually the simplest possible orthogonal wavelet system. The *Haar wavelet decomposition* of an one-dimensional data vector consists of a coefficient representing the overall average of the data values followed by *detail* coefficients in the order of increasing resolution. Each detail coefficient is the difference of (the second of) a pair of averaged values from the computed pairwise average. Accordingly, a vector $\vec{d} = \{d_0, \ldots, d_7\}$ of 8 data values can be represented by 4 average values and 4 detail coefficients, from which the original data can be easily reconstructed. After applying the same process recursively, we end up with the full wavelet decomposition, made of a single overall average value followed by three hierarchical levels of 1, 2, and 4 detail coefficients respectively, in order of increasing resolution. The original vector $\vec{d}$ of 8 data values can be fully reconstructed from the vector $W_{\vec{d}}$ of these 8 wavelet coefficients.

Error Trees The *error tree*, introduced in [13], is a hierarchical structure that illustrates the key properties of the Haar wavelet decomposition. Each internal node $c_i (i = 0, \ldots, 7)$ of this tree is associated with a wavelet coefficient value, while each leaf node $d_i (i = 0, \ldots, 7)$ is associated with a data value in the original array. Figure 1 shows the Haar wavelet decomposition in the form of an error tree for the data sequence $\vec{d} = \{11, -1, -6, 8, -2, 6, 6, 10\}$. Given an error tree T and an internal node c_k of T, we let leaves_k denote the set of data nodes in the the subtree rooted at c_k. This notation is extended to $\mathsf{leftleaves}_k$ ($\mathsf{rightleaves}_k$) for the left (right) subtrees of c_k. We let path_k be the set of all nodes with nonzero coefficients in T which lie on the path from a node c_k (or d_k) to the root of T. Finally, for any two data nodes d_l and d_h, we use $d(l : h)$ to denote the range sum $\sum_{i=l}^{h} d_i$.

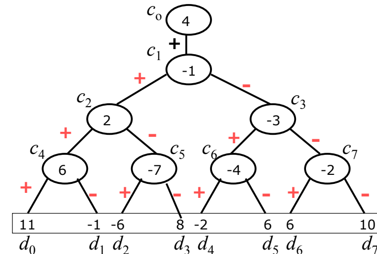


Figure 1: An error tree

Given the error tree representation T of a one-dimensional Haar wavelet decomposition, we can reconstruct any data value d_i using only the nodes that lie on the path to d_i. In other words, $d_i = \sum_{c_j \in \mathsf{path}_i} \delta_{ij} \cdot c_j$, where the (sign) factor $\delta_{ij} = +1$ if $d_i \in \mathsf{leftleaves}_j$ or $j = 0$. $\delta_{ij} = -1$, otherwise. For example, in Figure 1, value d_2 can be reconstructed by following the path from the root to the third leaf and adding/subtracting the visited coefficients, i.e., $d_2 = +4 + (-1) - (2) + (-7) = -6$. A range sum $d(l\!:\!h)$ can be computed using only nodes $c_j \in \mathsf{path}_l \cup \mathsf{path}_h$, by $d(l\!:\!h) = \sum_{c_j \in \mathsf{path}_l \cup \mathsf{path}_h} x_j$, where $x_j = (h-l+1) \cdot c_j$, if $j = 0$ and $x_j = (|\mathsf{leftleaves}_{j,l:h}| - |\mathsf{rightleaves}_{j,l:h}|) \cdot c_j$, otherwise. Here $\mathsf{leftleaves}_{j,l:h} = \mathsf{leftleaves}_j \cap \{d_l, d_{l+1}, \ldots, d_h\}$ and $\mathsf{rightleaves}_{j,l:h} = \mathsf{rightleaves}_j \cap \{d_l, d_{l+1}, \ldots, d_h\}$. In other words, node c_j contributes to the range sum $d(l : h)$ positively as many times as there are leaf nodes of the left subtree of c_j in the summation range, and negatively as many times as there are leaf nodes of the right subtree of c_j, while the value of c_0 contributes positively for

each leaf node in the summation range. In our example, $d(2:7) = 6\times4+(2-4)\times(-1)+(0-2)\times2 = 22$. Hence, if N is the size of the original data, the reconstruction of a single data value involves a summation of $\log N + 1$ coefficients, while the reconstruction of a range sum involves a summation of at most $2\log N + 1$ coefficients.

2.2 Wavelet-based Data Reduction

The complete Haar wavelet decomposition $W_{\vec{d}}$ of a data sequence $\vec{d}$ is a representation of equal size as the original array. Given a 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 omitted coefficients are implicitly set to zero. The resulting *wavelet synopsis* $\hat{W}_{\vec{d}}$ can be used as a compressed approximate representation of the original data.

As an example, consider the decomposition of Figure 1 and assume that only coefficients $\{c_0, c_5, c_6\}$ are retained in the synopsis, whereas the rest of them are implicitly set to 0. The reconstructed value $\hat{d}_5$ for d_5 is $\hat{d}_5 = 4 - (-4) = 8$, whereas the actual value $d_5 = 6$. Popular aggregate error measures for assessing the quality of a wavelet synopsis include the *mean squared error* (L_2), the *maximum absolute error* (max_abs), and the *maximum relative error* (max_rel):

$$L_2(\hat{W}_{\vec{d}}, W_{\vec{d}}) = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(\hat{d}_i - d_i)^2} \quad (1)$$

$$max_abs(\hat{W}_{\vec{d}}, W_{\vec{d}}) = \max_{i=1}^{N}\{|\hat{d}_i - d_i|\} \quad (2)$$

$$max_rel(\hat{W}_{\vec{d}}, W_{\vec{d}}) = \max_{i=1}^{N}\left\{\frac{|\hat{d}_i - d_i|}{\max\{|d_i|, S\}}\right\} \quad (3)$$

In the above equations, $\hat{d}_i$ denotes the reconstructed value for d_i and S is a *sanity bound* used to prevent the influence of very small values in the aggregate error.

2.2.1 The Conventional Approach

A preliminary approach to the thresholding problem is based on two basic observations about a coefficient's contribution in the reconstruction of the original data values (and range-sums). First, coefficients of larger values are more important, since their absence causes larger absolute error in reconstructed values. Second, a coefficient's significance is larger if its level in the error tree is higher, as it then participates in more reconstruction paths of the error tree. Putting both together, the significance c_i^* of a coefficient is defined by $c_i^* = |c_i|/\sqrt{2^{\text{level}(c_i)}}$, where $\text{level}(c_i)$ denotes the *level of resolution* at which the coefficient resides (0 corresponds to the "coarsest" resolution level).

Accordingly, a conventional thresholding approach is to retain the B wavelet coefficients with the greatest significance. It has been shown [15] that this scheme minimizes the L_2 error. Nevertheless, the L_2 error minimization does not provide maximum error guarantees for individual approximate answers. As a result, the approximation error of individual values can be arbitrarily large, resulting into high variance in the quality of data approximation and severe bias in favor of certain regions of the data. This problem is particularly striking whenever a series of omitted coefficients lies along the same path of the error tree. max_abs and max_rel prove more robust error measures [5, 6], since they set a maximum error guarantee on individual values.

Matias et al. [13] proposed *greedy* thresholding algorithms, which, however, are based on an initial selection of *large* coefficients that introduces bias towards L_2-minimization. In this paper, we also propose greedy algorithms; however, our methods are suited for *maximum error* metrics. In addition, we study effective adaptations of them for *streaming* data sequences.

2.2.2 Optimal Max-Error Thresholding

Garofalakis and Kumar [6] proposed a deterministic wavelet thresholding algorithm, based on dynamic programming, that succeeds in minimizing the maximum absolute or relative error in the data approximation. This approach offers a way to alleviate the setbacks of the conventional method. In addition, it improves upon an earlier probabilistic scheme [5], which is flawed, according to [10], since it is based on a questionable application of *probabilistic expectation*.

Unfortunately, the optimal solution [6] has a high complexity of $O(N^2 B \log B)$ in time and $O(N^2 B)$ in total space, where N is the total number of coefficients and B the number of retained coefficients. This burden is too high a price to pay for the minimization of maximum-error metrics. Given that the problem at hand involves the quick and space-efficient summarization of vast amounts of data into manageable general-purpose synopses, it is clear that the complexity handicap renders the proposed techniques impracticable for their intended application. Thus, no computationally efficient technique has been proposed to-date that can target these error metrics under tight time and space constraints, and certainly not under the time and space limits associated with streaming data.

2.2.3 Wavelet Thresholding over Streams

[7] studied the construction of L_2-minimal wavelet synopses for aggregate data values (e.g. frequency counters) computed from a data stream and [9] extended this approach for data with multiple measures. Meanwhile, [2] proposed a similar incremental approach for hierarchical stream summarization which focuses on simple queries and communication caching issues for wavelet coefficients, while [14] developed an incremental algorithm for auto-

matic sensor stream mining, using a wavelet representation of the series and capturing correlations by applying linear regression in the wavelet domain. These approaches do not involve a sophisticated on-the-fly thresholding of coefficients. After all, obtaining a wavelet synopsis of minimal L_2 error from a data stream sequence is straightforward, as it corresponds to an on-the-fly selection of the B most significant coefficients. On the other hand, obtaining an one-pass data stream wavelet synopsis that minimizes max_abs or max_rel is not trivial. In this paper we deal with this stream compression problem, without aggregation of streaming information involved. To our knowledge, there is no previous work on this important problem.

3 Greedy Max-Error Thresholding

Given the impracticable cost of minimizing individual maximum-error metrics in the synopsis, the question arises whether we can design algorithms providing reliably low individual error guarantees, of no practical disadvantage in comparison to the optimal solution, while obtaining a major practical advantage in terms of time and space efficiency. In this section we devise such techniques, based on a *greedy* approach, that extract synopses of near-optimal individual maximum error metrics in near-linear time and space. We first focus on wavelet thresholding for static data. Section 4 extends our solutions for streaming data.

3.1 Maximum Absolute Error

We first design a practicable greedy algorithm for the wavelet thresholding problem aiming at a minimum possible absolute reconstruction error of individual values (see Equation 2). The key idea behind this GreedyAbs algorithm is to first compute the wavelet decomposition $W_{\vec{d}}$ and then select the coefficients to discard *one at a time*. At each step, the coefficient that causes the least maximum absolute error on the *running* synopsis if discarded is removed. This process is repeated until B coefficients have been left in the synopsis.

Let $\mathsf{err}_j = \hat{d}_j - d_j$ be the signed accumulated error for a data node d_j in a synopsis $\hat{W}_{\vec{d}}$, yielded by the deletions of some coefficients. To assist the iterative step of the greedy algorithm, for each coefficient c_k, not yet discarded, we introduce the *maximum potential absolute error* MA_k that c_k will effect on the running synopsis, if discarded:

$$\mathsf{MA}_k = \max_{d_j \in \mathsf{leaves}_k} \{|\mathsf{err}_j - \delta_{jk} \cdot c_k|\} \quad (4)$$

The computation of MA_k normally requires information about all err_j in leaves_k. A naive method to compute MA_k is to access all leaves_k, where err_j are explicitly maintained. The disadvantages of this approach are the explicit maintenance of err_j's at each step and the aggregation cost required to update MA_k's after the removal of a coefficient.

We can accelerate the computations and updates of MA_k by exploiting the fact that the removal of a coefficient affects *equally* the signed costs of all data values in its left or right subtree. For example, in Figure 1, the removal of coefficient $c_2 = 2$ decreases the signed errors of data nodes d_0, d_1, and increases the signed errors of d_2, d_3, by 2. Accordingly, the *maximum* and *minimum* signed errors in the left (right) subtree of a removed coefficient c_i are decreased (increased) by c_i. The maximum absolute error incurred by the removal necessarily occurs at one of these four positions of existing error extremum. Hence the computation of MA_k requires that only four quantities be maintained at each internal node of the tree: $\max_{d_j \in \mathsf{leftleaves}_k} \{\mathsf{err}_j\}$, $\min_{d_j \in \mathsf{leftleaves}_k} \{\mathsf{err}_j\}$, $\max_{d_j \in \mathsf{rightleaves}_k} \{\mathsf{err}_j\}$, and $\min_{d_j \in \mathsf{rightleaves}_k} \{\mathsf{err}_j\}$, for simplicity denoted as max^l_k, min^l_k, max^r_k, and min^r_k, respectively. It follows that Equation 4 is equivalent to:

$$\mathsf{MA}_k = \max\{ \; |max^l_k - c_k|, |min^l_k - c_k|, |max^r_k + c_k|, |min^r_k + c_k| \; \} \quad (5)$$

In the complete wavelet decomposition, these four quantities are all 0, since $err_j = 0, \forall d_j$. Thus, $\mathsf{MA}_k = |c_k|, \forall k$ and our greedy algorithm removes the smallest $|c_k|$ first. In general, in order to efficiently decide which coefficient to choose next, all coefficients are organized in a heap structure H based on their MA_k. After the removal of a coefficient c_k, err_j for all leaves_k changes, so we must update the information of all descendants and ancestors of c_k. All the error quantities of the descendants in the left (right) sub-tree of c_k are decreased (increased) by c_k. During this process, a new MA_i is computed for each descendant c_i of c_k. In accordance, the changes in error quantities are propagated upwards to ancestors c_i of c_k and MA_i's are updated as necessary. While updating error quantities and MA's, the positions of c_k's descendants and affected ancestors are dynamically updated in H. Figure 2 summarizes the described GreedyAbs algorithm.

Note that by maintaining the four error quantities at each non-deleted node, we only need space to store these nodes. In other words, we do not have to explicitly update err_j for each data node d_j. In addition, the data nodes and deleted coefficients do not have to be physically stored in memory. In section 4 we will see that this approach is very convenient in a streaming environment, where the available memory has to be dynamically allocated only to remaining coefficients and we have no luxury of extra space. Alternatively, we may choose to store only two quantities $\max_{d_j \in \mathsf{leaves}_k} \{\mathsf{err}_j\}$ and $\min_{d_j \in \mathsf{leaves}_k} \{\mathsf{err}_j\}$ at each coefficient, and compute max^l_k, min^l_k (max^r_k, min^r_k) from the left (right) child of c_k.

Another important thing to note is that max_abs (and max_rel) does not change monotonically when a coefficient is removed. In other words, after deleting a coefficient c_k the maximum absolute error of its affected data values may *decrease*. As a result, choosing exactly B coefficients may not be the best solution given a space budget B. A

more effective variant of GreedyAbs is to keep removing coefficients also after the limit of B has been reached, until no coefficient remains in the tree. From all $B+1$ coefficient sets (including the empty set) produced at the last B steps of the algorithm, we keep the one with the minimum max_abs.

```
Algorithm GreedyAbs(W_d, B)
Input: Set W_d = [c_0, ..., c_{N-1}] of N Haar wavelet coefficients
Output: Set Ŵ_d ⊂ W_d of B Haar wavelet coefficients
1.  H := create_heap(W_d); //create heap with all c_k ∈ W_d
2.  while (more than B coefficients in H)
3.      discard c_k := H.top; //coefficient with smallest MA_k
4.      max^l_k := max^l_k − c_k; min^l_k := min^l_k − c_k;
5.      max^r_k := max^r_k + c_k; min^r_k := min^r_k + c_k;
6.      for each node c_i in left sub-tree of c_k
7.          decrease all error measures in c_i by c_k;
8.          if c_i not discarded
9.              recalculate MA_i; update c_i's position in H;
10.     for each node c_i in right sub-tree of c_k
11.         increase all error measures in c_i by c_k
12.         if c_i not discarded
13.             recalculate MA_i; update c_i's position in H;
14.     max_err := max(max^l_k, max^r_k);
15.     min_err := min(min^l_k, min^r_k);
16.     c_i = c_k.parent;
17.     while (c_i ≠ NULL)
18.         if (c_k in the left subtree of c_i) then
19.             max^l_i := max_err; min^l_i := min_err;
20.         else // c_k in the right subtree of c_i then
21.             max^r_i := max_err; min^r_i := min_err;
22.         if any of {max^l_i, min^l_i, max^r_i, min^r_i} changed then
23.             if c_i not discarded
24.                 recalculate MA_i; update c_i's position in H;
25.             max_err := max(max^l_i, max^l_i);
26.             min_err := min(min^l_i, min^r_i);
27.             c_i := c_i.parent;
28.         else break for-loop; //no other ancestor can change
```

Figure 2: Greedy thresholding for maximum absolute error

Complexity Analysis The initial heap H can be constructed in $O(N)$ time. Assuming that $B \ll N$, the algorithm performs $O(N)$ coefficient discarding operations. A dropped coefficient c_k at height h of the error-tree has at most 2^h non-deleted descendant coefficients which must be updated and there are $2^{\log N - h}$ coefficients at height h. Thus, the total number of updates in descendants for all deleted coefficients is $\sum_{h=1}^{\log N} 2^h \times 2^{\log N - h} = \sum_{h=1}^{\log N} N = O(N \log N)$. A dropped coefficient c_k has at most $\log N$ non-deleted ancestors, thus the total number of updates in ancestors for all deleted coefficients is also $O(N \log N)$. Each update in a descendant or ancestor costs its re-positioning in H, which is a $O(\log N)$ operation. Summing up, the overall worst-case cost of GreedyAbs is $O(N \log^2 N)$. As we will see in the experimental section, the algorithm's complexity is linear, in practice. The space complexity is $O(N)$, since constant information is kept for every node of the error tree.

3.2 Maximum Relative Error

Minimizing the maximum relative error is arguably more essential compared to absolute error minimization in approximate query processing, as the same absolute error in two different data values may express huge differences in relative error. At the same time, relative error measures tend to be inordinately dominated by small data values. For instance, returning 2 as the approximate answer for 1 amounts to an 100% relative error, while in fact it is insignificant in a data context dominated by much larger values. In order to overcome such problems, several techniques have been developed for combining absolute and relative error metrics [11, 16]. As in earlier approaches [5, 6], we have opted for the relative error metric with a sanity-bound $S > 0$. Our aim is to produce wavelet synopses in near-linear time and space such that, for each approximation $\hat{d}_i$ of a data value d_i, the ratio $\frac{|\hat{d}_i - d_i|}{\max(|d_i|, S)}$ is kept lower than a feasible bound.

Accordingly, we can follow the greedy paradigm introduced in the previous section, wherein, instead of using MA_k, we choose to discard the coefficient with the minimum *maximum potential relative error*, defined as follows:

$$\mathsf{MR}_k = \max_{d_j \in \mathsf{leaves}_k} \left\{ \frac{|\mathsf{err}_j - \delta_{jk} \cdot c_k|}{\max(|d_j|, S)} \right\} \qquad (6)$$

Nevertheless, we can not simply use the four error quantities described in Section 3.1 in order to calculate or update MR_k. The reason is the denominator in 6, which implies that the effect of removing a coefficient c_k is different in the signed relative error of different data values. Thus, information about the data leaves with the currently maximum/minimum signed accumulated error in the left/right subtrees of c_k is useless, as the leaf that determines MR_k does not necessarily belong to this quadruplet.

Thus, in order to apply greedy thresholding to the case at hand, we need to explicitly maintain with each coefficient c_i the potential relative error $\mathsf{pot_err}_{ij}$ caused by its removal on each data value d_j of the subtree rooted at c_i. We can then directly compute MR_i using Equation 6. The cost of computing MR_i is not constant (as in the absolute error case), but depends on the height of c_i; in the worst case it is $O(N)$ (i.e., if c_i is the root). In order to reduce this cost, for each coefficient c_k we organize the data values that c_k affects in an augmented heap H_k on c_k, such that the data value that causes the maximum potential error after the removal of c_k can be accessed in constant time. Therewith the increase of cost in comparison to the GreedyAbs algorithm is balanced to a $\log N$ factor in both time and space. After removing a node c_k, MR_i is updated for every node c_i (ancestor or descendant of c_k). This requires an update in each H_i of the potential relative error for those data descendants of c_i which are also in the subtree of c_k. Figure 3 shows a pseudo-code for the GreedyRel algorithm.

Complexity Analysis The space requirements of GreedyRel are $O(N \log N)$, due to the the extra space necessitated for storing the descendants' heaps. The time complexity is as follows. A dropped coefficient c_k at height h of the error-tree has at most 2^h leaves, the error in

Algorithm GreedyRel($W_{\vec{d}}, B$)
Input: Set $W_{\vec{d}} = [c_0, \ldots, c_{N-1}]$ of N Haar wavelet coefficients
Output: Set $W_{\vec{d}} \subset W_{\vec{d}}$ of B Haar wavelet coefficients
1. **for each** $c_k \in W_{\vec{d}}$ H_k := create_heap(leaves$_k$);
2. H := create_heap($W_{\vec{d}}$); //create heap with all $c_k \in W_{\vec{d}}$
3. **while** (more than B coefficients are left)
3. **discard** $c_k := H.top$; //coefficient with smallest MR_k
4. **propagate_error**(leftleaves$_k$,$-c_k$);
5. **propagate_error**(rightleaves$_k$,c_k);
6. **for each** *affected leaf* d_j **do**
7. **for each** non-discarded *ancestor* c_i of d_j **do**
8. $\text{pot_err}_{ij} := \frac{|\text{acc_error}_j - \text{sign}_{ij} \cdot c_i|}{\max(|d_j|, S)}$;
9. update d_j's position in heap H_i;
10. **for each** ancestor/descendant c_i of c_k **do**
11. update c_i's position in heap H;

Figure 3: Greedy thresholding for maximum relative error

which must be updated and there are $2^{\log N - h}$ coefficients at height h. Thus, the total number of updates in leaves $\sum_{h=1}^{\log N} 2^h \times 2^{\log N - h} = O(N \log N)$. Each update in a leaf is propagated to all its $O(\log N)$ ancestors and in each such ancestor an update in H_i costs $O(\log N)$. Thus, the overall cost for updates in leaves and local heaps of each coefficient is $O(N \log^3 N)$ (lines 4–9). Finally, after each removal, each of the affected nodes updates its position in the global heap H at $O(\log N)$ time (lines 10–11). Summing up, the overall cost of GreedyRel is $O(N \log^3 N)$.

4 Extension to Data Streams

The case of streaming data is a major application area of the suggested synopses. However, the heretofore presented algorithms are not directly applicable on data streams, where the totality of the data is not accessible at once and the space is constrained. In such a context, we can assume that the memory budget B sets not only an upper limit on the size of the final synopsis, but also delimits the memory available for intermediate information storage. From an asymptotic viewpoint, we can accordingly introduce an $O(B)$ bound to the required memory.

In order to adapt our techniques into such a streaming environment, we have to construct the wavelet transform and greedily truncate it in an integrated one-pass process of $O(B)$ space complexity, while preserving the error tree structure as well as information about the incurred errors. In the following description we assume that a minimization of max_abs is aimed. Then, in Section 4.5, we discuss the differences required to adapt the same algorithm for max_rel minimization.

4.1 Algorithm overview

The wavelet decomposition and error-tree for the first B data items is constructed straightforwardly. Thereafter, a pair of coefficients is discarded for every arriving data pair. This pair is selected greedily as in the static case, so as to incur the least possible error, with the scope now limited by necessity to the hitherto constructed part of the error tree.

During this process, as each pair of two new data values is read, their error-tree ancestor(s) are constructed as necessary. Therefrom a higher level of the error tree is created for every higher power of two of data that arrives. While the number of stream data that have arrived is unequal to a power of two, the error tree includes unconnected, *hanging* coefficient nodes. In order to accommodate such nodes, we use an auxiliary data structure (*frontline*), which consists of one node for each level of the error tree ($\lfloor \log N \rfloor$ nodes while N stream data have arrived). We use 'fnode' to refer to a node in this structure, in order not to be confused with a node of the error-tree. If all coefficients are kept, there can be at most one hanging node in each error-tree level, as two created nodes on the same level suffice for the creation of their parent on the next level. The address of this single hanging node is stored in the fnode at the same level.

The frontline structure also stores the average of the data values in the subtree rooted at a hanging coefficient. For instance, assume that a data stream generates the series of numerical data {9, 3, 9, -5, 5, 13, 13, 17, 14, -2, 9, 7, 7, 3, . . . }. The error-tree construction process applied on this series will produce the tree shown on Figure 4.

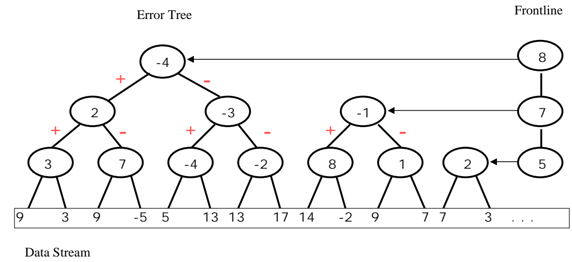


Figure 4: Error tree construction from a data stream

A high-level description of the streaming algorithm is shown in Figure 5.

Algorithm StreamGreedy(S, B)
Input: Data Stream S of raw data
Output: A synopsis W_S of S under budget B
1. N := 0; //current number of data
2. **for** the first B data in S
3. N := N+2; read next two data (d_1, d_2);
4. climbup(N, d_1, d_2)
5. **while** (stream active)
6. N := N+2; read next two data (d_1, d_2);
7. climbup(N, d_1, d_2)
8. discard two coefficients from existing error-tree
9. perform *padding* on hanging edges of error-tree
10. return root node of error-tree
end

Figure 5: Thresholding on a data stream

As we discard coefficients from the error tree, the parent-child relationships between nodes are reconfigured, thus a node may now have several different direct descendants at several different levels, if some immediate child has been deleted. For this purpose, we represent the error tree as a *sibling* tree. For instance, a view of the full error tree of Figure 4 in its sibling tree form is shown in Figure 6.

The sparse error tree constructed during this process, can then be used to produce a reconstruction of an approx-

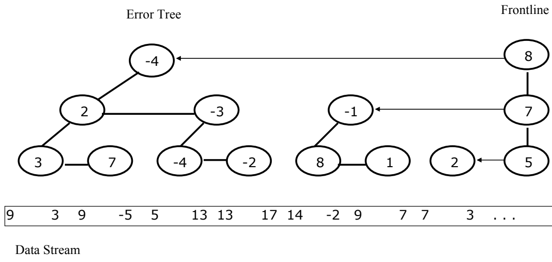


Figure 6: Sibling tree representation

variable	meaning
v_k	coefficient value
max_k^l, min_k^l, etc.	error quantities
par_k	ptr to c_k's parent
$child_k$	ptr to c_k's leftmost child
$next_k$	ptr to c_k's next sibling
$prev_k$	ptr to c_k's previous sibling
$front_k$	ptr to fnode where c_k is hanged
l_k	level of c_k in error-tree
o_k	order of c_k in error-tree level l_k
$index_k$	index of c_k in full error-tree

Table 1: Information in an error-tree node c_k

imation for any data in the streamed data set, or of the complete streamed data set as a whole. For that purpose we have also developed a recursive reconstruction procedure for sparse error-trees. Finally, our algorithm uses a coefficient-padding procedure that fills in the gaps in the constructed error-tree when the acquired data set has a size unequal to a power of two.

4.2 Auxiliary Information

In order to facilitate the operations of our streaming algorithm each node c_k in the error tree stores the information (of constant size) shown in Table 1. v_k is the differential value of the coefficient in node c_k in the error tree.[1] The *data error information* corresponds (for maximum absolute error) to the four quantities (max_k^l, min_k^l, etc.) introduced in Section 3.1. Pointers par_k, $child_k$, $next_k$, $prev_k$, and $front_k$ are used for easy navigation in the error-tree and transition to the frontline structure.

The two integer values l_k, o_k are needed to calculate the *index* of the stored coefficient in the final decomposition array, and hence in the final synopsis. Since the length of the stream is unknown in advance, the index can only be computed after all the stream data has been read, using the level l_k in which the node resides, counting from the bottom, and the order o_k in which it appears in this level. After the stream has been read, the total number of error tree levels L is known. Then the *index* of a coefficient c_k which has been preserved in the ensuing synopsis can be calculated as $index_k(= k) = 2^{L-l_k} + o_k - 1$.

Besides, l_k and o_k are used to calculate a node's relationship (left or right child) to its parent. In contrast to the static case, where discarded nodes were not permanently withdrawn from the error-tree structure, in the streaming case this relationship is dynamic, since the parent of a node may have been physically deleted. If for a node c_k, o_k mod 2 is 0, we know that c_k is a right child of its parent (and the left child, otherwise). In general, if c_i is an ancestor of c_j, we know that c_j is in the left subtree of c_i if $2 \cdot o_j - 1 < (2 \cdot o_i - 1) \cdot 2^{l_i - l_j}$; otherwise c_j is the right subtree of c_i. The computation of this relationship is encountered in several operations.

[1] In the previous section, we used c_k to denote both the coefficient and its value; here we use v_k to explicitly refer to its value.

variable	meaning
$f.ptr$	ptr to error-tree node hanging from f
$f.v$	mean data value in subtree hanging from f
$f.max$, $f.min$	error quantities from deleted orphans
$f.prev$	ptr to previous (lower) fnode
$f.next$	ptr to next (upper) fnode
$f.l$	level of f

Table 2: Information in a frontline node f

Information is also stored in the fnodes, in order to preserve the error-tree structure and accommodate for the loss of information incurred by deleting error-tree nodes. Table 2 summarizes this information. The first three items in the table are used when a new error-tree node is created in the level above f: the hanging node $f.ptr$ is attributed as child to the newly created node; the mean value $f.v$ in the subtree hanging from f is used to calculate the coefficient value of the new node, thus assisting in the one-pass decomposition process; and the *error quantities* on f are used in order to calculate the error quantities in the new node. These quantities receive error information from deleted *orphan* nodes in the subtree under the scope of f and store it until their missing parent is created in the level above, thereby preventing the loss of such information that would occur when an *orphan path* leading to a data item was deleted. In the case of the maximum absolute error algorithm, this information consists of the maximum and minimum error values, $f.max$ and $f.min$, among the deleted orphans. Note that, since the tree is created from left to right, an orphan's relationship to its missing parent is always left. A boolean variable is used to denote whether error information is stored on the fnode. $f.prev$ and $f.next$ are used to navigate in the frontline structure. Finally, the error-tree level $f.l$ to which f corresponds is used for *padding* an incomplete error-tree with nodes after the stream has ended.

4.3 Basic Operations

We now describe in more detail the basic operations of the algorithm, assisted by the variables of Tables 1 and 2. The climbup process updates the current error-tree/frontline structures for every two data values that arrive from the stream. A sketch of its operation is shown in Figure 7. For a new pair of data values, a whole path in the error-tree may be created (as many nodes as the factors of 2 in the number n of data that have already been read). In each loop (lines 4–10) a new error node is created at a higher level. If the fnode f in that level does not point to any error node (either due to previous deletions or because the peak of the climbing process is reached), then c_i is hanged from it. In both cases, the hanged node is duly connected to its parent when

this is created in the next level.

It is important to note that at each iteration at level $l > 1$ a new error-tree node is created, while at the same time a frontline node is released at the previous level (line 13). Let h be the length of the path added by climbup in the error tree. The process will create one new frontline node at level h plus h error-tree nodes. In addition, $h-1$ fnodes are released (one for every level, but the last). Thus, the total space added by climbup is exactly two nodes (one new error-tree leaf node and one new fnode). This space is released by discarding the two nodes with the minimum potential maximum error. Thus the space used by StreamGreedy is always $O(B)$.

In order to compute the error quantities (max_k^l, min_k^l, etc.) for a newly created node (line 11) we access the existing *direct descendants* of c_i using $child_i$ and following its sibling pointers. Note that these *direct descendants* may be multiple, as the left child of c_i and, thereafter, additional *orphan* nodes in its left subtree, may have been previously discarded; in addtion, we access the error quantities that have been stored in the previous frontline node, in order to retrieve error information stemming from deleted orphan paths which would have otherwise been lost.

Algorithm climbup(n, d_1, d_2)
Input: Number n of data already acquired from the Data Stream; last two data items, d_1, d_2

```
1.  f := bottom-most fnode; cur_avg := NULL; l := 0;
2.  while (n > 0 ∧ n mod 2 = 0) // ascending is possible
3.      n := n/2; l := l + 1; //n:order, l:level
4.      if (cur_avg = NULL) then //first loop
5.          avg := (d1 + d2)/2; v:=d1 − avg, ch := NULL
6.      else
7.          avg := (avg + cur_avg)/2; v := cur_avg − avg;
8.          ch := f.prev.ptr; f.prev.ptr := NULL; //de-hang node
9.      if (f.l = l) then s := last sibling of node hanging from f;
10.     c_i := new node (l_i = l, o_i = n, v_i = v, child_i = ch, prev_i = s);
11.     compute error values for c_i from children, fnode below f;
12.     compute MA_i (or MR_i); put c_i in min-heap H;
13.     delete fnode below f;
14.     if (no fnode in level l) then
15.         f := new fnode at level l; f.v := avg;
16.     else cur_avg := f.v;
17.     if (f.ptr=NULL) then //no hanged on f
18.         hang f.ptr := c_i; //hang c_i from f;
19.     f := f.next; // fnode at upper level
```

Figure 7: Ascending and constructing the error-tree on-the-fly while reading the data stream

Line 12 of the climbup operation adds the computed MA_i (or MR_i) for the new node c_i to a min-heap H, used to detect which two error nodes to delete (i.e. the ones with the minimum MA) in order to compensate for the newly occupied memory.

The discarding of a coefficient is similar to the case of the static GreedyAbs algorithm presented in Section 3.1. The main differences are that (i) the discarded node has to be physically deleted from memory and pointers must be adjusted accordingly and (ii) the propagation of the deletion to the error quantities of ancestors and descendants is not straightforward, since some nodes may have been discarded. A sketch for this discard operation is shown in Figure 8. Before the node is discarded, its error quantities are propagated to its ancestors and descendants. The leftmost child $child_k$ of the deleted node (exists such) takes its position in the structure: it is affiliated to the parent and connected to the left sibling of c_k; its rightmost sibling (or itself, in appropriate cases) is attached to the right sibling of c_k; and it substitutes c_k as a hanging node (if applicable). Otherwise, if c_k has no children, its position is taken by its right sibling accordingly.

Algorithm discard(c_k)
Input: error-tree node c_k to be discarded

```
1.  propagate_error(c_k);
2.  if (child_k ≠ NULL) then //handle children/siblings
3.      if (c_k is hanged on frontline) hang child_k in its place;
5.      connect child_k as right sibling of prev_k;
6.      connect last sibling of child_k as left sibling of next_k;
7.  else
8.      if (c_k is hanged on frontline) hang next_k in its place;
9.      connect prev_k and nex_k to each other;
10. if (c_k is first child of par_k) then //handle parent
11.     if (child_k ≠ NULL) then child_par_k:=child_k;
12.     else child_par_k:=next_k;
13. delete c_k
```

Figure 8: Discarding a coefficient c_k

The propagate_error function is illustrated in Figure 9. As in the static case, information about the incurred error is propagated to all existing ancestors and descendants of the deleted node. If no ancestors exist, the information is merged in the fnode that hangs the deleted node's leftmost sibling (it *must* be *hanging* from the frontline structure). If no such sibling exists either, then the deleted orphan node itself is *hanging* from an fnode, on which the error information is stored/merged. This information is used when the missing parent is later created by climbup.

Algorithm propagate_error(c_k)
Input: error-tree node c_k being discarded

```
1.  compute error incurred after deletion of c_k
2.  if (child_k ≠NULL) then
3.      propagate_down(c_k,child_k,error)
4.  if (par_k =NULL)
5.      if c_k is not hanging then
6.          store/merge error in fnode of c_k's leftmost sibling
7.      else
8.          store/merge error in front_k
9.  else
10.     propagate_up(par_k,error)
```

Figure 9: Propagating error up and down

This procedure employs two sub-processes propagate_up and propagate_down, handling the propagation of error upwards and downwards in the sibling error-tree. Propagation is done in a similar way as in the static case. However, upwards error propagation in the sparse sibling error tree requires some elaboration. In the static case, the nodes of discarded coefficients are maintained in the error-tree for the sole purpose of assisting in error propagation; hence a propagation step is always made from a child to its parent at the next level. On the contrary, in the sparse sibling

error-tree, propagation may be made from a node to its parent several levels up if intermediate logical ancestors have been discarded. In the cases where a sibling propagates upwards a new error quantity while the current corresponding quantity on the parent was derived from another sibling, or it was derived from the propagating sibling itself and the propagated version is magnified, the propagation is straightforward. Still, in the possible case that a propagating sibling's previously prevalent error quantity has been diminished, all the other siblings of the same affiliation (left or right) have to be examined in order to determine the new error quantity on the parent.

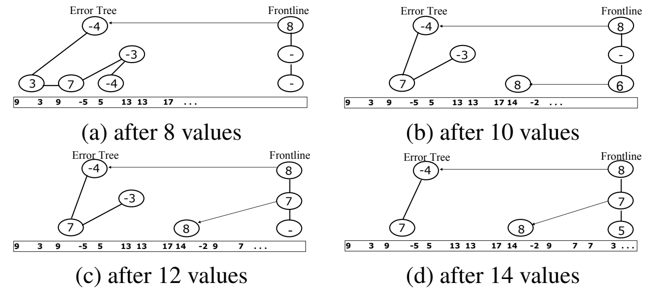
If the total number of stream data has not been a power of two, then the inactivation of the stream leaves the error-tree incomplete, with more than one of its nodes hanging on the frontline. The decomposition is then finalized through a *padding* process that uses average value and level information in the frontline in order to create additional error-tree nodes that turn the current structure into a rooted (sparse) sibling tree, which may be used to reconstruct any data value. By a traditional method [15], a full error-tree is constructed by adding as many zero-value coefficients as necessary. We use a similar coefficient padding approach, yet without adding superfluous zero-value coefficients, but only those necessitated to make the error-tree connected. Details are omitted due to space constraints.

Complexity Analysis The space complexity of GreedyStream is $O(B)$, as discussed; at each step, climbup creates two new (error-tree and/or frontline) nodes and discard deletes two error-tree nodes. The time complexity is $O(B)$ at each step, as $O(B)$ nodes are affected by climbup and discard. If B is large, the complexity of each step converges to that of the static GreedyAbs algorithm; i.e., $O(\log^2 N)$.

4.4 An example

We now illustrate the steps of the StreamGreedy algorithm by an example. Assume that the data stream $\{9, 3, 9, -5, 5, 13, 13, 17, 14, -2, 9, 7, 7, 3, \ldots\}$ is read under a space budget of $B = 6$. After reading the seventh and eighth items, the algorithm discards the first two error tree nodes, resulting in the structure of Figure 10a. The values of the discarded coefficients lead to minimization of the maximum absolute error (to 2) for the first 8 values of the stream. Figure 10b shows the tree after the next two values (14, -2) are processed from the stream. A new node with value 8 is created, which is hanged from a new frontline node at level 1. In addition, two error nodes are deleted (max_abs is currently 5). Figures 10c and 10d show the resulting tree after two more pairs arrive. Observe that the constructed synopsis always has $B = 6$ (error-tree and frontline) nodes.

Assume that we want to use the synopsis after having read 14 items from the stream (Figure 10d). We apply the padding process, to convert the information stored in the *frontline* to a connected error tree that reconstructs any data value, as shown in Figure 11.



(a) after 8 values (b) after 10 values

(c) after 12 values (d) after 14 values

Figure 10: On-the-fly thresholding example

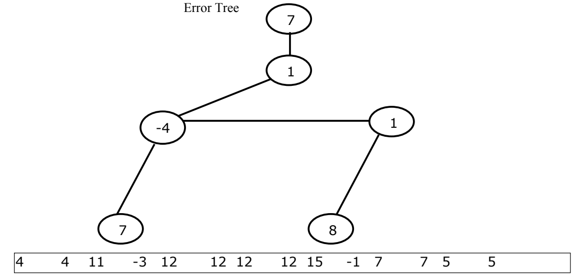


Figure 11: Sparse error tree after padding

4.5 Maximum Relative Error Thresholding for Streaming Data

We have already discussed the extension of the static thresholding solution into a streaming environment when aiming at a minimum max_abs error. An analogous derivation of a streaming algorithm for maximum relative error from the static-case algorithm is not feasible; as we have seen in Section 3.2, the relative error algorithm necessitates additional storage for the accumulated errors on all data leaves of the error-tree, so as to calculate the maximum potential relative error for each coefficient.

Given the importance of relative error minimization thresholding, we device a heuristic to calculate a good estimate for the maximum potential relative error incurred by a coefficient's withdrawal at each step. In the absolute error case, we store four error values on each coefficient node c_k of the error-tree, and calculate MA_k, using Equation 5. In order to compute an estimate for MR_k, while processing the streaming data, we store and maintain at each node c_k:

- The four quantities $max_k^l, min_k^l, max_k^r, min_k^r$, used for max_abs, along with the corresponding data values with these errors.
- The minimum absolute data value m_k^l (m_k^r) in the left (right) subtree together with its accumulated error.
- A *sample* data value s_k^l (s_k^r) in the left (right) subtree together with its accumulated error. The error of this sample serves as a speculative estimate for the maximum relative error in the left (right) subtree. For leaf nodes of the error tree, $s_k^l = m_k^l$ (and $s_k^r = m_k^r$). For intermediate nodes, s_k^l (s_k^r) is the sample in the left (right) subtree with the largest relative error. When a

node c_k is deleted, the sample with the maximum relative error in the subtree of c_k is propagated upwards, as long as its error is larger than the error of the sample in the visited nodes.

The MR_k estimate that our algorithm employs is set to be the maximum relative error incurred by the deletion of c_k to the above eight data values. As our experiments have shown, the maximum relative errors achieved through these estimates are close to the ones of the explicit relative error static algorithm and of the optimal solution itself.

5 Experimental Evaluation

In this section, we evaluate experimentally the effectiveness of the greedy algorithms for static and streaming data, denoted by GSTA and GSTR, respectively. GSTA corresponds to the GreedyAbs algorithm when the goal is to produce a synopsis with minimum absolute error, and to the GreedyRel method, when aiming at maximum relative error minimization. Accordingly, GSTR corresponds to the StreamGreedy for maximum absolute or relative error, depending on the target of thresholding. As a basis of comparison, we use the optimal algorithm [6], denoted by OPT, and also the conventional approach of picking the B coefficients with the largest significance (see Section 2.2.1) — denoted by CON. All algorithms were implemented in C++ and the experiments were run on a Pentium 4 2.26GHz machine with 512MB of RAM.

5.1 Description of Data

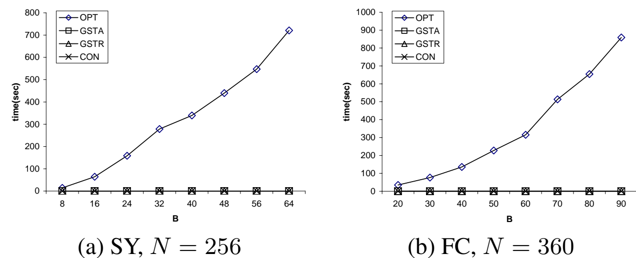
For our experiments, we used synthetically generated and real data. The synthetic datasets (SY) contain random integers, uniformly selected from $[0, 1000)$. In addition, we used three real datasets. The first is extracted from a relation of 581,012 tuples describing the forest cover type for 30 x 30 meter cells, obtained from US Forest Service. FC is a histogram, counting the frequencies of the 360 distinct values of attribute `aspect` in the relation. The frequencies range quite uniformly, averaging at 1613 (standard deviation: 730) and featuring spikes of large values (min value: 499, max value: 6308). The second real dataset contains calibrated and re-sampled ring profiles based on the Voyager 2 spacecraft Photopolarimeter Subsystem (PS) stellar occultation experiments.[2] We assembled a data file of 16,384 float values of mean photon counts received by the PS instrument at its radial location, with the background signal included. The average value in PS is 27.32 and the set has a large standard deviation (14.02). The third dataset (TM) is a sequence of 178,080 sea surface temperature measures extracted from drifting buoys positioned throughout the equatorial Pacific. The average value in TM is 26.75 and the set has a small standard deviation (1.91). FC and TM were downloaded from the UCI KDD Archive[3] and postprocessed.

[2]Available at http://pds-rings.arc.nasa.gov/voyager/datasets/

[3]http://kdd.ics.uci.edu/

5.2 Experimental Results

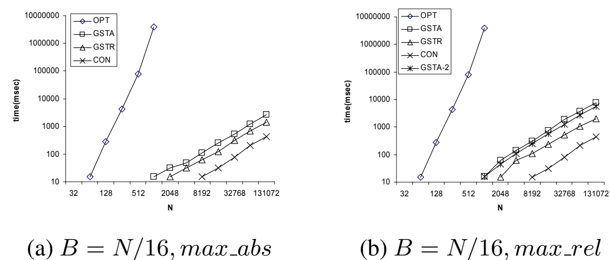
Run-time comparison Figure 12 shows the time performance of all techniques as a function of B for small datasets (aiming at minimization of max_abs). Note that all algorithms finish instantaneously, except from the optimal algorithm, which requires a long time to complete even for small values of B.



(a) SY, $N = 256$ (b) FC, $N = 360$

Figure 12: Run-time comparison

The next experiment evaluates the run-time scalability of the thresholding techniques, with respect to N, using various-sized subsets of the TM dataset and setting $B = N/16$. Figure 13 shows the time required for each algorithm to complete. All algorithms scale well with N, except from OPT which is extremely expensive and inapplicable even for moderately-sized datasets. Naturally, the conventional approach is the fastest of all.[4] For max_rel error minimization, we also used a static implementation (GSTA-2) employing the heuristic error measures introduced in Section 4.5 in order to achieve $O(N \log^2 N)$ time complexity (i.e., same as GSTA for absolute error). This version has lower running-time than the original GreedyRel algorithm (denoted by GSTA). As expected, the streaming versions of our greedy algorithms are faster than the static ones, since they scan the data only once and they do not operate on an initially large dataset. On the other hand, OPT is not scalable and can only be applied for the approximation of tiny datasets.



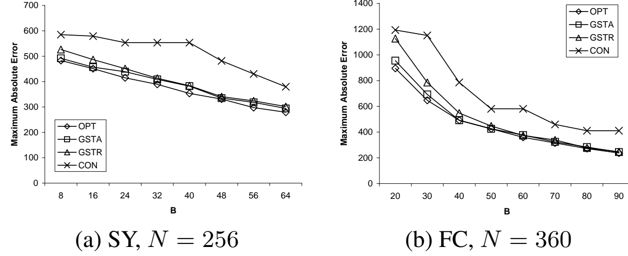
(a) $B = N/16, max_abs$ (b) $B = N/16, max_rel$

Figure 13: Scalability with N

Approximation Quality We first compare the resulting synopses when applying the thresholding algorithms on SY and FC. For these small datasets OPT can be applied, so we can directly evaluate the synopses, using the optimal error as a basis. Figure 14 compares the thresholding methods,

[4]We implemented a streaming version of this algorithm, which constructs coefficients on-the-fly and maintains the set of B most significant ones, using a priority queue.

when aiming at a max_abs minimization. Observe that the error of GSTA is very close to the optimal (4.5% and 3.5% worse than OPT on the average for SY and FC, respectively) for all values of B. GSTR has also low error (7.5% and 10% worse than OPT on the average for SY and FC, respectively), which improves over B. On the other hand, the synopsis obtained by the conventional approach of keeping the greatest B coefficients in terms of significance has a relatively high absolute error (38% and 55% worse than OPT on the average for SY and FC, respectively).



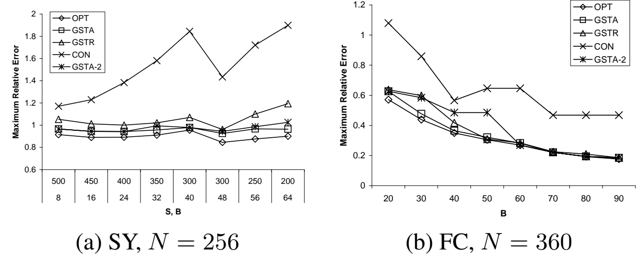
(a) SY, $N = 256$ (b) FC, $N = 360$

Figure 14: Quality comparison for max_abs

Next, we compare the approximation quality of the thresholding methods, when aiming at a max_rel minimization. In this case we compare the maximum relative error achieved by the corresponding optimal algorithm (OPT) and the conventional L_2-minimization scheme (CON) to three different implementations of our greedy techniques: static (GSTA), streaming (GSTR), and the alternative static implementation (GSTA-2). Note that in cases where the optimal algorithm cannot achieve a maximum relative error lower than 1, it resorts to the trivial solution of discarding *all* wavelet coefficients in order to achieve a zero approximation value for all data and a maximum relative error by definition equal to 1. Since this trivial solution can be also achieved without employing computational resources, we decided to tune the sanity bound S for a given B so that the optimal algorithm will achieve maximum relative error less than 1.

Figure 15 shows the qualitative comparison for various values of B when approximating SY and FC. The x-axis of Figure 15a shows the chosen value for S, for each B. In Figure 15b, S is not used, since all values to be approximated in FC are already quite large. The error of GSTA is very close to the optimal in this case as well (6.4% and 4.7% worse than OPT on the average for SY and FC, respectively). GSTR is less effective (on the average, 17% and 11% worse than OPT), yet quite close to optimal when compared to the conventional approach. GSTA-2 performs well for SY (8% worse than OPT), but not well for FC (18% worse than OPT), indicating that the heuristic measures of Section 4.5 are not always effective in choosing a good coefficients set for maximum relative error minimization. The synopsis obtained by CON is always far worse than that of the greedy algorithms.

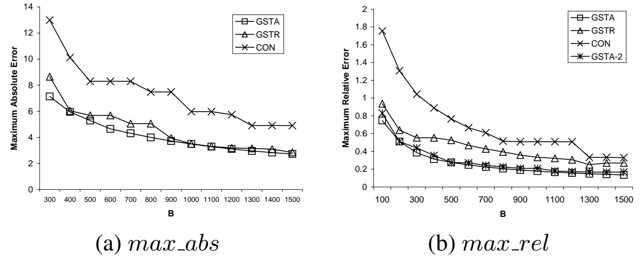
Scalability The last set of experiments verifies the ability of our methods to produce good synopses for larger datasets of different characteristics. We first validate the



(a) SY, $N = 256$ (b) FC, $N = 360$

Figure 15: Quality comparison for max_rel

quality of the approximations, using PS and various values of B. OPT could not be used, due to its extreme space/time requirements. Thus, we only verified the maximum absolute/relative error produced by the other algorithms. Figure 16a shows the maximum absolute reconstruction error of GSTA, GSTR, and CON when the goal is minimization of max_abs. GSTA has good performance, which becomes increasingly better with B. For $B = 600$ or with space budget just 3.6% of N the maximum absolute error stabilizes at a small value (4.65) compared to the standard deviation (14.02). Although we could not run the optimal algorithm for this case, we can speculate that its error will be very close to GSTA, as the trends of Figures 14 show. The error of the one-pass GSTR algorithm is very close to that of GSTA (just 9% higher on the average). CON is inappropriate also for this dataset. Figure 16b compares the algorithms aiming at min_rel minimization. The trend is similar; the static greedy algorithm performs best and the error improves with B. GSTR does not succeed as much in achieving errors similar to GSTA as for smaller datasets (see Figure 15), although it is still better compared to CON (especially for small B).

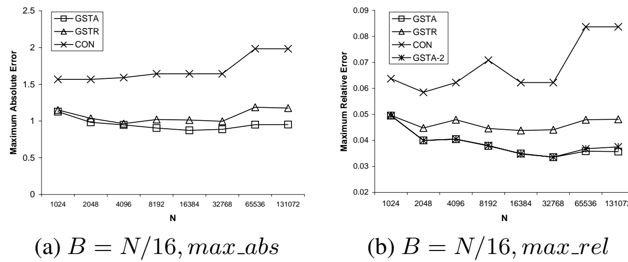


(a) max_abs (b) max_rel

Figure 16: Quality comparison for the PS dataset

Finally, we evaluate the performance of the thresholding algorithms on various-sized subsets of the TM dataset, keeping B constant to $N/16$. Figure 17 shows the maximum errors produced by the synopses for max_abs and max_rel minimization. The results show that accuracy is not affected much by N. In the relative error minimization case, GSTA and GSTA-2 improve a little, since the data pattern does not change much with N and the additional space by a larger B helps improving the compression quality. The errors are very small, due to the low variance of the data. The relative performances of the algorithms are consistent with previous experiments; for max_abs the greedy algorithms have similar results and for max_rel GSTR computes a worse synopsis than the static algorithms, albeit still

much better than that computed by CON.



(a) $B = N/16, max_abs$ (b) $B = N/16, max_rel$

Figure 17: Quality comparison for the TM dataset

6 Discussion

As we have seen in Section 3.2, our static relative error algorithm requires explicit error computation using the synopsized data. This method has been adopted in a streaming environment in Section 4.5 through a heuristic technique with satisfactory results, but not always very close to those of the static solution. On the other hand, the streaming adaptation of GreedyAbs has always very good performance (close to that of the static algorithm). This situation is a result of the inherently more demanding nature of the relative error measure. In the future, we plan to devise alternative heuristics towards achieving better approximations for this measure.

Another direction for future research is the extension of the introduced greedy wavelet thresholding techniques for multi-dimentional wavelets. In the multidimensional case, the optimal solution is computationally harder to derive and approximation techniques have been proposed for it [6]. Our methods are easily extendable, towards this direction; an interesting question is how extensions of greedy algorithms would compare to the techniques of [6].

7 Conclusions

In this paper, we proposed efficient and effective greedy algorithms for wavelet thresholding aiming at the minimization of maximum absolute and relative error metrics. We studied both cases of (i) static, apriori available and directly accessible data and (ii) data that arrive from a data stream and must be processed in one-pass, by the order they arrive, at the availability of a small memory budget. Contrary to the optimal max-error thresholding algorithm [6], our methods are not constrained by large space/time complexities, thus they can be applied in large real-world problems, such as signal compression and approximate query evaluation. In addition, as demonstrated by experiments, our time-efficient solutions achieve near-optimal quality of results. Finally, we demonstrate that the conventional approach of choosing the largest normalized coefficients, although effective in minimizing the L_2 metric, does not produce synopses with small maximum individual errors.

References

[1] C. Blatter. *Wavelets: A Primer*. A K Peters, 1998.

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

[3] C. S. Burrus, R. A. Gopinath, and H. Guo. *Introduction to Wavelets and Wavelet Transforms*. Prentice Hall, 1998.

[4] K. Chakrabarti, M. Garofalakis, R. Rastogi, and K. Shim. Approximate query processing using wavelets. *VLDB Journal*, 10(2-3):199–223, 2001.

[5] M. Garofalakis and P. B. Gibbons. Wavelet synopses with error guarantees. In *Proc. of SIGMOD Conf.*, pages 476–487, 2002.

[6] M. Garofalakis and A. Kumar. Deterministic wavelet thresholding for maximum-error metrics. In *Proc. of PODS*, pages 166–176, 2004.

[7] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. J. Strauss. One-pass wavelet decompositions of data streams. *IEEE TKDE*, 15(3):541–554, 2003.

[8] A. Graps. An introduction to wavelets. *IEEE Computational Sciences and Engineering*, 2(2):50–61, 1995.

[9] S. Guha, C. Kim, and K. Shim. XWAVE: Approximate extended wavelets for streaming data. In *Proc. of VLDB Conf.*, pages 288–299, 2004.

[10] S. Guha, K. Shim, and J. Woo. REHIST: Relative error histogram construction algorithms. In *Proc. of VLDB Conf.*, pages 300–311, 2004.

[11] P. J. Haas and A. N. Swami. Sequential sampling procedures for query size estimation. In *Proc. of SIGMOD Conf.*, pages 341–350, 1992.

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

[13] Y. Matias, J. S. Vitter, and M. Wang. Wavelet-based histograms for selectivity estimation. In *Proc. of SIGMOD Conf.*, pages 448–459, 1998.

[14] S. Papadimitriou, A. Brockwell, and C. Faloutsos. Adaptive, hands-off stream mining. In *Proc. of VLDB Conf.*, pages 560–571, 2003.

[15] E. J. Stollnitz, T. D. Derose, and D. H. Salesin. *Wavelets for computer graphics: theory and applications*. Morgan Kaufmann, 1996.

[16] J. S. Vitter and M. Wang. Approximate computation of multidimensional aggregates of sparse data using wavelets. In *Proc. of SIGMOD Conf.*, pages 193–204, 1999.