Distributed Wavelet Thresholding for Maximum Error Metrics

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

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

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

ABSTRACT

Modern data analytics involve simple and complex computations over enormous numbers of data records. The volume of data and the increasingly stringent response-time requirements place increasing emphasis on the efficiency of approximate query processing. A major challenge over the past years has been the efficient construction of fixed-space synopses that provide a deterministic quality guarantee, often expressed in terms of a maximum error metric. For data reduction, wavelet decomposition has proved to be a very effective tool, as it can successfully approximate sharp discontinuities and provide accurate answers to queries. However, existing polynomial time wavelet thresholding schemes that minimize maximum error metrics are constrained with impractical time and space complexities for large datasets. In order to provide a practical solution to the problem, we develop parallel algorithms that take advantage of key-properties of the wavelet decomposition and allocate tasks to multiple workers. To that end, we present i) a general framework for the parallelization of existing dynamic programming algorithms, ii) a parallel version of one such DP-based algorithm and iii) a new parallel greedy algorithm for the problem. To the best of our knowledge, this is the first attempt to scale algorithms for wavelet thresholding for maximum error metrics via a state-of-the-art distributed runtime. Our extensive experiments on both real and synthetic datasets over Hadoop show that the proposed algorithms achieve linear scalability and superior running-time performance compared to their centralized counterparts. Furthermore, our distributed greedy algorithm outperforms the distributed version of the current state-of-the-art dynamic programming algorithm by 2 to 4 times, without compromising the quality of results.

1. INTRODUCTION

The technological and societal developments of our era have resulted in an unprecedented production and processing of enormous data volumes, referred to with the term "Big Data". Applications, businesses, government organizations and digital infrastructures alike contribute to this Big Data reality. Processing over huge, heterogeneous and often imprecise data (the Internet of Things [3] is such an example) is considered common practice nowadays.

Approximate query processing has emerged as a viable alternative for dealing with the huge amount of data and the increasingly stringent response-time requirements [5]. Due to the *exploratory nature* of many data analytics applications, there exists a number of scenarios in which an exact answer is not required. Users are often willing to forgo accuracy in favor of achieving better response times. In one such example, visualizing available tradeoffs between accuracy and execution-time helps users to fine-tune the execution of queries [30]. Moreover, approximate answers obtained from appropriate *synopses* of the data may be the only option when the base data is remote and unavailable[7].

To that end, several approximation techniques have been developed, including: sampling [5, 14, 4], histograms [19, 15, 20], wavelets [9, 13, 22, 24, 23] and sketches [16, 6]. Wavelet decomposition [29] provides a very effective data reduction tool, with applications in data mining [25], selectivity estimation [26], and approximate and aggregate query processing of massive relational tables [9, 31] and data streams [17, 11]. 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 [9].

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 linear-time deterministic thresholding scheme that minimizes the overall mean squared error [29]. Still, the synopses produced by this method exhibit significant drawbacks [13], 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 [12, 13].

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA,USA

DOI: http://dx.doi.org/10.1145/2882903.2915230

However, the existing algorithms that minimize maximum error metrics are strictly centralized and are usually based on dynamic programming (DP) approaches, that demand a lot of communication, memory and processing power. As such, they cannot be executed over modern analytics platforms and fail to scale to big datasets. In [22], *GreedyAbs*, a heuristic-based solution is proposed, that is linear in practice. GreedyAbs is more efficient in terms of running-time than the DP-based algorithms but at the cost of loosened quality guarantees. Yet, this algorithm cannot scale to big data either, as it follows a sequential path of execution that prevents a data-parallel approach.

In this work, we present a general framework for adapting the existing DP algorithms to run over scalable, high-throughput modern platforms. We also present *DGreedyAbs*, a new distributed greedy algorithm for the problem. In this way, we offer scalable solutions to the wavelet thresholding for maximum error metrics problem and thus, we enhance the usability of wavelets in modern applications. To our knowledge, this is the first effort to adapt such algorithms towards big data scenarios. In summary, we make the following contributions:

- We present a general and scalable framework for the parallelization of all the existing DP algorithms for the problem and calculate its communication overhead. Our framework is based on a novel error tree decomposition that allows parallel processing of DP table rows. We demonstrate that our proposed partitioning scheme has general applicability, proving to be very efficient for other wavelet-based algorithms. In order to demonstrate the benefits of our approach, we create the *DIndirectHaar* algorithm, by applying the proposed framework on *IndirectHaar* [24], that is the current state-of-the-art. For datasets demanding intensive computations, we show that DIndirectHaar outperforms IndirectHaar by a factor of 2.7.
- We propose *DGreedyAbs*, a new distributed, heuristic-based algorithm and study its computational complexity. Our algorithm is based on three key ideas: 1) locality-preserving partitioning similar to the one applied for the DP algorithms, 2) speculative execution of the centralized greedy algorithm and 3) merging and filtering of intermediate results. DGreedyAbs is 2×–4× faster than DIndirectHaar and over 7 times faster than the centralized greedy algorithm. Moreover, our experiments show that, compared to GreedyAbs, the achieved performance exhibits no quality degradation.
- We implement[1] DGreedyAbs and DIndirectHaar on top of the Hadoop processing framework and perform an extensive experimental evaluation using both synthetic and real datasets. Previous approaches on the problem used datasets of up to 262K datapoints. To put emphasis on the merits of our approach, we experiment with datasets of up to 537M datapoints, demonstrating the applicability of our algorithms over big data scenarios.

The remainder of this paper is organized as follows: Section 2 presents the basic theoretical background for the wavelet decomposition. In Section 3, we give an overview of the related work. Section 4 proposes a novel framework for the parallelization of the DP algorithms and Section 5 presents DGreedyAbs. Finally in Section 6, we experimentally evaluate our algorithms and in Section 7, we present the conclusions.

[1]https://github.com/giagulei/dwmaxerr.git

Table 1: Wavelet decomposition example

Resolution	Averages	Detail Coef.
3	$[5, 5, 0, 26, 1, 3, 14, 2]$	–
2	$[5, 13, 2, 8]$	$[0, -13, -1, 6]$
1	$[9, 5]$	$[-4, -3]$
0	$[7]$	$[2]$

2. WAVELET PRELIMINARIES

Wavelet analysis is a major mathematical technique for hierarchically decomposing functions in an efficient way. The wavelet decomposition of a function consists of a coarse overall approximation together with detail coefficients that influence the function at various scales [29]. Thus, wavelets can successfully approximate sharp discontinuities. In this Section, we provide the basic background to the Haar wavelet transform and its properties.

2.1 Haar Wavelets

Haar wavelets constitute the simplest possible orthogonal wavelet system. Assume a one-dimensional data vector A containing $N = 8$ data values $A = [5, 5, 0, 26, 1, 3, 14, 2]$. The Haar wavelet transform of A can be computed as follows: We first average the values in a pairwise fashion to get a new "lower-resolution" representation of the data with the following average values: $[5, 13, 2, 8]$. The average of the first two values (i.e., 5 and 5) is 5, the average of the next two values (i.e., 0 and 26) is 13, 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 differences of the (second of the) averaged values from the computed pairwise average. In our example, for the first pair of averaged values, the detail coefficient is 0 (since $5 - 5 = 0$) and for the second is -13 ($13 - 26 = -13$). 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, in order of increasing resolution (see Table 1). In our example, the wavelet transform (also known as the wavelet decomposition) of A is $W_A = [7, 2, -4, -3, 0, -13, -1, 6]$. 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 zeros) introduces only small errors when reconstructing the original array and thus results to a very effective form of lossy data compression.

2.2 Error Trees

The *error tree*, introduced in [26], 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 data nodes in the subtree rooted at c_k. This notation is extended to $leftleaves_k$ ($rightleaves_k$) for the left (right) sub-tree of c_k. We denote

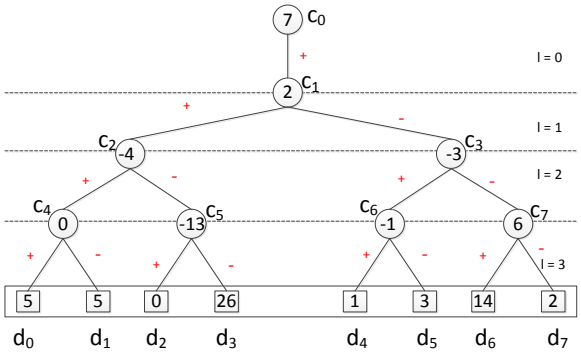


Figure 1: An error tree that illustrates the hierarchical structure of the Haar wavelet decomposition

$path_k$ as the set of all nodes with nonzero coefficients in T which lie on the path from a node c_k (d_k) to the root of the tree T. Moreover, for any two data nodes d_l and d_h, we use $d(l:h)$ to denote the range sum $\Sigma_{i=l}^{h} d_i$.

Given the error tree representation T of a one-dimensional Haar wavelet transform, we can reconstruct any data value d_i using only the nodes that lie on the path from d_i to the root of T. That is $d_i = \Sigma_{c_j \in path_i} \delta_{ij} \cdot c_j$, where the factor $\delta_{ij} = 1$ if $d_i \in leftleaves_j$ or $j = 0$ and $\delta_{ij} = -1$ otherwise. For example, in Figure 1, value $d_5 = 7 - 2 - 3 - (-1) = 3$. A range sum $d(l:h)$ can be computed using only nodes $c_j \in path_l \cup path_h$, by $d(l:h) = \Sigma_{c_j \in path_l \cup path_h} x_j$, where $x_j = (h - l + 1) \cdot c_j$, if $j = 0$ and $x_j = (|leftleaves_{j,l:h}| - |rightleaves_{j,l:h}|) \cdot c_j$, otherwise. Here, $leftleaves_{j,l:h} = leftleaves_j \cap \{d_l, d_{l+1}, .., d_h\}$ and $rightleaves_{j,l:h} = rightleaves_j \cap \{d_l, d_{l+1}, .., d_h\}$. That means that node c_j contributes to the range sum $d(h:l)$ 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, $d(3:6) = (-1)\cdot(-13) + (-1)\cdot(-4) + (-2)\cdot 2 + 4\cdot 7 + 1\cdot(-3) + 6 = 44$.

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.

2.3 Wavelet Thresholding

The complete Haar wavelet decomposition W_A of a data vector A is a representation of equal size as the original array. 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 non-selected coefficients are implicitly set to zero. The resulting wavelet synopsis $\hat{W}_A$ can be used as a compressed approximate representation of the original data.

We consider the decomposition of Figure 1 and assume that only coefficients $\{c_0, c_5, c_3\}$ are retained in the synopsis, whereas all the rest are implicitly set to zero. In this case, the reconstructed value for d_5 is $\hat{d}_5 = 7 - 3 = 4$ instead of the actual value $d_5 = 3$. For assessing the quality of a wavelet synopsis, many aggregate error-measures have been proposed [10]. Among the most popular metrics are the *mean squared error* (L_2), the *maximum absolute error* (max_abs) and the *maximum relative error* (max_rel):

$$L_2\left(W_A, \hat{W}_A\right) = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(\hat{d}_i - d_i\right)^2} \quad (1)$$

$$max\_abs\left(W_A, \hat{W}_A\right) = max_{i=1}^{N}\left\{|\hat{d}_i - d_i|\right\} \quad (2)$$

$$max\_rel\left(W_A, \hat{W}_A\right) = max_{i=1}^{N}\left\{\frac{|\hat{d}_i - d_i|}{max\{|d_i|, S\}}\right\} \quad (3)$$

In the above equations, $\hat{d}_i$ denotes the reconstructed (approximate) value for d_i and S is a *sanity bound* used to prevent the influence of very small values in the aggregate error [31, 12, 13].

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). The first observation is that coefficients of larger values are more important, since their absence causes a larger absolute error in the reconstructed values. Second, a coefficient's significance is larger if its level in the error tree is higher, as it participates in more reconstruction paths of the error tree. Putting both together, the significance c_i^* of a coefficient is defined by $c_i^* = |c_i|/\sqrt{2^{level(c_i)}}$, where $level(c_i)$ denotes the level of resolution at which the coefficient resides (0 corresponds to the "coarsest" resolution level).

Accordingly, the conventional thresholding scheme is to retain the B wavelet coefficients with the greatest significance. It has been shown [29] that this approach 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. Metrics max_abs and max_rel prove more robust error measures [12, 13], since they set a maximum error guarantee on individual values. The problem of minimizing these error metrics can be formulated as follows:

PROBLEM 1. *Given a data vector A of size N and a budget B, construct a representation $\hat{W}_A$ of A that minimizes a maximum error metric, while it retains at most B non-zero coefficients.*

For ease of readability, we also define the dual of Problem 1:

PROBLEM 2. *Given a data vector A of size N and an error bound ϵ, construct a representation $\hat{W}_A$ of A such that $max\_abs \leq \epsilon$ and the number of non-zero entries s^* in $\hat{W}_A$ is minimized.*

In this work, we focus on designing algorithms for Problem 1 that can specifically scale in big data scenarios. The existing algorithms for the problem either need to load the whole data set in memory or work on a small working set and make very often disk accesses to update it. The increasing sizes of data to be processed render centralized approaches (with excessive disk accesses) unusable in terms of performance and scalability. In this work, we instead propose a novel problem decomposition to smaller local sub-problems that can be more easily handled. Following that, we utilize partial and parallel computed solutions to derive the final one.

Table 2: Notation

Symbol i∈0..N-1	Semantics
A	Input data array
W_A	Wavelet transform array
N	Number of data points
B	Target size of synopsis
T_i	Error tree rooted at node i
$T_L(c_i)$ ($T_R(c_i)$)	Sub-tree rooted at left (right) child of node i
d_i	Data value at cell i of the data array
$\hat{d}_i$	Reconstructed data value at cell i
$leaves_i$	Set of data nodes in T_i
c_i	Wavelet coefficient at cell i
M	Matrix used by DP algorithm
err_i	Signed accumulated error for d_i
R	Size of the root sub-tree
S	Size of a base sub-tree

3. RELATED WORK

In this Section, we present an overview of the existing approaches to Problem 1 and discuss the relation of our work to past research. In [12], a probabilistic DP algorithm that minimizes the maximum error metrics was proposed for the wavelet thresholding. The running-time of the algorithm is $O\left(N\delta^2 Blog\left(\delta B\right)\right)$, where δ is a quantization factor. However, as there is always a possibility of a "bad" sequence of coin flips, this approach can lead to a poor quality synopsis.

For this reason, a deterministic DP-based approach is proposed in [13]. Unfortunately, the optimal solution provided has a high time complexity of $O\left(N^2 BlogB\right)$, where N is the total number of coefficients and B the synopsis size.

In [23], the Haar+ tree is presented. This is a modified error tree that allows the design of a DP algorithm with running-time complexity of $O\left(\left(\frac{\Delta}{\delta}^2\right) NB\right)$, where Δ is the range of dataset values.

These solutions are very expensive in terms of time and space complexity. 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 [9].

In order to decrease space complexity, Guha introduces a generally applicable, space efficient technique [18] for all these DP-based approaches. This technique only needs to keep a small subset of the data in memory while all the rest reside on disk. Nevertheless, such an approach considerably increases the I/O cost.

Furthermore, all the approaches mentioned so far contain, in their complexity formula, a term for budget B which can be $O(N)$ and can thus lead to quadratic or cubic complexity. Such running-times are prohibitive for big datasets, which may be in the order of gigabytes or terabytes. In order to eliminate term B from the complexity, a different DP-based approach is proposed in [24], where Problem 2 [24, 27, 28] is exploited and the resulting complexity is $O\left(\left(\frac{\mathcal{E}}{\delta}\right)^2 N\left(log\epsilon^* + logN\right)\right)$, where $\mathcal{E}$ is the minimum maximum error that can be achieved with $B-1$ coefficients and ϵ^* is the real maximum error. This algorithm is considered to be the current state-of-the-art for the problem, as it provides the optimal data reconstruction for the given budget and has the best running-time complexity among the corresponding DP algorithms.

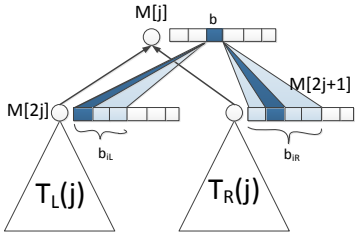


Figure 2: DP recursion on the error tree. Node c_j combines the M-rows of its children in order to produce $M[j]$

However, all of these algorithms run in a centralized fashion and, to the best of our knowledge, there is no proposed solution that adapts them to a distributed environment. Thus, problems like excessive demand for main memory capacity and disk I/O have not yet been resolved.

In order to decrease running-time, greedy algorithms have been proposed [22] for the minimization of the maximum absolute and relative error with worst-case running-time complexities of $O\left(Nlog^2 N\right)$ and $O\left(Nlog^3 N\right)$ respectively. These algorithms present almost linear behavior in practice and require less memory capacity than most of the DP-based ones. Nevertheless, as data scales close to the memory constraints of the machine, their performance significantly deteriorates. Moreover, these algorithms have inherent difficulties in their parallelization and thus, the decomposition to local sub-problems is not an easy task to accomplish.

Finally, the work in [21] considers a distributed setting for the wavelet decomposition, implemented on top of Hadoop, but it only targets the conventional thresholding scheme, which is a considerably easier task.

4. SCALING DP ALGORITHMS

Since the majority of the proposed algorithms for Problem 1 are based on dynamic programming, in this Section we present a general framework that can be used for their parallelization and efficient execution over modern distributed platforms. To achieve that, we exploit the structure of the error tree as well as the local properties of these algorithms and propose a locality-preserving partitioning scheme.

In all these algorithms, each row of the DP-matrix M is assigned to a node of the error tree. The contents of such a row differ between algorithms. Despite the different structure of the rows of M, all these algorithms follow a bottom-up fashion, where the rows corresponding to the leaves of the error tree are computed first. The row for each internal node is computed by combining the already computed rows of its children according to an optimality criterion. To compute the values for a single cell of a row j, many cells of the children-rows are examined and, eventually the one that optimizes a defined metric is selected, according to each algorithm. Thus, computing the row for any node of the error tree, demands two more rows to be in memory.

For example, in the MinRelVar algorithm presented in [12], each row $M[j]$ stores three values for every possible space allotment b to the sub-tree rooted at c_j. Thus, each cell $M[j,b]$ of a row is 3-dimensional and contains the following three values: $M[j,b].v, M[j,b].y, M[j,b].l$, which represent: i) the minimum error, ii) the probability to retain

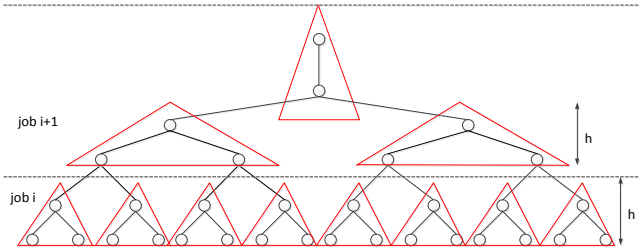


Figure 3: Partitioning used for parallelizing DP-based algorithms for Problem 1

coefficient c_j, and iii) the space allotment for the left subtree respectively. In order to compute $M[j,b]$, the algorithm needs to examine all $M[2j, b_{i_L}], b_{i_L} \in [0, b - M[j,b].y]$ and all $M[2j+1, b_{i_R}], b_{i_R} \in [0, b - M[j,b].y - b_{i_L}]$ and finally select the cells from $M[2j]$ and $M[2j+1]$ that incur the minimum error $M[j,b].v$. This procedure is illustrated in Figure 2. The shadowed cells represent the examined values and the bold colored ones the finally selected values.

In Figure 2, we observe that the left and right sub-tree of node c_j ($T_L(j)$ and $T_R(j)$ respectively) can be computed independently of each other. Based on this observation, the idea is to apply a partitioning scheme that hierarchically decomposes the error tree to sub-trees of a fixed height $h, h < logN$. Such a partitioning is presented in Figure 3 and results to $\lceil \frac{logN}{h} \rceil$ layers of sub-trees. We denote $Layer_i$ to be all the sub-trees located in layer i and it holds that:

$$|Layer_i| = \begin{cases} \frac{N}{2^{h \cdot i + i - 1}} & i = 1, .., \lfloor \frac{logN+1}{h+1} \rfloor \\ 1 & i = \lceil \frac{logN+1}{h+1} \rceil \end{cases} \quad (4)$$

For the parallelization of the existing DP algorithms for both Problems 1 and 2, we use Algorithm 1. The idea is to first run the DP algorithm in parallel over the sub-trees of the bottommost layer. When the processing is over, the computed rows for the roots of these sub-trees are sent over to the next layer in order to repeat the same process towards the root. More specifically, if the local root is the node c_j, the emitted key-value is $(j, M[j])$. The workers of the next stage collect the emitted key-values and repeat the same process. Naturally, a proper partitioning should be applied between different stages, in order to preserve the sub-tree locality in the next layer.

Algorithm 1 Parallel execution of a DP algorithm for Problem1

Require: Data size N, sub-tree height h
1: Partition the error tree to sub-trees of fixed height h.
2: $i = 1$
3: **while** $i \leq \lfloor \frac{logN+1}{h+1} \rfloor$ **do**
4: **if** $i > 1$ **then** Combine M-rows from layer $i-1$
5: **for all** $T_j \in Layer_i$ in parallel **do**
6: Run DP on T_j
7: Send the computed row of node j to the next layer
8: **end for**
9: $i = i + 1$
10: **end while**
11: Run DP on topmost sub-tree.

As a distributed approach, it is clear that this idea incurs a communication overhead. For every sub-tree of the error tree, the row of M that corresponds to the local root is transferred over to the workers of the next stage. Let $|M[j]|$ denote the size of the row corresponding to node c_j. Then, according to Equation 4, the communication overhead for the i-th stage is:

$$O\left(|Layer_i| \cdot max_{j \in Layer_i}\{|M[j]|\}\right) = O\left(\frac{1}{2^{h \cdot i + i - 1}} N \cdot max_{j \in Layer_i}\{|M[j]|\}\right) \quad (5)$$

and thus, the overall communication overhead:

$$O\left(\sum_{i=1}^{\lfloor \frac{logN+1}{h+1} \rfloor} \frac{1}{2^{h \cdot i + i - 1}} N \cdot max_{j \in Layer_i}\{|M[j]|\}\right) = O\left(\frac{N \cdot max\{|M[j]|\}}{2^h}\right) \quad (6)$$

Equation 6 represents the generic communication complexity of all DP algorithms when our partitioning scheme is applied. The maximum M-row size $max\{|M[j]|\}$, which determines the complexity, depends on the used algorithm.

After the completion of Algorithm 1, it is only the optimal approximation error for Problem 1 (synopsis size for Problem 2) that is computed and not the synopsis itself. To compute the synopsis, all DP algorithms require one additional step: a top-down recursive procedure on the error tree in order to select the appropriate coefficients. Starting from the root this time, we re-enter the sub-problem of the topmost sub-tree and select the coefficients to retain. When the processing of the topmost sub-tree is over, we know which coefficients are retained from this sub-tree and also the leaves of the sub-tree know which cells of the M-rows of their children are the best choice in order to obtain the optimal synopsis. Thus, each leaf-node of the topmost sub-tree sends a message to its children to inform them about the optimal choice they can make. With this message, the children recursively re-enter the sub-problems of the next layer of sub-trees.

Unfortunately, the space complexity of most DP algorithms for Problem 1 relies on budget B. Thus, for a bad case scenario where B is large enough ($O(N)$), matrix M may not fit in memory even for a moderate or small-sized sub-tree. Furthermore, in such a case, the communication overhead may be also too high to sustain. For example, for the algorithm in [12] it holds that $max\{|M[j]|\} = O(B \cdot \delta)$. If we substitute this quantity in Equation 6, the communication complexity is $O\left(\frac{NB\delta}{2^h}\right)$, which can become $O\left(N^2\right)$.

In order to avoid the impact of budget B, we focus on Problem 2 instead. The DP algorithm that solves the dual problem and on which we are going to apply the proposed framework is MinHaarSpace [24]. At each visited node c_j, MinHaarSpace computes the corresponding M-row, $M[j]$. $M[j]$ holds an entry $M[j,v]$ for each possible *incoming* value v at node c_j. An *incoming* value v at node c_j is a value reconstructed in the path of ancestor coefficients from the root node up to c_j. For example, in Figure 1, the incoming value of c_2 is $7 + 2 = 9$. For a specified incoming value v, $M[j,v]$ is a 3-dimensional cell that contains: (i) the minimum number of non-zero coefficients that need to be retained in the sub-tree T_j, (ii) the optimal value[2] to assign at c_j, and (iii) the actual minimum error in the scope of c_j.

[2]We use MinHaarSpace for unrestricted wavelets

By applying the proposed framework, we create algorithm *DMHaarSpace*, the new distributed version of MinHaarSpace. The workers of DMHaarSpace execute MinHaarspace over local data and emit the M-row of the local-root node, in a manner similar as described before. The running-time complexity of DMHaarSpace is the same as that of MinHaarspace, i.e., $O\left(\left(\frac{\epsilon}{\delta}\right)^2 NlogN\right)$ and the communication complexity is $O\left(\frac{N\epsilon}{\delta 2^h}\right)$, as derived from Equation 6 and the size of the M-row that is $O\left(\frac{\epsilon}{\delta}\right)$ [24], where δ is a user-defined parameter that quantizes the solution space.

Since DMHaarSpace targets Problem 2, we need to run the algorithm multiple times to derive the final solution. For this reason in [24], there is the IndirectHaar that solves Problem 1 by running MinHaarSpace multiple times. Algorithm 2 presents DIndirectHaar, a modified version of IndirectHaar that uses DMHaarSpace instead of MinHaarSpace.

As Algorithm 2 describes, the DIndirectHaar algorithm performs binary search in the space of possible errors. Obviously, this results in multiple distributed jobs of input size N. Furthermore, in order to compute the lower and upper error bounds (lines 1-2), an overhead of two extra jobs is required. For the lower bound, we compute the (B+1)-largest coefficient. Each worker emits its local wavelet coefficients in reverse order, i.e., largest first, and in a next step these coefficients are merged and the first $B+1$ are retained. For the upper bound, assuming that a B-term synopsis fits in memory, we load the B-largest-terms synopsis in the main memory of each worker and we bottom-up compute the max_abs.

Algorithm 2 DIndirectHaar

1: e_u =maximum absolute error for B-largest-terms synopsis
2: $e_l = (B+1)$-largest coefficient
3: $e_{low} = e_l$;$e_{high} = e_u$
4: **while** not finished **do**
5: $e_{mid} = \frac{e_{high}+e_{low}}{2}$
6: $\hat{W}_A$ =DMHaarSpace(e_{mid});$\bar{B}$ =size of $\hat{W}_A$
7: $\bar{e}$ =actual maximum absolute error of $\hat{W}_A$
8: **if** $\bar{B} < B$ **then**
9: $\tilde{W}_A$ =DMHaarSpace($< \bar{e}$);$\tilde{B}$ =size of $\tilde{W}_A$
10: **if** $\tilde{B} > B$ **then** finished=1
11: **else** $e_{high} = \bar{e}$
12: **else**
13: **if** $\bar{B} > B$ **then** $e_{low} = e_{mid}$
14: **else** finished=1
15: **end if**
16: **end while**

5. A DISTRIBUTED GREEDY APPROACH

As the DP-based solutions incur high computational overhead, there is often a need for a faster approach at the cost of approximation quality. This is exactly what the GreedyAbs [22] algorithm achieves. However, as explained in Section 3, this algorithm is not easily parallelizable and cannot scale for big datasets. In this Section, we present a fully parallelizable version of the greedy algorithm based on: (i) a partitioning scheme similar to the one presented in Section 4, (ii) speculative execution of the centralized algorithm, (iii) merging and filtering of results. We present our idea in detail for the maximum absolute error metric and discuss the modifications needed to support the maximum relative error metric problem.

5.1 GreedyAbs

For the ease of understanding of our algorithm, we first give a description of the GreedyAbs algorithm presented in [22]. Let $err_j = \hat{d}_j - d_j$ be the signed accumulated error for a data node d_j in a synopsis $\hat{W}_A$, 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 contribute on the running synopsis, if discarded:

$$MA_k = max_{d_j \in leaves_k}\{|err_j - \delta_{jk} \cdot c_k|\} \quad (7)$$

Computing MA_k normally requires information about all err_j values 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 all err_j values at each step and the cost required to update MA_k values after the removal of a coefficient.

A more efficient solution for updating MA_k is reached by exploiting the fact that the removal of a coefficient equally affects the signed costs of all data values in its left or right sub-tree. For example, in Figure 1, the removal of coefficient $c_2 = -4$ increases the signed errors of data nodes d_0, d_1, and decreases the signed errors of d_2, d_3 by 4. 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. These are the maximum and minimum signed errors for the $leftleaves_k$ and $rightleaves_k$, and are denoted by max_k^l, min_k^l, max_k^r, and min_k^r, respectively. It follows that Equation 7 is equivalent to:

$$MA_k = max\{|max_k^l - c_k|, |min_k^l - c_k|, |max_k^r + c_k|, |min_k^r + c_k|\} \quad (8)$$

In the complete wavelet decomposition, these four quantities are all 0, since $err_j = 0, \forall d_j$. Thus, $MA_k = |c_k|, \forall k$ and the greedy algorithm removes the smallest $|c_k|$ first. In order to efficiently decide which coefficient to choose next, all coefficients are organized in a min-heap structure based on their MA_k. After the removal of a coefficient c_k, err_j for all $leaves_k$ changes, so the information of all descendants and ancestors of c_k must be updated. 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 values are updated as necessary. While updating error quantities and MA values, the position of c_k's descendants and affected ancestors are dynamically updated in the heap. These procedure of removing nodes is repeated until only B nodes are left on the tree.

Another important thing to note is that the maximum absolute error 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. For this

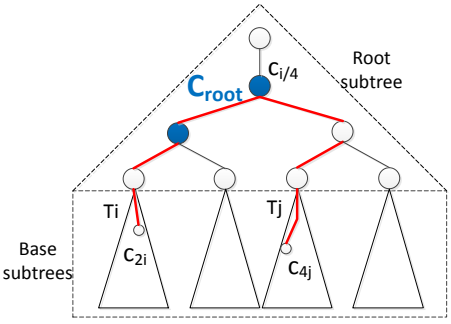


Figure 4: Partitioning used for parallelizing GreedyAbs. The red line illustrates an example of communication of two base sub-trees through the root sub-tree. The blue-filled nodes show a possible C_{root} set.

reason, we 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 (B coefficients left, B-1 coefficients left, etc) produced at the last B steps of the algorithm, the one with the minimum *max_abs* is kept.

5.2 Scaling the Greedy Algorithm

GreedyAbs presents an inherent drawback for its parallelization. At each step, the algorithm needs global knowledge of the whole error tree. To solve the problem in parallel, we need to consider a partitioning similar to the one we used for the parallelization of the DP algorithms. In the proposed scheme, the error tree is partitioned into one *root sub-tree* and many *base sub-trees*, as shown in Figure 4.

At each iteration of GreedyAbs, the node c_k with the smallest MA is selected to be discarded. After its deletion, all the other nodes that lie either in $path_k$ or T_k may update their MA values. Ideally, we would like to take decisions at each base sub-tree independently of each other. For the parallelization of the algorithm, the main difficulty is that the base sub-trees communicate with each other through the root sub-tree. For example, consider a scenario, like the one depicted in Figure 4, where node c_{2i} is selected to be removed from the base sub-tree T_i and node c_{4j} is selected from T_j. The removal of c_{2i} updates the MA value of node $c_{i/4} \in path_{2i}$ to $MA'_{i/4}$. Suppose also that for node c_{4j} holds $c_{i/4} \in path_{4j} \wedge \left(MA_{4j} < MA_{i/4}\right)$ and also $MA'_{i/4} < MA_{4j}$. This means that the removal of c_{2i} in T_i changed the preference of GreedyAbs in T_j. This example shows that the communication of base sub-trees through the root sub-tree can change the order in which nodes are discarded and interferes with a straight-forward parallel implementation.

In order to proceed towards a parallel and correct computation, we need to offer more isolation to the partitions base sub-trees. The idea behind our solution is the following. Let us assume that we somehow know which nodes of the root sub-tree are retained in the final synopsis. Let us call this set of nodes C_{root}. Having selected the retained nodes from the root sub-tree, we can remove the remaining root sub-tree and there are $B - |C_{root}|$ nodes that remain to be selected.

Consider now a base sub-tree T_j. The deletion of the nodes $c_i \in$ root sub-tree$\backslash C_{root}$ incurs an incoming error to T_j. For example, in the error tree of Figure 1, if we delete nodes $\{c_0, c_2\}$, there is an incoming error $-7 - 4 = -11$ to sub-tree T_5. Thus, if the incoming error to sub-tree T_j is e_{in}, we set then the signed accumulated errors to: $err_i = \delta_j e_{in}, \forall d_i \in T_j$ ($\delta_j = -1$ if T_j is a left sub-tree and 1, otherwise) and we run GreedyAbs on T_j. The output of $GreedyAbs\,(T_j)$ is an ordered list L_j of $|T_j|$ coefficients. The list is ordered according to the order that GreedyAbs discards the coefficients $c_i \in T_j$. More specifically, each element of the list is a tuple $(error, index)$ that indicates the maximum incurred absolute error when this node is discarded and the index of the discarded node in the error tree. This procedure of locally executing GreedyAbs on a sub-tree, is carried out in parallel for all base sub-trees.

When this stage of parallel GreedyAbs runs is over, we collect, sort error-wise and merge all the outputs from all the base sub-trees (i.e., $\forall T_j \in$ base sub-trees, sort and merge L_j), thus obtaining a globally ordered list. By keeping the $B - |C_{root}|$ elements of the list with the highest error, let us call them C_{base}, we form the final synopsis: $C_{root} \cup C_{base}$.

We have already mentioned that, in GreedyAbs, the error does not change monotonically when a coefficient is removed. For example, consider the key-values $KV_1 = (50, 8)$, $KV_2 = (42, 10)$ discarded in respective order. At the final global order, we expect KV_1 prior to KV_2, since node 8 was discarded first. However, as the error of KV_2 is smaller than the one of KV_1, the sorting will place KV_1 after KV_2. In order to avoid this unwanted behavior, we modify the logic according to which L_j lists are emitted to the sorting and merging mechanism. Instead of emitting a single node per key-value, we emit a list of nodes. While the error incurred by the deletion of a node is smaller or equal to the maximum so far error, we append the node to be discarded to a list. Otherwise, we emit the key-value and create a new one with an empty list. Consider the key-values KV_1, KV_2 of the previous example. Instead of emitting both of them, we emit $(50, [8, 10])$. Apart from correctness, in this way we also achieve better I/O efficiency, since the $|T_j|$ coefficients are emitted in batches and not one at a time.

This scheme, where lists are emitted as values, can be further improved. Assume the error from discarding a node is 132.44 and the error for the next node is 132.45. As the absolute difference is very small, we may decide that we can tolerate this error and, for I/O efficiency, we do not emit two different key-values. In general, we consider a user-specific parameter e_b and partition the domain of errors to buckets of width e_b. The algorithm is then modified as to emit a new key-value only when an error from the next bucket appears. In Algorithm 3 we present the modified *discardNode* function that we use for GreedyAbs.

Algorithm 3 *discardNode*

Require: node index k, error e_k incurred by the deletion of node k, width e_b of the error bucket, list L of discarded nodes, the current max error *max_error*

1: $\hat{e} = \lfloor \frac{e_k}{e_b} \rfloor \cdot e_b$
2: **if** $\hat{e} \leq max\_error$ **then**
3: L.append(c_k)
4: **else**
5: emit $(\hat{e}, L)$;$max\_error = \hat{e}$;$L = []$
6: **end if**

So far, we have ignored the procedure that finds the appropriate nodes to be retained from the root sub-tree, assuming it is "magically" available. As we cannot compute a-priori which these nodes are, we need to *speculatively* create the synopses for different C_{root} sets and finally retain the one that produces the best approximation. Let R denote the

size of the root sub-tree. Since we do not know the number of nodes that should be retained from the root sub-tree, we should compute the synopsis for at least $min\{R, B\}+1$ different C_{root} sets, with each candidate C_{root} having different size: The empty set, as we may keep none of these nodes, keep only 1 node, keep 2 nodes, etc, until we examine the case where $min\{R, B\}$ nodes are kept. In order to find $min\{R, B\}+1$ candidate C_{root} sets, we run GreedyAbs on the root sub-tree. The intuition behind this choice is that, since only the root sub-tree is considered known at this stage, we should try to optimize the local problem and each time discard the node that incurs the minimum error. The candidate C_{root} sets are generated by the *genRootSets* function presented in Algorithm 4.

For example, we consider as root sub-tree the nodes $\{c_0, c_1, c_2, c_3\}$ of the error tree depicted in Figure 1. The run of GreedyAbs selects to discard the nodes according to the following order: $[c_1, c_3, c_2, c_0]$. Thus, the candidate C_{root} sets are the following 5: $\{c_1, c_3, c_2, c_0\}$,$\{c_3, c_2, c_0\}$,$\{c_2, c_0\}$,$\{c_0\}$,$\{\}$

Algorithm 4 *genRootSets*

Require: root sub-tree,B
1: $L_{root} = GreedyAbs$(root sub-tree)
2: $C = \{\{\}\}$
3: $lastIndex = L_{root}.size$
4: **for** $(i = lastIndex; i > lastIndex - B; i = i - 1)$ **do**
5: $\quad C_{root,i} = \{L_{root}[i], .., L_{root}[lastIndex]\}$
6: $\quad C = C \cup \{C_{root,i}\}$
7: **end for**
8: **return** C

In order to execute our algorithm in a distributed environment, we use the following scheme. Initially, we run GreedyAbs, in a centralized fashion, on the root sub-tree and we get as output a set C with all the candidate C_{root} sets. Since the root sub-tree can be exponentially smaller than the original dataset, its processing on a single machine can be done without compromising performance. For the distributed processing of the remaining dataset, we consider an architecture of two *levels* of workers. Each base sub-tree T_j is assigned to a different level-1 worker. Level-1 worker j runs GreedyAbs over its local data for every $C_{root} \in C$ that affects the incoming error to T_j. For every list of nodes that the GreedyAbs discards, the worker emits a key-value, in the form $([C_{root}, error], List)$ to the next level of workers. Therefore, apart from the maximum incurred error, the key contains information on which nodes are retained from the root sub-tree, for this run of GreedyAbs.

From the description of the algorithm above, it is derived that each worker runs GreedyAbs for all $C_{root} \in C$ that affect the incoming error. As we emit every node for each run of GreedyAbs, we observe that each initial coefficient is emitted $O(min\{R, B\}+1)$ times. This quantity can be multiple times the size of the initial dataset and leads to significant I/O overhead.

In order to avoid the excessive I/O, instead of emitting the list of the actual coefficients, we emit a histogram. For each emitted key-value, we keep the key as is and as value we emit the number of nodes that are discarded producing a maximum error that falls in a specific error bucket. For example, instead of emitting the key-value $(50, [8, 10])$ as in the previous example, we now emit $(50, 2)$, where 2 is the length of the list $[8, 10]$. This way, we emit sizeOf(int) bytes instead of the actual size of the list. We call *ErrHistGreedyAbs* the modified version of GreedyAbs that emits error histograms.

With the use of an appropriate partitioning scheme, all the emitted key-values that refer to the same C_{root} end up at the same level-2 worker. Key-values are sorted and merged according to the error information that is stored in the key, with the node yielding the highest error expected to appear first. When sorting and merging are over, there is a global ordering of all the nodes of the dataset for a particular C_{root}. By examining the first $B+1$ key-values, the B most important nodes are identified and the worker extracts the best achieved maximum absolute error. When the processing is over, each level-2 worker has computed the best error for the C_{root} sets it was assigned. We call *combineResults* the described procedure and we formally describe it in Algorithm 5. In a final step, all the level-2 workers send their results to a single process that decides the most accurate synopsis.

Algorithm 5 combineResults

Require: space budget B
Require: the set C' of C_{root} sets the worker is responsible for
Require: a list $L_{C_{root}} = [L_{1,C_{root}}, L_{2,C_{root}}, ..], \forall C_{root} \in C'$, where $L_{j,C_{root}}$ is the emitted key-values for C_{root} from sub-tree T_j.
1: $minError = infinity, bestCroot = \{\}$
2: **for all** $C_{root} \in C'$ **do**
3: $\quad L = merge(L_{C_{root}})$
4: $\quad$ **if** $L[B - |C_{root}|].error < minError$ **then**
5: $\quad\quad minError = L[B - |C_{root}|].error$
6: $\quad\quad bestCroot = C_{root}$
7: $\quad$ **end if**
8: **end for**
9: **return** $(minError, bestCroot)$

With the level-1 workers emitting histograms, when the job is over, we do not know which coefficients to choose for the synopsis but only which C_{root} we should use and what is the best error that can be achieved. By knowing the nodes that should be retained from the root sub-tree, we initiate a second job that computes and constructs the synopsis. In this second job, each level-1 worker runs exactly once GreedyAbs over its local data, only for the C_{root} that yields the best quality synopsis. This time, the workers do not emit histograms but the actually removed nodes. Furthermore, as at this time the final error ϵ^* is known, the workers do not have to emit all the nodes of their local sub-tree but only the ones that their removal incurs an error higher than ϵ^*. We call our algorithm DGreedyAbs and we formally describe it in Algorithm 6.

5.3 Complexity Analysis

The worst-case cost of GreedyAbs is $O\left(Nlog^2N\right)$ [22]. However, as we will see in the experimental Section, the algorithm's complexity is linear in practice and also leads to linear behavior of our algorithm. Assume $R = min\{R, B\}$. The run of GreedyAbs on the root sub-tree produces a powerset C, such that:
$|C| = R + 1 \wedge \forall C_{root,i} \in C, 0 \leq |C_{root,i}| \leq R$
$\wedge |C_{root,i}| \neq |C_{root,j}| \forall i, j \in [0, R-1], i \neq j$
That means that every two different C_{root} sets differ from each other by at least one coefficient and since $|C| = R+1$, eventually all the coefficients of the root sub-tree will be selected, one at a time. In this way, we have a different incoming error to T_k for every node in $path_k$. Since

Algorithm 6 DGreedyAbs

Require: error tree, space budget B
1: C = $genRootSets$(root sub-tree,B)
2: Assign base sub-trees to level-1 workers
3: **for all** $T_j \in$ base sub-trees in parallel **do**
4: **for all** $C_{root} \in C$ that affect incoming error in T_j **do**
5: e_{in} = incoming error in T_j from C_{root}
6: $err_i = err_i + \delta_j e_{in}, \forall err_i \in T_j$
7: $ErrHistGreedyAbs\left(T_j\right)$
8: **end for**
9: **end for**
10: $minError = infinity, bestCroot = \{\}$
11: **for all** w $\in$ level-2 workers in parallel **do**
12: Let $C_w \subseteq C$, the C_{root} sets that w is responsible for.
13: $(error, Croot) = combineResults\left(B, C_w, L_{C_w}\right)$
14: **if** $error < minError$ **then**
15: $minError = error$
16: $bestCroot = Croot$
17: **end if**
18: **end for**
19: **for all** $T_j \in$ base sub-trees in parallel **do**
20: e_{in} = incoming error in T_j from $bestCroot$
21: $err_i = err_i + \delta_j e_{in}, \forall err_i \in T_j$
22: $L_j = GreedyAbs\left(T_j\right)$ {emit only nodes yielding error higher than $minError$}
23: **end for**
24: $L = merge(L_j$ lists)
25: **return** $\{L\left[0\right], .., L\left[B - |bestCroot|\right]\}$

the length of the path of each base sub-tree to the root is $logR + 1$, each mapper runs GreedyAbs exactly $logR + 2$ times; one for every node in $path_k$ and one for the empty set. Thus, the running-time complexity for the level-1 workers is $O\left(\left(logR + 2\right) Nlog^2 N\right) = O\left(logRNlog^2 N\right)$. For a fixed size of base sub-trees, it holds that $R \sim N$, that is for a new $N' = 2^k N$ the new size of the root sub-tree will be $R' = 2^k R$ and so our solution is linearly scalable, as there is only a constant overhead in the running-time complexity of a level-1 worker. The running-time complexity of a level-2 worker is $O\left(N\right)$, as it only merges the already sorted outputs from the previous stage of execution.

An interesting question relates the size of the base and root sub-trees. Does the size of the sub-trees affect performance? Which is a "good" size for them? We study again the running-time complexity for a level-1 worker. Let us denote with S the size of a base sub-tree. It holds that $N = R + R \cdot S$. Consider now the same dataset N, but differently partitioned, so as $R' = 2^k R$. Since the dataset is the same, we have:

$$R + R \cdot S = R' + R' \cdot S' \Rightarrow S' = \frac{S+1}{2^k} - 1$$

So, the complexity becomes:

$$O\left(logR'S'log^2 S'\right) = O\left(\left(logR + k\right)\left(\frac{S+1}{2^k} - 1\right) log^2\left(\frac{S+1}{2^k} - 1\right)\right) = O\left(logRSlog^2 S\right) \quad (9)$$

Next, we examine the communication complexity between level-1 and level-2 workers. In the worst case, where each node is discarded with a different error, S key-values are emitted for each $C_{root} \in C$. As $|C| = R+1$, there are $O\left(RS\right)$ emitted key-values. In practice, the communication is much lower, as with the described algorithm many discarded nodes are compacted to a single emitted key-value.

Finally, we compute the impact that the size of the sub-trees has on communication. As before, we consider a different partitioning with R', S'. It holds that:

$$O\left(R'S'\right) = O\left(2^k R\left(\frac{S+1}{2^k} - 1\right)\right) = O\left(RS\right)$$

We see that the size of the base sub-trees does not asymptotically affect neither running-time nor communication.

5.4 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 [31]. As in earlier approaches ([12], [13]), 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 is kept lower than a feasible bound.

For this problem, in [22] the GreedyRel algorithm is presented. GreedyRel follows the greedy paradigm introduced in Section 5.1, wherein, instead of using MA_k, it chooses to discard the coefficient with the minimum *maximum potential relative* error, defined as follows:

$$MR_k = max_{d_j \in leaves_k}\{\frac{|err_j - \delta_{jk} \cdot c_k|}{max\left(|d_j|, S\right)}\} \quad (10)$$

Nevertheless, the four error quantities of Equation 8 cannot be used for the calculation or update of the MR_k. The reason is the denominator in Equation 10, which implies that the effect a coefficient c_k is different in the signed relative error of different data values.

In order to provide a scalable solution to this problem, we use a similar approach with that of DGreedyAbs, but instead of using GreedyAbs at the workers, we use GreedyRel.

6. EXPERIMENTAL EVALUATION

In the experimental Section we focus on evaluating the utility of the proposed distributed algorithms with respect to their efficiency (measuring creation time and approximation accuracy), data- and resource-based scalability and discuss parameters that affect their performance. All algorithms are implemented in Java. The distributed algorithms were developed using the MapReduce programming model.

Datasets. The experiments were conducted using both synthetic and real datasets. Synthetic data (SYN) allows easy testing over different data distributions and value ranges. Distributions utilized are uniform and zipfian (with exponents 0.7 and 1.5). Data values lie between $[0, M]$, with $M \in \{1K, 100K, 1000K\}$. For the real-life datasets we utilize NYCT [2] and WD [1]. NYCT describes taxi trips in the New York City in 2013 containing records for the trip time in seconds. WD consists of observations on wind direction that were captured by sensors in the U.S.A. during the hurricanes Ike, Bill, Bertha and Katrina. Wind direction is reported in azimuth degrees. Both datasets were partitioned

Table 3: Characteristics of NYCT and WD datasets

Name	#Records	Avg	Stdv	Max
NYCT2M	2M	672	483	10800
NYCT4M	4M	511	519.5	10800
NYCT8M	8M	255	646.6	10800
NYCT16M	16M	127	745	10800
NYCT32M	32M	63	3566.3	4293410
NYCT64M	64M	31	25410.3	4294966
WD2M	2M	121	119.7	655
WD4M	4M	122	119.9	655
WD8M	8M	138	119.4	655
WD16M	16M	127	118.8	655

in order to test scalability over different sizes. The smallest partition comprised the first $2M$ records, while each subsequent partition was twice as big as the previous one. Table 3 gives an overview of all real datasets used.

Platform setup. For our deployment platform, we used a Hadoop 2.6.0 cluster of 9 machines, each featuring eight Intel(R) Xeon(R) CPU E5405 @ 2.00GHz cores and 8 GB of main memory. One machine was used as the master node with the remaining ones setup as slaves. Each slave was allowed to run simultaneously up to 5 map tasks and 2 reduce tasks. Each of these tasks was assigned 1 physical core and 1 GB of main memory. For all the remaining properties, we kept the default Hadoop configuration.

DGreedyAbs and DIndirectHaar were deployed on the described platform. For the centralized algorithms, i.e., GreedyAbs and IndirectHaar, we used one machine with the same configuration as the ones of our Hadoop cluster.

6.1 Scalability

In this subsection, we assess the scalability of the algorithms with respect to the sub-tree size, the budget space for the synopsis B, the number of datapoints N and the number of tasks running in parallel. For the scalability experiments of this Section, we use a synthetic dataset of uniformly distributed data values in the range of $[0, 1K]$.

Varying sub-tree size. In the described distributed algorithms and according to the proposed partitioning schemes, each worker is assigned to process a sub-tree of the error tree. A first question we need to answer is how the size of the local sub-problems affects the overall performance of the algorithm. Figure 5a presents the running-time of the DIndirectHaar and DGreedyAbs algorithms when different sub-tree sizes are used. We examine values from $131K$ (2^{17}) to $1M$ (2^{20}) nodes per sub-tree. Values smaller than $131K$ are not appropriate as the resulting partitions are too small and incur a very high overhead to Hadoop. Similarly, values bigger than $1M$ do not fit in our mapper's main memory, degrading performance. For this experiment we use a dataset of $N = 17M$ and a budget space $B = N/8$. Figure 5a shows that the size of the sub-trees does not significantly affect the running-time of the job. This observation verifies the theoretical complexity for both algorithms. For the remainder of the experiments we consider a sub-tree size of $1M$ nodes for both algorithms.

Varying budget space. In the next experiment we examine the scalability to the budget space B. We run both DGreedyAbs and DIndirectHaar for a data size $N = 17M$ and we vary B from $N/64$ to $N/8$. Results are shown in Figure 5b. The running-time of DGreedyAbs is not considerably affected by the size of the synopsis. This is not always true for the DIndirectHaar algorithm. As we can see for $B = 0.5M$ for DIndirectHaar, the running-time may decrease as B is increased. This is because higher B-values lead to tighter errors and thus higher convergence rate for the algorithm. As the running-time of our algorithms is not significantly affected by B, for all subsequent experiments, we consider $B = N/8$.

Varying data size and number of parallel tasks. Figures 5c, 5d show the scalability with respect to N and the number of tasks running in parallel for DGreedyAbs and DIndirectHaar respectively. We vary the datasize from 2M to 537M datapoints for all the algorithms and the number of parallel map tasks from 10 to 40. For DGreedyAbs, we fix the number of reducers to four, as they do not have a high impact on performance and for DIndirectHaar we use only one reducer. We also compare both algorithms with the corresponding centralized implementations in order to assess the difference in performance. For DIndirectHaar, we set the parameter δ to 50, as for this value it achieves the best running-time results. We also note that the y-axis follows a logarithmic scale in these Figures.

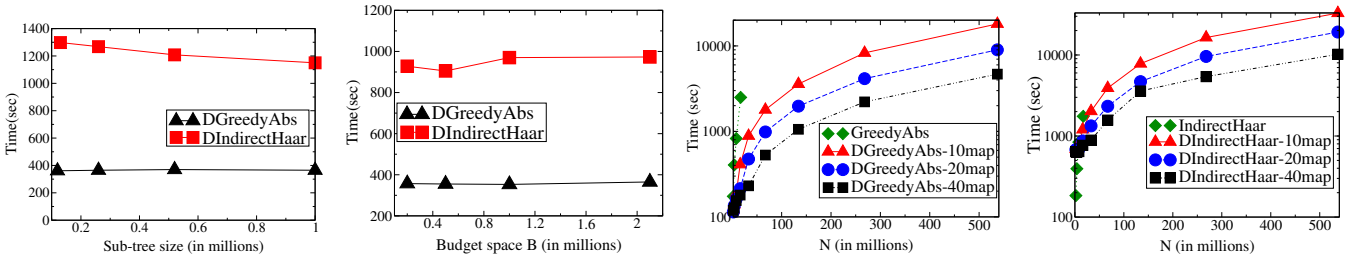
Both algorithms scale linearly with the dataset size. The running-time is almost constant at first, when all data can be processed fully in parallel, and is linearly growing as the cluster is fully utilized and more tasks need to be serialized for execution. Linear scalability is also observed with the number of parallel running tasks. By halving the capacity of the cluster, running-time is doubled for DGreedyAbs and is increased by a factor of 1.7 on average for DIndirectHaar.

Compared to the centralized GreedyAbs, DGreedyAbs is 7.4× faster for a dataset of 17M datapoints. For sizes greater than $17M$ points, neither GreedyAbs nor IndirectHaar could run, as their execution demanded more main memory than the available 8GB. In Figure 5d, we see that IndirectHaar is faster than DIndirectHaar when the dataset size is small or there are few parallel running tasks. That is because the centralized implementation loads the whole dataset in memory and the multiple jobs required by IndirectHaar do not need to perform I/O operations. Therefore, in order to get benefit from DIndirectHaar, we need large datasets and compute-intensive jobs, where parallelization can be exploited. The degree of compute-intensity of a job is dependent on the dataset. We remind that the complexity of IndirectHaar (DIndirectHaar) is $O\left(\left(\frac{\varepsilon}{\delta}\right)^2 N\left(log\epsilon^* + logN\right)\right)$. For two same-sized datasets, running-time is determined by the term $\left(\frac{\varepsilon}{\delta}\right)^2$, which is dataset-specific.

6.2 Dataset impact

In this subsection, we use synthetic datasets to evaluate the impact of different data distributions and value ranges on both running-time and approximation quality. For all the experiments in this subsection, we use datasets of size $N = 17M$ and synopsis size $B = N/8$.

Varying distribution and δ. As the parameter δ of DIndirectHaar provides a "knob" for tuning the tradeoff between resource requirements and solution quality, in Figure 6 we show the impact of data distribution on DIndirectHaar when different δ-values are used. The main observation is that biased distributions favor both the synopsis construction time and the approximation quality [28]. In Figure 6a, we see that for the Zipf-0.7 distribution and for all δ-values, the algorithm is about 25% faster compared to the Uniform



(a) Scalability with the sub-tree size.

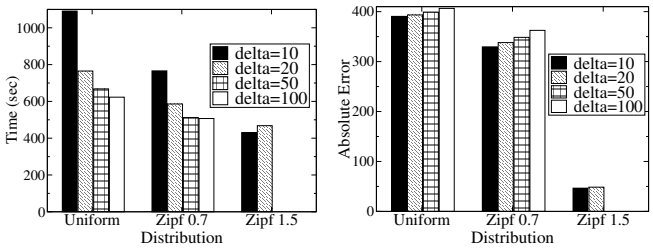
(b) Scalability with the budget space

(c) Scalability of DGreedyAbs with N and cluster size.

(d) Scalability of DIndirectHaar with N and cluster size.

Figure 5: Scalability with the sub-tree size, budget space, dataset size and number of parallel tasks

distribution. Furthermore, the run for the Zipf-1.5 distribution outperforms the one for Zipf-0.7 by 45% when $\delta = 10$ and 20% when $\delta = 20$. Accordingly, in Figure 6b we see that when the Zipf-1.5 distribution is the case, the maximum absolute error is 8.4 times smaller than the one achieved for the Uniform data.



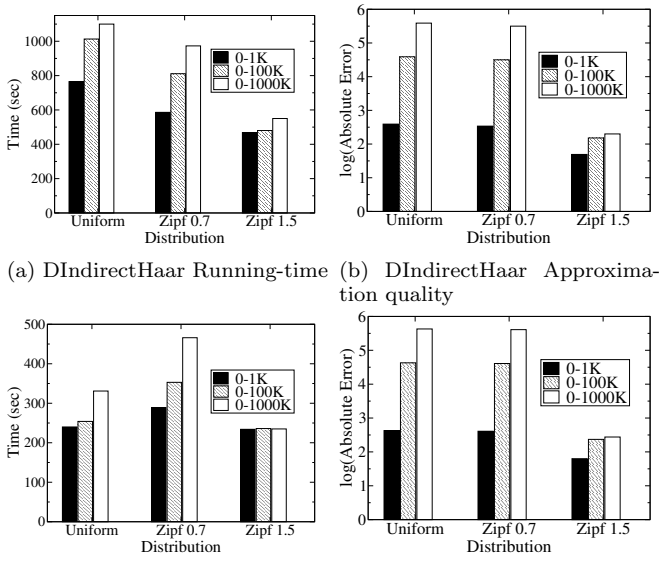
(a) Running-time

(b) Approximation quality

Figure 6: Impact of data distribution and δ on the performance and approximation quality of DIndirectHaar.

We also see that usually the smaller δ is, the higher is the running-time of the algorithm and the better the approximation quality, since more candidate values are examined for the incoming values and the wavelet coefficients. For values of δ equal to 50 or 100 we see that the algorithm reaches its lower bound of execution time on this data and thus, higher values for δ do not affect performance. For the Zipf-1.5, we see that the run for $\delta = 10$ outperforms the one for $\delta = 20$. As we get more approximate results for higher values of δ, DIndirectHaar requires more jobs to converge and provide the final answer. Moreover, the algorithm could not run for Zipf-1.5 and $\delta = 50, 100$ as these values were higher than the space they need to quantize.

Varying distribution and value ranges. Figure 7 shows how both DIndirectHaar and DGreedyAbs are affected by the range of dataset values. For the DIndirectHaar, we use $\delta = 20$, as it could not run for the Zipf-1.5 distribution when higher values for δ were used. For both algorithms and for all data distributions, we observe that datasets with wide ranges to select values from, lead to higher running-time and maximum absolute error. Intuitively, the larger the range of data values, the more likely are discontinuities present and thus, more coefficients are needed for an accurate synopsis. Since we keep the budget space fixed to $N/8$, we expect the maximum error to increase with the value range. In Figures 7b and 7d we observe that for the Uniform and the Zipf-0.7 distributions, an increase of an order of magnitude in the range of values is reflected to a corresponding increase to the maximum absolute error.



(a) DIndirectHaar Running-time

(b) DIndirectHaar Approximation quality

(c) DGreedyAbs running-time

(d) DGreedyAbs Approximation quality

Figure 7: Impact of different data value ranges and distributions on the performance and approximation quality.

However, as we can see from the example of the Zipf-1.5 distribution for both algorithms, both the achieved running-time and error for very biased datasets are very robust to changes in the range of data values. Moreover, in Figures 7a and 7c we observe that, for all distributions, running-time is more affected by the range of data values for DIndirectHaar than for DGreedyAbs. For example, the running-time of DIndirectHaar for the range $[0, 100K]$ is 25% higher than for $[0, 1K]$ for both Uniform and Zipf-0.7 distributions. The corresponding numbers for DGreedyAbs are 5% for the Uniform and 15% for the Zipf-0.7.

In Figure 7c, we notice a counter-intuitive result: The best running-time is achieved for the Uniform distribution. This is due to the I/O cost between the map and the reduce phase. In the case of the Uniform distribution, when the root of the error tree is not retained in the C_{root}, the first deletion of a node causes a large absolute error X. This first deleted node c_k happens to be located in the last level of the error tree and thus it affects only the nodes in $path_k$, i.e., $logS$ nodes. The algorithm continues by discarding other nodes. As the deletion of the majority of the nodes causes an error $e < X$, in this case, almost the whole sub-tree is emitted as a single key-value achieving high I/O-efficiency.

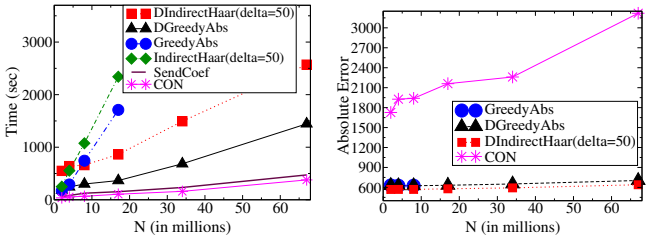
6.3 Direct Comparison

In this subsection we compare DGreedyAbs and DIndirectHaar with each other, as well as with their centralized counterparts using real-life datasets. Furthermore, we compare them against algorithms that construct a conventional synopsis (i.e., L_2-optimal). As these algorithms are less compute-intensive, we want to investigate the tradeoffs in running-time and produced maximum error. For constructing the conventional synopsis we implement CON, a parallel algorithm that uses the partitioning scheme described in Section 4 and retains the B largest coefficients in absolute normalized value. We also use Send-Coef [21], which computes in parallel the conventional synopsis by using a different partitioning. Both CON and Send-Coef are described in detail in Appendix A. For the approximation quality experiments, we do not include IndirectHaar and Send-Coef, as they theoretically achieve exactly the same results with DIndirectHaar and CON respectively.

NYCT dataset. In Figure 8, we present the results for the NYCT dataset. For the IndirectHaar and the DIndirectHaar we set $\delta = 50$, as for this value they achieve the best execution times. The construction of an accurate synopsis for this dataset is a difficult task to accomplish as it contains values of high magnitude and variance. In Table 3, we see that the NYCT64M dataset for example, has a maximum value 4294966 and standard deviation 25410.3. The difficulty in approximating NYCT is illustrated in Figure 8b, where the maximum absolute error for $B = N/8$ is more than 550 for all data sizes and algorithms. However, the most important deduced result from Figure 8b is that DGreedyAbs does not compromise approximation quality for the performance gain it offers and achieves the same maximum absolute error with its centralized counterpart. The synopsis produced by DGreedyAbs is 3 to 4.5 times more accurate, with respect to *max_abs*, than the conventional one.

Figure 8a presents the running-time results for the same dataset. With the maximum absolute error over 570 for all data sizes, the multiplicative factor $\left(\frac{\varepsilon}{\delta}\right)^2$ of the complexity formula of IndirectHaar and DIndirectHaarand is equal to 121. As such, for this dataset, the execution of the DP algorithms is very compute-intensive. Nevertheless, as we have already mentioned, datasets that demand intensive computations favor DIndirectHaar over IndirectHaar which is 2.7 times slower for a 17M dataset. As seen in Section 6.2, the execution of DGreedyAbs is more robust to different datasets and thus, its running-time is not significantly affected by the data distribution. DGreedyAbs is the most time-efficient algorithm, targeting maximum absolute error, being 5× faster than GreedyAbs for a 17M dataset and 1.8×–2.9× faster than DIndirectHaar for all data sizes. As the conventional synopsis is easier to be computed, we observe CON to be 4.2× and Send-Coef 2.8× faster on average than DGreedyAbs. CON outperforms Send-Coef, due to its partitioning scheme that preserves sub-tree locality.

WD dataset. Figure 9 shows the results for the WD dataset. In this experiment, we run IndirectHaar and DIndirectHaar with $\delta = 20$, as they could not run for larger values of δ. The values of this data set do not present large discontinuities and can thus be more easily approximated. This is verified in Figure 9b, where the maximum absolute errors are about 5 times smaller than the corresponding ones for the NYCT dataset. Regarding approximation quality, DGreedyAbs achieves again the same maximum absolute error

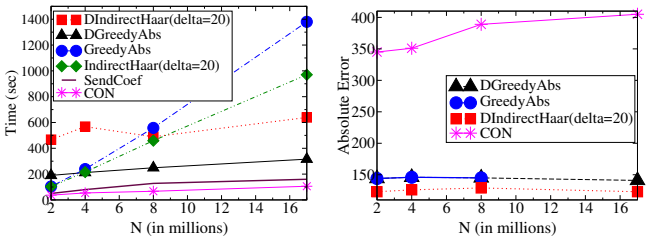


(a) Running-time (b) Approximation quality

Figure 8: Comparison for the NYCT dataset, $B = N/8$.

with GreedyAbs and on average 2.6× smaller than CON.

With a maximum absolute error about 125, the factor $\left(\frac{\varepsilon}{\delta}\right)^2$ is only 36 and thus, the execution of the DP algorithms demands fewer computations than for the NYCT dataset. Similarly to Section 6.1, in Figure 9a we see that IndirectHaar outperforms DIndirectHaar for data sizes up to 8M datapoints. We also observe that for this dataset and for all data sizes, IndirectHaar outperforms GreedyAbs and CON outperforms Send-Coef. Still, the most efficient algorithm, that targets the minimization of maximum error metrics, is DGreedyAbs as it outperforms GreedyAbs by a factor of 4.4 for a $17M$ dataset and achieves half the running-time of DIndirectHaar for all data sizes.



(a) Running-time (b) Approximation quality

Figure 9: Comparison for the WD dataset, $B = N/8$.

7. CONCLUSIONS

In this paper, we have examined the problem of wavelet thresholding aiming at the minimization of maximum error metrics. Having established that the existing approaches do not scale for big datasets, we focused on designing algorithms with linear scalability over scale-out infrastructures. We first presented a novel technique that allows the parallel decomposition of an error tree and can be used to support all the existing DP algorithms for the problem. In order to demonstrate the power of the proposed technique, we applied it on IndirectHaar thus implementing the DIndirectHaar algorithm. Our results show that DIndirectHaar scales linearly to data sizes that IndirectHaar is incapable of processing. Moreover, in order to further improve the running-time for the synopsis construction, we proposed DGreedyAbs, a new heuristic-based algorithm based on GreedyAbs. DGreedyAbs is over 7 times faster than GreedyAbs for a 17M dataset and 2–4 times faster than DIndirectHaar for all data sizes. Despite its efficient execution time, DGreedyAbs does not compromise quality, as in all experiments it achieves the same approximation results with its centralized counterpart.

8. ACKNOWLEDGEMENTS

The research leading to these results has been partially supported by the European Commission, in terms of the ASAP FP7 ICT Project under grant agreement no 619706.

9. REFERENCES

[1] Linked sensor data. https://wiki.knoesis.org/index.php/SSW_Datasets.

[2] Nyc taxi trip data 2013. https://archive.org/details/nycTaxiTripData2013.

[3] The Internet of Things, Hadoop, and the Big Data Approach. http://data-informed.com/internet-things-hadoop-big-data-approach/.

[4] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join synopses for approximate query answering. In *ACM SIGMOD Record*, volume 28, pages 275–286. ACM, 1999.

[5] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 29–42. ACM, 2013.

[6] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 20–29. ACM, 1996.

[7] L. Amsaleg, M. J. Franklin, A. Tomasic, and T. Urhan. Improving responsiveness for wide-area data access. In *IEEE Data Engineering Bulletin*. Citeseer, 1997.

[8] P. Cao and Z. Wang. Efficient top-k query calculation in distributed networks. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 206–215. ACM, 2004.

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

[10] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 4(1–3):1–294, 2012.

[11] G. Cormode, M. Garofalakis, and D. Sacharidis. Fast approximate wavelet tracking on streams. In *Advances in Database Technology-EDBT 2006*, pages 4–22. Springer, 2006.

[12] M. Garofalakis and P. B. Gibbons. Wavelet synopses with error guarantees. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 476–487. ACM, 2002.

[13] M. Garofalakis and A. Kumar. Deterministic wavelet thresholding for maximum-error metrics. In *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 166–176. ACM, 2004.

[14] P. B. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. In *ACM SIGMOD Record*, volume 27, pages 331–342. ACM, 1998.

[15] P. B. Gibbons, Y. Matias, and V. Poosala. Fast incremental maintenance of approximate histograms. In *VLDB*, volume 97, pages 466–475, 1997.

[16] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *VLDB*, volume 1, pages 79–88, 2001.

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

[18] S. Guha. Space efficiency in synopsis construction algorithms. In *Proceedings of the 31st international conference on Very large data bases*, pages 409–420. VLDB Endowment, 2005.

[19] Y. E. Ioannidis and V. Poosala. Histogram-based approximation of set-valued query-answers. In *VLDB*, volume 99, pages 174–185, 1999.

[20] H. V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. C. Sevcik, and T. Suel. Optimal histograms with quality guarantees. In *VLDB*, volume 98, pages 275–286, 1998.

[21] J. Jestes, K. Yi, and F. Li. Building wavelet histograms on large data in mapreduce. *Proceedings of the VLDB Endowment*, 5(2):109–120, 2011.

[22] P. Karras and N. Mamoulis. One-pass wavelet synopses for maximum-error metrics. In *Proceedings of the 31st international conference on Very large data bases*, pages 421–432. VLDB Endowment, 2005.

[23] P. Karras and N. Mamoulis. The haar+ tree: a refined synopsis data structure. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 436–445. IEEE, 2007.

[24] P. Karras, D. Sacharidis, and N. Mamoulis. Exploiting duality in summarization with deterministic guarantees. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 380–389. ACM, 2007.

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

[26] Y. Matias, J. S. Vitter, and M. Wang. Wavelet-based histograms for selectivity estimation. In *ACM SIGMOD Record*, volume 27, pages 448–459. ACM, 1998.

[27] S. Muthukrishnan. Subquadratic algorithms for workload-aware haar wavelet synopses. In *FSTTCS 2005: Foundations of Software Technology and Theoretical Computer Science*, pages 285–296. Springer, 2005.

[28] C. Pang, Q. Zhang, D. Hansen, and A. Maeder. Unrestricted wavelet synopses under maximum error bound. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 732–743. ACM, 2009.

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

[30] I. Trummer and C. Koch. An incremental anytime algorithm for multi-objective query optimization. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1941–1953. ACM, 2015.

[31] J. S. Vitter and M. Wang. Approximate computation of multidimensional aggregates of sparse data using wavelets. In *ACM SIGMOD Record*, volume 28, pages

193–204. ACM, 1999.

APPENDIX

A. CONSTRUCTING THE CONVENTIONAL SYNOPSIS IN PARALLEL

In this Appendix, we describe in detail the CON and Send-Coef [21] algorithms used in Section 6.3 for computing the conventional wavelet synopsis. Moreover, we describe the Send-V and H-WTopk algorithms that also compute the conventional synopsis in parallel and are presented in [21]. All the algorithms in [21] compute wavelet synopses over histograms. Thus, in order to compare them against our algorithms, we first modify them not to compute histograms and perform the wavelet transform directly on the input data. For all the descriptions that follow, we denote N as the dataset size, m the number of map tasks, S the input size of a map task and R the size of the root sub-tree in datapoints.

A.1 The CON Algorithm

In order to compute the conventional synopsis in parallel, we partition the data as described in Section 4 (see Figure 3). Each mapper reads a portion of the input in the size of a power of two and locally constructs the corresponding sub-tree by pairwise averaging and differencing coefficients, as explained in Section 2. As the wavelet transform is of linear complexity and the mapper computes the coefficients only for its local data, the computational complexity of each map task is $O(S)$. After the construction of the sub-trees is over, each mapper emits all the computed coefficients to the reduce stage. Thus, the communication between the map and reduce phase is $O(N)$. The reducer reads all the coefficients that are computed in the map phase and inserts them in a priority queue, where only the B largest ones in absolute normalized value are retained. It also computes the wavelet coefficients of the root sub-tree and inserts them in the queue as well. When this process is over, there are B coefficients in the queue which comprise the conventional synopsis.

A.2 Send-V

The simplest algorithm presented in [21] for the computation of a conventional synopsis is Send-V. The Send-V algorithm computes a histogram in the map phase of the job. The reducer centrally computes the wavelet coefficients and retains the B largest ones. As the histogram computation is not required in our case, Send-V is, in effect, a sequential algorithm, where the reducer reads and centrally computes the wavelet transform for all the input data.

A.3 Send-Coef

Send-Coef is based on another method to compute wavelet coefficients, that is used especially in streaming settings. This method uses the wavelet basis vectors. The first wavelet basis vector is $\psi_0 = [1,..,1]/\sqrt{N}$, where $[1,..,1]$ is a vector of ones. To define the other $N-1$ basis vectors, we first introduce, for $j = 1,..,logN$ and $k = 0,..,2^j - 1$, the vector:

$$\phi_{j,k}(l) = \begin{cases} 1 & k\frac{N}{2^j} + 1 \leq l \leq k\frac{N}{2^j} + \frac{N}{2^j} \\ 0 & elsewhere \end{cases} \quad (11)$$

For $j = 1,..,logN - 1$ and $k = 0,..,2^j - 1$, we define the i-th wavelet basis vector for $i = 2^j + k + 1$ as $\psi_i = (-\phi_{j+1,2k} + \phi_{j+1,2k+1})/\sqrt{\frac{N}{2^j}}$, where $\sqrt{\frac{N}{2^j}}$ is a scaling factor. If A is the data vector, the wavelet coefficients are the dot product of A with these wavelet basis vectors, i.e., $w_i = \langle A, \psi_i \rangle$, for $i = 1,..,N$.

The distributed computation of Send-Coef is based on the following observation:

$$w_i = \langle A, \psi_i \rangle = \sum_{j=1}^{m} \langle A_j, \psi_i \rangle,$$

where A_j is the j-th partition of the initial input data. Thus, every wavelet coefficient is a linear combination of the data values that belong to its sub-tree in the error tree.

Send-Coef partitions the data in a different way than the one proposed in Section 4. Each mapper takes up as many datapoints that fit as possible in a HDFS block size. The block size does not need to be aligned to a power of two. For every datapoint d_i, the mapper computes its contribution to the final value of every wavelet coefficient in $path_{d_i}$. Thus, a mapper partially computes all the coefficients along the path from its datapoints to the root of the error tree and thus sub-tree locality is not preserved. The reducer computes the final coefficients by aggregating the partially computed values and then retains the B largest ones in absolute normalized value. Algorithm 7 gives the pseudocode for the mappers of the Send-Coef algorithm.

As data locality is not preserved and for every data value we need to compute its contribution to $logN + 1$ nodes (the path to the root), the computational complexity of a mapper is $O(SlogN)$. Furthermore, every mapper emits $O(S(logN - logS))$ key-values to the reducer. By having m mappers, the whole communication cost is $O(mS(logN - logS)) = O(N(logN - logS))$. Compared to Send-Coef, our approach in Section A.1 achieves better computational complexity by a factor of $logN$ and communication cost by $logN - logS$.

Algorithm 7 *Send-CoefMapper*

Require: S: mapper input data
1: **for all** datapoints $d_i \in S$ **do**
2: **for all** error tree nodes $j \in path_{d_i}$ **do**
3: compute contribution $c_{i,j}$ of d_i to coefficient c_j
4: **if** c_j is fully computed **then** emit (j, c_j)
5: **end for**
6: **end for**
7: **for all** datapoints $d_i \in S$ **do**
8: **for all** error tree nodes $j \in path_{d_i}$ **do**
9: **if** c_j is partially computed **then** emit $(j, c_{i,j})$
10: **end for**
11: **end for**

A.4 H-WTopk

In order to reduce the communication cost between the map and the reduce phase, the H-WTopk algorithm is proposed in [21]. H-WTopk is based on the TPUT [8] algorithm for the distributed top-k problem. In contrast to TPUT, H-WTopk can handle both positive and negative values, as both are possible for a wavelet coefficient. The intuition behind the algorithm is to use a partial sum to prune items that cannot be in the top-k. Thus, a mapper does not need to

send all of its data to the reducer but only a set of candidate nodes, according to the local partial sums. The algorithm requires three communication rounds between the mappers and the reducer. For a coefficient x, $c(x)$ denotes its value and $c_j(x)$ its partially computed value at mapper j.

Round 1: Each mapper first emits the coefficients with the k highest and k lowest (i.e., most negative) values. For each coefficient x seen at the reducer, a lower bound $\tau(x)$ is computed on its total value's magnitude $|c(x)|$ (i.e., $|c(x)| \geq \tau(x)$), as follows. First, an upper bound $\tau^+(x)$ and a lower bound $\tau^-(x)$ are computed on its total value $c(x)$ (i.e., $\tau^-(x) \leq c(x) \leq \tau^+(x)$): If a mapper sends out the value of x, its exact value is added. Otherwise, for $\tau^+(x)$, the k-th highest value this mapper sends out is added and for $\tau^-(x)$ the k-th lowest value is added. Then we set $\tau(x) = 0$ if $\tau^+(x)$ and $\tau^-(x)$ have different signs and $\tau(x) = min\{|\tau^+(x)|, |\tau^-(x)|\}$ otherwise. Doing so ensures $\tau^-(x) \leq c(x) \leq \tau^+(x)$ and $|c(x)| \geq \tau(x)$. Now, the k-th largest $\tau(x)$, denoted as T_1, is used as a threshold for the magnitude of the top-k coefficients.

Round 2: A mapper j next emits all local coefficients x having $|c_j(x)| > T_1/m$. This ensures a coefficient in the true top-k in magnitude must be sent by at least one mapper after this round, because if a coefficient is not sent, its aggregated value's magnitude can be no higher than T_1.

Now, with more values available from each mapper, upper and lower bounds $\tau^+(x), \tau^-(x)$ are refined for each coefficient $x \in L$, where L is the set of coefficients ever received. If a mapper did not send the value for some x, T_1/m ($-T_1/m$) is now used for computing $\tau^+(x)$ ($\tau^-(x)$). This produces a new better threshold, T_2 (calculated in the same way as computing T_1 with improved $\tau(x)$'s), on the top-k coefficients' magnitude.

Next, coefficients are further pruned from L. For any $x \in L$ a new threshold $\tau'(x) = max\{|\tau^+(x)|, |\tau^-(x)|\}$ is computed based on refined upper and lower bounds $\tau^+(x), \tau^-(x)$. If $\tau'(x) < T_2$, coefficient x is deleted from L. The final top-k coefficients must be in the set L.

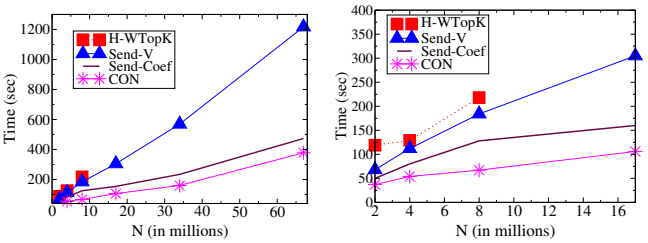
Round 3: Finally, the values of all coefficients in L are requested from each mapper. Then the aggregated values of exactly these coefficients are computed, and the k of largest magnitude among them are selected as the synopsis.

A.5 Experimental Evaluation of Algorithms for the Conventional Synopsis

For any given dataset, all four described algorithms produce exactly the same synopses. Thus, we do not need to compare them in terms of approximation quality. In this Section, we evaluate their performance with respect to running time. For the evaluation we use the NYCT and WD datasets over the same platform presented in Section 6. For all the experiments, we have configured a cluster for 20 available map and 1 reduce slots.

Figure 10 shows the running time results for both datasets when a synopsis of size $B = N/8$ is requested. Since Send-V ends up to be a sequential algorithm, it presents much worse running time performance than CON and Send-Coef for both examined datasets.

In Figure 10, we also observe that our algorithm (CON) is the most time-efficient for computing the conventional synopsis. The performance gain of CON stems from its locality-preserving partitioning, which results in less computational and communication complexity. CON is 1.5× faster on average than Send-Coef, that is the second most efficient algorithm, both for the NYCT and the WD datasets.



(a) NYCT (b) WD

Figure 10: Running time comparison for constructing a conventional synopsis with $B = N/8$.

For both datasets, we observe that despite the communication optimizations, H-WTopk presents the worst performance. Furthermore, for datasizes larger than 8 millions of datapoints, it runs out of memory. This is because of the selected synopsis size. H-WTopk can be very efficient if B is much smaller than the input size of the mapper. Otherwise, since it needs to emit the B largest and B smallest coefficients, it ends up emitting twice the input size. Furthermore, it also has the extra overhead of three MapReduce jobs. In [21], the wavelet transform was applied to a histogram and thus, data had been already compacted and smaller budget space was needed to achieve accurate results. The impact of B in the communication cost is discussed in [21], where the corresponding values were only chosen in the range $[10, 50]$.

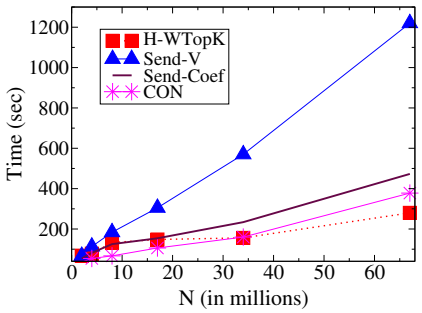


Figure 11: Running time results for the NYCT dataset and B=50.

Figure 11 shows the corresponding results for the NYCT dataset when a synopsis of stable size $B = 50$ is used. This figure verifies the results of [21]: H-WTopk dominates the other approaches only when B is very small and the dataset size large enough to not be affected by the overhead of the three MapReduce jobs. Thus, in our case, where the transform is applied directly on the data and not on a histogram, this algorithm is not of practical use as it is very difficult to construct a good quality synopsis with so few coefficients.