Scaling the Construction of Wavelet Synopses for Maximum Error Metrics

Ioannis Mytilinis, Dimitrios Tsoumakos, and Nectarios Koziris

Abstract—Modern analytics involve computations over enormous numbers of data records. The volume of data and the stringent response-time requirements place increasing emphasis on the efficiency of approximate query processing. A major challenge over the past years has been the construction of synopses that provide a deterministic quality guarantee, often expressed in terms of a maximum error metric. By approximating sharp discontinuities, wavelet decomposition has proved to be a very effective tool for data reduction. However, existing wavelet thresholding schemes that minimize maximum error metrics are constrained with impractical complexities for large datasets. Furthermore, they cannot efficiently handle the multi-dimensional version of the problem. In order to provide a practical solution, 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 algorithm, and (iii) two highly efficient distributed greedy algorithms that can deal with data of arbitrary dimensionality. 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.

Index Terms—Approximate Query Processing, Wavelet Synopses, Hadoop, Distributed Runtimes, Maximum Error Metrics

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’. Businesses, government organizations and digital infrastructures alike contribute to this Big Data reality. This abundance of datasets has in turn given rise to data-driven approaches in both academia and industry. Yet, there exist cases where existing data processing tools have become the bottleneck. When huge heterogeneous data is the case, even the fastest database systems can take hours or even days to answer the simplest of queries [1]. As data-driven discovery is often an *interactive and iterative process* [2], such response times are unacceptable to most users and applications. In order to meet the demands of interactive analytics, both commercial and open source systems continuously strive to provide lower response times through various techniques such as parallelism, indexing, materialization and query optimization.

Traditionally, most of these approaches try to better utilize available memory. However, keeping all useful data in main memory may not be an affordable or realistic option in the Big Data era. Even caching only a working-set of some GB does not do the trick. As analytics usually include streaming and iterative processes, loading different parts of the dataset each time incurs significant delays that may be not acceptable.

Approximate query processing has emerged as a viable alternative for dealing with the huge amount of data and the increasingly stringent response-time requirements [1]. 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 [3]. Moreover, approximate answers obtained from appropriate *synopses* of the data may be the only option when the base data is remote or unavailable [4].

To that end, several approximation techniques have been developed, including: sampling [1], [5], [6], histograms [7], [8], [9], wavelets [10], [11], [12], [13], [14] and sketches [15], [16]. Wavelet decomposition [17] provides a very effective data reduction tool, with applications in data mining [18], selectivity estimation [19], approximate and aggregate query processing of massive relational tables [10], [20] and data streams [21], [22]. 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 [10].

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 [17]. Still, the synopses produced by this method exhibit significant drawbacks [11], 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 [11], [23].

However, the existing algorithms that minimize maximum

• *Ioannis Mytilinis and Nectarios Koziris are with the Departmentof Electrical and Computer Engineering, National Technical University of Athens (NTUA), Greece.*
E-mail: {gmytil, nkoziris}@cslab.ece.ntua.gr.
• *Dimitrios Tsoumakos is with the Ionian University, Greece.*
Email: dtsouma@ionio.gr

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 [12], *GreedyAbs*, a heuristic-based solution is proposed. This algorithm is more time-efficient than the DP-based algorithms but at the cost of loosened quality guarantees. Yet, it cannot scale to Big Data either, as it follows a sequential path of execution that prevents a data-parallel approach.

Moreover, most approaches handle only one-dimensional data and cease to work or come at a prohibitive complexity when more dimensions are involved. Nevertheless, multidimensional datasets are a common case in real-world applications and such a limitation makes the use of wavelets impractical.

In this work, we present a general framework for parallelizing the existing DP algorithms to run over scalable, high-throughput modern platforms. Our framework covers both the cases of one- and multi-dimensional datasets. However, as DP algorithms are very costly, we also propose two heuristic-based algorithms that improve on the running-time at the cost of loosened error guarantees. To our knowledge, this is the first effort to offer scalable solutions to the wavelet thresholding for maximum error metrics problem and thus, we enhance the usability of wavelets in modern applications. In summary, we make the following contributions:

- First, considering one-dimensional datasets, we present a general framework for scaling the existing DP algorithms for the problem. Our approach is based on a novel error tree decomposition that allows the parallel processing of DP table rows. In order to demonstrate the benefits of our framework, we apply it on the state-of-the-art *IndirectHaar* [13] algorithm and create *DIndirectHaar*: a distributed and scalable DP algorithm.
- We extend *IndirectHaar* to run over datasets of multiple dimensions and show that our framework for parallelizing DP algorithms can be applied in that case too.
- We propose *DGreedyAbs*, a distributed, heuristic-based algorithm and study its computational complexity. *DGreedyAbs* follows three key ideas: 1) hierarchical decomposition of the error tree , 2) multiple executions of the centralized algorithm, and 3) merging and filtering of intermediate results. According to our experiments, *DGreedyAbs* is more than 2× faster than *DIndirectHaar*. However, as *DGreedyAbs* initiates multiple distributed jobs, we improve on its running-time, and present *BUDGreedyAbs*: A distributed heuristic algorithm that requires a single job for constructing the synopsis.
- We extend *GreedyAbs* to handle multidimensional datasets and provide a complete theoretical analysis for the complexity of the new algorithm.
- We implement all the proposed algorithms on top of the Hadoop processing framework and perform an extensive experimental evaluation using both synthetic and real datasets. Previous approaches to 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 268M datapoints.

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 the proposed greedy algorithms. Section 6 describes the required modifications in order to handle multi-dimensional data. Finally in Section 7, we experimentally evaluate our algorithms and in Section 8, we present the conclusions.

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 [17]. The wavelet decomposition is computationally efficient (linear time) and has excellent energy compaction and decorrelation properties, which can be used to effectively generate compact representations that exploit the structure of data.

2.1 One-Dimensional 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 (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

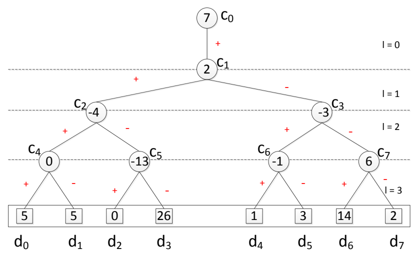


Fig. 1: An error tree that illustrates the hierarchical structure of the Haar wavelet decomposition

The *error tree*, introduced in [19], is a hierarchical structure that illustrates the key properties of the Haar wavelet decomposition. Fig. 1 depicts the error tree for our simple example data vector A. Each internal node $c_i\,(i = 0, \dots, 7)$ is associated with a wavelet coefficient value, and each leaf $d_i\,(i = 0, \dots, 7)$ 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) subtree of c_k. We denote $path_k$ 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.

Given the error tree representation of a one-dimensional Haar wavelet transform, we can reconstruct any data value d_i using only the nodes that lie on $path_i$. That is $d_i = \Sigma_{c_j \in path_i} \delta_{ij} \cdot c_j$, where the factor $\delta_{ij} = 1$ if $d_i \in leftleaves_j$ or $j = 0$ and $\delta_{ij} = -1$ otherwise.

2.3 Multidimensional Haar Wavelets

The Haar wavelet decomposition can be extended to multiple dimensions using two distinct methods, namely the *standard* and *nonstandard* decomposition [10]. Each of these transforms results from a natural generalization of the one-dimensional decomposition. Considering a D-dimensional array A of size N, where N is the number of datapoints, the wavelet transform produces a D-dimensional array W_A of the same shape with A. To simplify the exposition to the basic ideas of multidimensional wavelets, we assume all dimensions of the input array to be of equal size.

The work presented in this paper is based on the nonstandard decomposition. Abstractly, the nonstandard decomposition alternates between dimensions during successive steps of pairwise averaging and differencing: given an ordering for the data dimensions $(1, 2, \dots, D)$, we perform one step of pairwise averaging and differencing for each one-dimensional row of array cells along dimension k, $\forall k \in [1, D]$. The results of earlier averaging and differencing steps are treated as data values for larger values of k. One way of conceptualizing this procedure is to think of a 2^D hyper-box being shifted across the data array, performing pairwise averaging and differencing. We then gather the average value of each individual 2^D hyper-box and we form a new array of lower resolution. The process is then repeated recursively on the new array. An example of this process for a two-dimensional 4×4 data array is illustrated in Fig. 2. We demonstrate the process for the lower left quadrant of the array. Initially, we have the values: 1, 4, 9 and 6. By pairwise averaging and differencing along the first dimension we get: $(1 + 4)/2 = 2.5$, $(1 - 4)/2 = -1.5$ and $(9 + 6)/2 = 7.5$, $(9 - 6)/2 = 1.5$. The quadrant is now transformed to the values: 2.5, -1.5, 7.5, 1.5. We repeat the same process along the second dimension and we have: $(2.5 + 7.5)/2 = 5$, $(2.5 - 7.5)/2 = -2.5$ and $(-1.5 + 1.5)/2 = 0$, $(-1.5 - 1.5)/2 = -1.5$ and the quadrant is transformed to the values: 5, 0, -2.5, -1.5 as shown in Fig. 2. We apply the same process on the other three quadrants of the array in order to complete the first level of the wavelet decomposition. In the next step, we gather the computed averages of each hyperbox (highlighted with grey color) and this way we form an array of lower resolution as shown in the 3rd step of Fig. 2. We then repeat the same procedure for the next level of resolution. More information about the wavelet transform can be found in [10].

Error tree structures are also defined for multidimensional Haar wavelets and can be constructed (once again in linear time) in a manner similar to the one-dimensional case. Nevertheless, the semantics and structure are somewhat more complex. Fig. 3

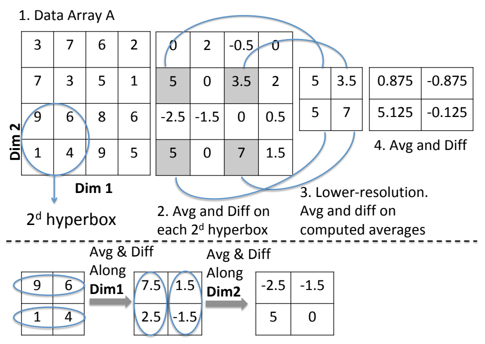


Fig. 2: Example of two-dimensional Haar wavelet decomposition

illustrates the error tree structure for the two-dimensional decomposition presented above, annotated with the sign-information for each coefficient. A major difference is that in a D-dimensional error tree, each node t (except for the root) contains a set of $2^D - 1$ wavelet coefficients c_{ti} that have the same support region but different signs and magnitudes for their contribution. Furthermore, each node t in a D-dimensional error tree has 2^D children corresponding to the quadrants of the support region of all coefficients in node t. The sign of each coefficient's contribution $(sign\,(j, i))$ to the j-th child of node t is determined by the coefficient's position in the 2^D-hyperbox. Coefficients located in the same position of different 2^D-hyperboxes will be assigned the same internal index. Thus, internal indexing determines the sign of contribution of a coefficient c_{ti} to each child of node t. For example, we observe in Fig. 3 the sign-information for the first coefficient of each node (internal index 0). Every coefficient with internal index equal to zero contributes positively to the first and third child and negatively to the second and fourth.

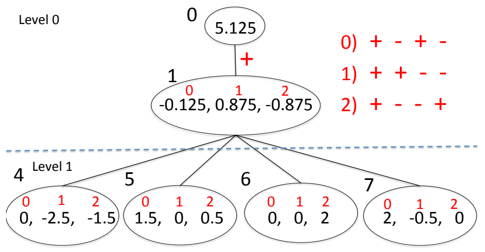


Fig. 3: Two-dimensional error tree. Each node contains $2^2 - 1 = 3$ coefficients and has $2^2 = 4$ children. The numbers in red color indicate the coefficients' indexing within a node.

2.4 Wavelet Thresholding

The complete Haar wavelet decomposition W_A of a data array 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. For assessing the quality of a wavelet synopsis, many aggregate error-measures have been proposed [24]. Among the most popular metrics are the *mean squared error* (L_2), the *maximum absolute error* and the *maximum relative error*. 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. 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 [17] 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. Maximum error metrics are more robust [11], [23], since they set a maximum error guarantee on individual values. The problem of minimizing maximum error metrics can be formulated as follows:

Problem 1. *Given a data array 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.*

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 dataset in memory or operate on a small working set and make very frequent disk accesses to update it. The increasing sizes of data to be processed render centralized approaches 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. We first describe our algorithms in detail for the one-dimensional case and in Section 6 we discuss the required modifications for handling multidimensional datasets.

3 Related Work

In this Section, we make a literature review and present related research. In [23], a probabilistic DP algorithm is proposed. The running-time of the algorithm is $O\left(N\delta^2 Blog\left(\delta B\right)\right)$. However, as there is always a possibility of a "bad" sequence of coin flips, this approach can lead to a poor quality synopsis. As an improvement, a deterministic DP approach is proposed in [11]. Unfortunately, the optimal solution provided has a high time complexity of $O\left(N^2 BlogB\right)$. These solutions are very expensive in terms of time and space and such requirements render them impracticable for the purpose of quick and space-efficient data summarization.

In order to decrease space complexity, Guha introduces a generally applicable, space efficient technique [25] for all these DP-based approaches, that needs linear space for the synopsis construction but at the cost of a $O\left(N^2\right)$ running time.

A more recent and sophisticated approach is presented in [14]. Karras and Mamoulis devise Haar+: a modified error tree, whose structure gives more flexibility on choosing which coefficients to keep. For the thresholding, a DP algorithm with running-time complexity $O\left(\left(\frac{\Delta}{\delta}^2\right)NB\right)$ is presented.

A different approach is proposed in [13]. The authors design a solution that tackles Problem 1 by running multiple times a DP algorithm for the dual problem [13], [26], [27]. The resulting complexity is $O((\frac{\mathscr{E}}{\delta})^2 N(log\varepsilon^* + logN))$, where $\mathscr{E}$ 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. However, problems like excessive demand for main memory capacity have not yet been resolved.

Similar DP algorithms have been also proposed for the minimization of general distributive errors like the L_p norm [28], [29]. The framework we propose at this work for the parallelization of DP algorithms for Problem 1 can be seamlessly used to speedup the execution of these algorithms too.

In order to decrease running-time, greedy algorithms have been proposed [12], [30] for the minimization of the maximum absolute and relative error with worst-case running-time complexities of $O\left(Nlog^2N\right)$ and $O\left(Nlog^3N\right)$ respectively. These algorithms present almost linear behavior in practice and require less memory capacity than most of the DP-based ones. Nevertheless, since they run in a centralized fashion, as data scales close to the memory constraints of the machine, their performance significantly deteriorates. Moreover, they have inherent difficulties in their parallelization and thus, the decomposition to local sub-problems is not an easy task to accomplish.

A first attempt to tackle this problem, i.e., parallelizing the greedy algorithms of [12], [30], is presented in [31]. In this work, a more time-efficient algorithm than *DGreedyAbs* [31] is presented for creating wavelet synopses that target maximum-error metrics. The proposed algorithms of this paper not only improve on the running-time but also handle data of arbitrary dimensionality while the work of [31] is limited to one dimension.

The work in [32] considers a distributed setting for the wavelet decomposition, implemented on top of Hadoop, but it only targets L_2 error minimization, which is a considerably easier task.

Although there is a lot of research for the one-dimensional case, few attempts have been made to approach the multi-dimensional version of the problem. In [10], algorithms for multi-dimensional wavelet decomposition and thresholding are presented. However, only conventional thresholding is studied and there is no proposed algorithm for maximum-error metrics.

In [11] the authors present deterministic, exact and approximate DP-based algorithms for the problem. The most time-efficient algorithm is a $(1+\varepsilon)-$approximation algorithm with running time: $O(\frac{logR_Z}{\varepsilon}2^{2^D+3D}Nlog^2NBlogB)$ for a D-dimensional dataset. Despite the optimal quality, the running-time of these algorithms is prohibitive for real-world scenarios even for small data dimensionalities (i.e., $D \in [2,5]$, where wavelet-based data reduction is typically employed[1]).

A multidimensional extension of the DP algorithm that targets the Haar+ tree is presented in [33]. The algorithm has a running-time complexity of $O\left(2^{2^D}\left(\frac{\Delta}{\delta}^{2^D}\right)NB\right)$, which is prohibitively large for processing real-life big datasets.

For dealing with multi-dimensional datasets, the authors of [30] propose mapping all data to one dimension by using a space filling curve. The drawback of this approach is that it destroys data locality and thus can lead to sub-optimal quality results.

1. Due to the "dimensionality curse", wavelets and other space-partitioning schemes become ineffective above 5-6 dimensions.

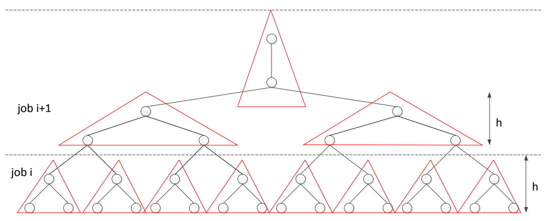


Fig. 4: Partitioning for parallelizing DP algorithms for Problem 1

In [34], an algorithm of $O(N)$ time complexity is proposed for solving the dual problem. Once again, a space-bounded synopsis can be constructed by employing the technique in [13]. However, the algorithm of [34] is presented in a centralized setting and there is no evidence of its performance on large scale datasets.

A similar algorithm is also presented in [35] for image compression and thus only covers two-dimensional datasets. However, the algorithm is still applicable on small datasets.

Wavelets opt for a hierarchical decomposition of a data distribution. Hierarchical decomposition is a powerful concept in data analysis that permits the approximation of a dataset at different accuracy levels. Data Canopy [36] makes use of similar ideas to break statistics down to basic aggregates that can be used as building blocks for subsequent computations.

4 Scaling DP algorithms

Since the majority of the proposed algorithms for Problem 1 are based on DP, 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 DP-based 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.

The left and right subtree of a node c_j 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 independent subtrees of a fixed height h, $h < logN$. This partitioning scheme is presented in Fig. 4 and results to $\lceil \frac{logN}{h} \rceil$ layers of subtrees.

For the parallelization of the existing DP algorithms for Problem 1, we use Algorithm 1. The idea is to first run the DP algorithm in parallel over the subtrees of the bottommost layer. When the processing is over, the computed rows for the roots of these subtrees 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, proper partitioning should be applied between different stages in order to preserve the subtree locality in the next layer.

As a distributed approach, it is clear that this idea incurs a communication overhead. For every subtree of the error tree, the

Algorithm 1 Parallel execution of a DP algorithm for Problem1

```
Require: Data size N, subtree height h
1: Partition the error tree to subtrees 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:     $i = i+1$
9: Run DP on topmost subtree.
```

row of M that corresponds to the local root is transferred over to the workers of the next stage. The following Lemma quantifies the cost of this overhead.

Lemma 1. *The overall communication cost of Algorithm 1 is:*

$$O\left(\frac{N \cdot max\{|M[j]|\}}{2^h}\right) \quad (1)$$

After the completion of Algorithm 1, it is only the optimal approximation error 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 subtree and select the coefficients to retain. When the processing of the topmost subtree is over, we know which coefficients are retained from this subtree and also the leaves of the subtree are aware of 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 subtree 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 subtrees.

For demonstrating the merits of our approach, in this work we apply our methodology on *IndirectHaar* [13] creating *DIndirectHaar*; a distributed version of the centralized algorithm. Our experiments in Section 7 show that *DIndirectHaar* scales linearly over both data and cluster size.

At this point, we also want to discuss the choice of *IndirectHaar*. An exact solution for Problem 1 demands tabulation over all possible space allocations for each node of the error tree. This burden renders the majority of DP-algorithms impractical in terms of memory consumption. *IndirectHaar* exploits the dual error-bound problem which is easier to be solved and employs a binary search procedure to derive a solution for the initial problem. Thus, in the case of *IndirectHaar*, the DP algorithm that is actually parallelized by our framework is *MinHaarSpace* [13] and targets the dual of Problem 1. For achieving even better results, *IndirectHaar* can also be applied on Haar+ trees [14]. However, as Haar+ trees have a slightly different structure and work on triads of coefficients, for the ease of understanding, we keep the presentation on the classic Haar error tree.

5 Parallel Greedy Approaches

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* [12] 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 two fully parallel greedy algorithms that both are based on : (i) a partitioning scheme similar to the one presented in Section 4, and (ii) merging and filtering of partial results.

5.1 *GreedyAbs*

To assist in our discussion, we first give a description of the *GreedyAbs* algorithm [12]. 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:

$$MA_k = max_{d_j \in leaves_k}\{|err_j - \delta_{jk} \cdot c_k|\} \tag{2}$$

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 Fig. 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) sub-tree 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^l_k, min^l_k, max^r_k, and min^r_k, respectively. It follows that Equation 2 is equivalent to:

$$MA_k = max\{|max^l_k - c_k|, |min^l_k - c_k|, \\ |max^r_k + c_k|, |min^r_k + c_k|\} \tag{3}$$

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. This 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 reason, we keep removing coefficients 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.

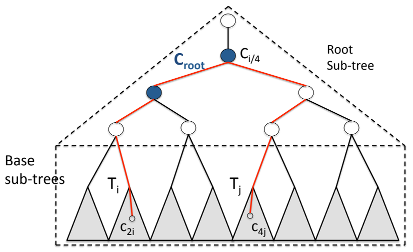


Fig. 5: Partitioning for parallelizing *GreedyAbs*. The red line illustrates an example of communication between two base subtrees. The blue-filled nodes show a possible C_{root} set.

5.2 *DGreedyAbs*: Scaling the Greedy Algorithm

GreedyAbs presents an inherent drawback for its parallelization. At each step, it needs global knowledge of the whole error tree. To solve the problem in parallel, we 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 subtree* and multiple *base subtrees*, as shown in Fig. 5.

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 subtree independently of each other. For the parallelization of the algorithm, the main difficulty is that the base subtrees communicate with each other through the root subtree. For example, consider a scenario, like the one depicted in Fig. 5, where node c_{2i} is selected to be removed from the base subtree T_i and, at the same time, node c_{4j} is selected from T_j. The removal of c_{2i} may dictate that node $c_{i/4}$ should be discarded at the next step. On the other hand, discarding c_{4j} can make $c_{i/4}$ a really important coefficient for subtree T_j and thus its deletion could produce a big maximum error. It is clear that such situations lead to conflicts that prohibit a straight-forward parallel implementation.

In order to proceed towards a correct parallel computation, we need to offer more isolation to the base subtrees. The idea behind our solution is the following: Let us assume that we somehow know which nodes of the root subtree are retained in the final synopsis and call this set of nodes C_{root}. Having selected a C_{root}, we can remove the remaining root subtree and there are $B - |C_{root}|$ nodes that still need to be selected for the synopsis. Consider now a base subtree T_j. The deletion of nodes $c_i \in$ root subtree$\setminus C_{root}$ incurs an incoming error to T_j. For example, in the error tree of Fig. 1, if we delete nodes $\{c_0, c_2\}$, there is an incoming error $-7-4 = -11$ to subtree T_5. Thus, if the incoming error to subtree T_j is e_{in}, we set the signed accumulated errors to: $err_i = e_{in}, \forall d_i \in T_j$ and run *GreedyAbs* on T_j. The output of *GreedyAbs*(T_j) is an ordered list L_j of $N_z(T_j)$ coefficients, where $N_z(T_j)$ is the number of non-zero coefficients in the subtree. The list is in reverse order of the one in which coefficients $c_i \in T_j$ were deleted by *GreedyAbs*. More specifically, each element of the list is a tuple $(delOrd, id, err)$ that indicates the order with which the coefficient with index id was deleted and the incurred maximum absolute error err. This procedure of locally executing *GreedyAbs* on a subtree, is carried out in parallel for all base subtrees.

When this stage of parallel *GreedyAbs* runs is over, we collect and error-wise merge the outputs from all the base subtrees (i.e., $\forall T_j \in$ base subtrees merge L_j), thus obtaining a global list where the node deletion order of each subtree is preserved. The synopsis needs to contain those coefficients that are the most important for each subtree, i.e., the ones that were last emitted. Therefore, by keeping the last $B-|C_{root}|$ elements of the global list, let us call them C_{base}, we form the final synopsis: $C_{root} \cup C_{base}$. This procedure is presented in Algorithm 2.

Algorithm 2 distrGAbs

```
Require: error tree, space budget $B$, $C_{root}$ set
1: for all $T_j \in$ base subtrees in parallel do
2:     $err_i = err_i + e_{in}, \forall err_i \in T_j$ // $e_{in}$: incoming error from $C_{root}$
3:     $L_j = GreedyAbs(T_j)$;emit $L_j$
4: $L = merge(L_j$ lists)
5: store last $C_{root} \cup (B-|C_{root}|)$ elements of $L$ as synopsis
6: return $L[B-|C_{root}|-1].error$
```

So far, we have ignored the procedure that finds the appropriate nodes to be retained from the root subtree, assuming it is provided by an "oracle". 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 subtree. Since we do not know the number of nodes that should be retained from the root subtree, we should consider 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 subtree. The intuition behind this choice is that, since only the root subtree 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. *GreedyAbs* on the root subtree runs in a centralized fashion. Since the root subtree can be exponentially smaller than the original dataset, its processing on a single machine is done without compromising performance. The candidate C_{root} sets are generated by the *genRootSets* function presented in Algorithm 3.

Algorithm 3 *genRootSets*

```
Require: root subtree,B
1: $L_{root} = GreedyAbs$(root subtree)
2: $C = \{\{\}\}$; $lastIndex = L_{root}.size$
3: for $(i = lastIndex; i > lastIndex - B; i = i-1)$ do
4:     $C_{root,i} = \{L_{root}[i], .., L_{root}[lastIndex]\}$; $C = C \cup \{C_{root,i}\}$
5: return $C$
```

For example, we consider as root subtree the nodes $\{c_0,c_1,c_2,c_3\}$ of the error tree depicted in Fig. 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: $\Gamma = [\{c_1,c_3,c_2,c_0\},\{c_3,c_2,c_0\},\{c_2,c_0\},\{c_0\},\{\}]$. For constructing the synopsis, we perform a search in the space of possible solutions. We start by examining the achieved quality of the corner cases, i.e., keeping in the synopsis 0 and $min\{R,B\}$ coefficients from the root subtree. If these extreme cases result in errors e_h, e_l with $|e_h - e_l| < \varepsilon \to 0$, then the algorithm finishes and we keep as a final synopsis the one that produced the $min\{e_h,e_l\}$. Otherwise, we replace the C_{root} that produced the $max\{e_h,e_l\}$ with another C_{root} produced by Algorithm 3 and repeat the same process. The selection of the next C_{root} does not come from a random choice. When the distributed execution of the greedy algorithm for a given C_{root} set is over, we know the maximum absolute error that appeared in each base subtree. By knowing that information, we know which subtrees need further improvement. Thus, we select C_{root} sets that contain coefficients which support these subtrees. In our example, we begin by running *GreedyAbs* on each base subtree for the C_{root} sets: $\{\}$, $\{c_1,c_3,c_2,c_0\}$. Let us assume that they yield synopses with errors 10 and 5 respectively. In that case, as $\{\}$ produced the worst error, it is replaced by $\{c_0\}$ and we now compare the quality of the synopses yielded by $\{c_0\}$ and $\{c_1,c_3,c_2,c_0\}$ C_{root} sets. The described procedure implies that $O(R)$ jobs may be demanded. However, our experiments in Section 7 show that the number of jobs that the algorithm needs in order to converge is constant in practice.

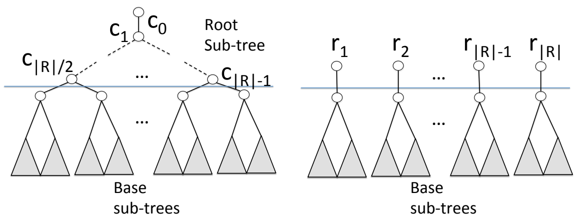


Fig. 6: Equivalent representations of an error tree.

The running-time and communication complexity of *DGreedyAbs* are provided by the following Lemma:

Lemma 2. *Let us denote with R the size of the root-subtree, S the size of a base-subtree and $Nz(S)$ the number of non-zero coefficients of a base-subtree. Then, the asymptotic running-time complexity of a* DGreedyAbs *worker is $O\left(Nz(S)\,log^2 Nz(S)\right)$ and the communication cost is $O(Rmax\{B,Nz(S)\})$.*

5.3 Speeding up the Distributed Greedy Solution

While the *DGreedyAbs* algorithm succeeds in offering a viable solution to the problem, it suffers from a basic drawback: There are multiple distributed jobs that may be required to create the synopsis, and thus, the centralized algorithm needs to run multiple times over the same data. Since we do not know in advance which are the appropriate nodes to retain from the root-subtree, we run *GreedyAbs* for many possible C_{root} sets, incurring extra computational and communication overhead.

In order to alleviate this overhead, in this Section we propose *BUDGreedyAbs*: a modified, bottom-up version of *DGreedyAbs* that makes only one pass over the dataset and executes the centralized greedy algorithm only once per subtree.

In order to avoid the multiple jobs, consider the following strategy. Assume that the size of the root-subtree is less than B and we a-priori decide to retain it all in the synopsis and then run *GreedyAbs* to all workers in parallel. One could say that this is the safest choice as we keep a maximal amount of information about the part of the tree that creates dependencies among partitions. However, this is not optimal. Some nodes of the root-subtree may be of negligible importance and by keeping them, we sacrifice budget space that could be allocated in a smarter way. The basic idea behind *BUDGreedyAbs* is to start from the safest choice of keeping all of the root-subtree and adaptively refining it.

We start with a Lemma that follows directly from the properties of the wavelet transform. The idea of Lemma 3 is also presented graphically in Fig. 6.

Lemma 3. *An error-tree partitioned to one root-subtree and many base subtrees $S_i, i=1,..,R$, is equivalent to R independent error-trees $S'_i, i=1,..,R$, where each $S'_i = S_i$ with an extra coefficient r_i*

as root. The value of r_i is defined as: $r_i = \Sigma_{c_j \in paths_{S_i}} \delta_{ij} \cdot c_j$ and is also equal to the average of all data values in base-subtree S_i.

According to Lemma 3, instead of computing the full wavelet transform of the error-tree, we can compute the transform up to the height of the base-subtrees and also keep the local root-average of each subtree. That is what *BUDGreedyAbs* does. It first computes a wavelet structure as the one of Fig. 6b. Then, it triggers a parallel execution of *GreedyAbs* at each base-subtree. The outputs are merged in the same way as in Section 5.2 and a maximum error is computed. As the yielded synopsis may contain some of the r_i coefficients, in order to better exploit the available space budget, in a next step the algorithm examines opportunities for further compression and computes the root-subtree solely based on these r_i coefficients contained in the synopsis.

As the first part of the algorithm is the same with *DGreedyAbs*, we discuss the algorithmic details of merging and how more accurate configurations for the root-subtree are explored. For explaining these details, we give the following example:

Example. In Fig. 7 we present two base-subtrees S_1, S_2. The lists L_1, L_2 show the most important coefficients from the corresponding outputs of *GreedyAbs*. Thus, in subtree 1, the last nine coefficients that the algorithm would delete are the ones in the array L_1 with c_{a8} discarded first. As we have said, in order to create the final synopsis, we need to merge L_1 and L_2 from left to right and examine the errors of the first B coefficients. Let us assume $B = 16$. The first nine coefficients of the synopsis would be $c_{a1}, ..., c_{a4}$ and $c_{b1}, ..., c_{b5}$. At this point, we check the r_i coefficients. Instead of keeping both of them and waste two slots in the synopsis, we examine the possible merits of increasing compression in the root-subtree. We calculate the wavelet transform of the root-subtree considering as data values the $r_i, i = 1,2$ coefficients. In our example of Fig. 7, the transform results in the creation of c_0 and c_1. According to Lemma 3, keeping both c_0 and c_1 is completely equivalent to keeping r_1 and r_2 both in terms of appoximation quality and space overhead. Thus, we also examine the chance of keeping only c_0 or c_1 or even none of them. In order to preserve correctness, as *GreedyAbs* has run at each subtree considering all r_i nodes retained in the synopsis, the posterior deletion of coefficients from the root-subtree should be accompanied by some error updates. Let us assume that we first examine the deletion of node c_1. Some of the nodes in L_1 and L_2 should update their observed signed errors by $-c_1$.

But which nodes need to be updated? The errors which are reported by the coefficients in the red box, i.e., the ones in the right side of r_1 in Fig. 7, are calculated taking into account that r_1 is retained in the synopsis. Thus, a deletion of a node that contributes to the r_1 value must be reflected to the errors observed by these nodes. On the other hand, the nodes in the left side of r_1 consider r_1 already discarded and as such, nothing more is needed to be done on them.

Which nodes of the root-subtree should we consider to delete? Do we have to try all R nodes in all possible combinations? We treat this issue the same way as we did for *DGreedyAbs*. We run *GreedyAbs* on the root-subtree and then execute Algorithm 3. The output of Algorithm 3 represents the candidate combinations for deletion. For each of them, we update the errors in L_i lists and merge them in a final list where the B first nodes are considered for the synopsis. *BUDGreedyAbs* is presented in Algorithm 4.

Complexity Analysis. The complexity of the parallel workers of *BUDGreedyAbs* is the same with that of *GreedyAbs*, i.e.,

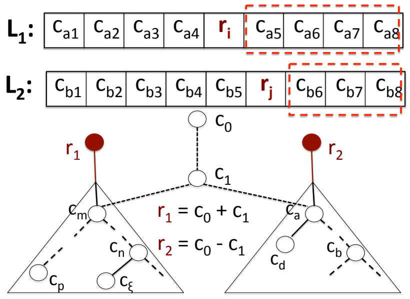

L1:
ca1 ca2 ca3 ca4 ri ca5 ca6 ca7 ca8
L2:
cb1 cb2 cb3 cb4 cb5 rj cb6 cb7 cb8
c0
c1
r1
r2
r1 = c0 + c1
r2 = c0 - c1
cm
cn
cp
cξ
ca
cb
cd

Fig. 7: Example of merging solutions for *BUDGreedyAbs*.

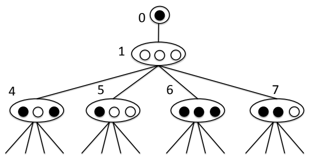


Fig. 8: Thresholding in a 2-dimensional error tree.

$O\left(Nlog^2 N\right)$. However, in the next stage of *BUDGreedyAbs*, as we have seen there are R merge operations that take place and in the worst case, each of them needs to process B elements. Furthermore, we also need to run *GreedyAbs* once on the root-subtree. Thus, the complexity of the reduce workers that derive the final synopsis is $O\left(Rlog^2 R + RB\right)$.

Algorithm 4 *BUDGreedyAbs*

```
Require: error tree, space budget $B$
 1: for all $T_i \in$ base subtrees in parallel do
 2:     $L_i = GreedyAbs(T_i)$; emit $L_i$
 3: $L = merge(L_i$ lists)
 4: $synopsis$ =first $B$ elements of $L$; $error = max\_abs(synopsis)$
 5: $RA = \{r_i | L[j] = r_i \wedge 0 \le j < B\}$ // $r_i$: avg of data values in $T_i$
 6: Root-subtree= $WaveletTransform(RA)$
 7: $\Gamma$ = $genRootSets$(Root-subtree,$B$)
 8: for all $C_{root} \in \Gamma$ do
 9:     for all $L_i$ do
10:         if $L_i[j] = r_i$ then update errors at $L_i[k], k > j$
11:     $L = merge(L_j$ lists)
12:     if $max\_abs$(first $B$ elements of $L$) $< error$ then
13:         $synopsis$ =first $B$ elements of $L$
14:         $error = max\_abs(synopsis)$
15: return $synopsis$
```

6 Extension to Multiple Dimensions

The algorithms discussed so far are applicable on one-dimensional datasets. However, datasets with multiple dimensions involved are a common case in real-life applications. In this Section, we discuss the modifications required in order to extend both the centralized *GreedyAbs* and *MinHaarSpace* [13] to deal with multiple dimensions. The main difference is that now the distributed algorithms of Sections 4, 5 run over a multidimensional error tree and instead of using *GreedyAbs* and *MinHaarSpace*, they employ the modified algorithms we are going to present in this Section.

As we mentioned in Section 2, the structure of a D-dimensional error tree (Fig. 3) is somewhat more complex. As opposed to the one-dimensional case, each node of the tree contains many coefficients and thus the terms *node* and *coefficient* should be distinguished. During thresholding, it is not necessary to pick or discard all coefficients of a node at the same time. In Fig. 8 we see a snapshot where the black-filled coefficients are retained in the synopsis, while the blank ones are discarded. For the navigation in a multidimensional error tree, we follow the indexing presented in Fig. 3. The first node at each level has index $2^{D \cdot level}$. The rest of the nodes of the same level maintain index values increased by one each. For our example and the two-dimensional case, in level 1, the first node has index $2^{2 \cdot 1} = 4$, the next node of this level has index 5 and the remaining two have indices 6 and 7 respectively. With that indexing scheme, we can easily navigate the error tree. Dividing a node's index by 2^D leads us to the parent of the node. For the identification of an individual coefficient within a node, we apply internal indexing. The internal index of a coefficient belongs in the interval: $[0, 2^D - 1)$. The notation c_{ij} denotes the j-th coefficient in node i.

6.1 *MDMSpace*: D-dimensional *MinHaarSpace*

In order to explain the required modifications for extending *MinHaarSpace* to multiple dimensions, we first present the idea behind the one-dimensional version of the algorithm. As *MinHaarSpace* solves the dual of Problem 1, its goal is to construct a synopsis of minimum size for a given error ε. An incoming value at node c_i of the error tree is a value reconstructed in the path of ancestor coefficients from the root node up to c_i. In a wavelet decomposition, this is the average value in the interval under the scope of c_i. Similarly, an assigned value z_i at node c_i is a coefficient value retained at that node [2]. *MinHaarSpace* is a DP algorithm that, following a bottom-up process, considers all possible incoming values v at each node c_i and for each v, considers all possible assigned values z_i^v. It then determines the optimal one to assign at c_i for v. For making the exploration of both incoming values at c_i and assigned values in the coefficients feasible, the real-valued domains of v and z_i^v are quantized into multiples of a small resolution step δ.

In order to enumerate all possible incoming values v for a node c_i and all permitted values for z_i^v, we need to find some bounds for the corresponding value domains. For delimiting the domain of incoming values the following lemma holds:

Lemma 4. *Let v_i be the real incoming value at node c_i, i.e., the one when no coefficient is discarded. Let v be an incoming value to c_i for which the error bound ε under the maximum absolute error metric can be satisfied; then $|v_i - v| \leq \varepsilon$.*

Lemma 4 holds also in the multidimensional case and implies that the finite set of possible incoming values we have to examine at node c_i consists of the multiples of δ in the interval $S_i = [v_i - \varepsilon, v_i + \varepsilon]$; thus, $|S_i| = \lfloor \frac{2\varepsilon}{\delta} \rfloor + 1 = O\left(\frac{\varepsilon}{\delta}\right)$.

For delimiting the domain of assigned coefficients z_i^v for a given incoming value v there is Lemma 5 that holds in the one-dimensional case:

Lemma 5. *Let v_i be the real incoming value to node c_i, z_i the real assigned value at c_i, $v \in S_i$ be a possible incoming value to c_i for which the maximum error bound ε can be satisfied, and z_i^v be a value that can be assigned at c_i for incoming value v, satisfying ε; then $|z_i - z_i^v| \leq \varepsilon - |v_i - v|$*

2. *MinHaarSpace* uses unrestricted wavelets. The values for the coefficients do not need to be the ones computed by the wavelet transform but any real number as long as the error tree properties are preserved. In order to distinguish the coefficient from the arbitrary value that is assigned, we use for the assigned value the notation z_i and $z_{i,j}$ for the 1- and D-dimensional cases respectively.

Proof. Let v, z_i^v be the incoming and the assigned values at node c_i and v_i, z_i be the real incoming and real assigned values respectively. Let also v_{2i}^* be the incoming value for c_{2i}. Then, the incoming and real incoming values at node c_{2i} are: $v_{2i} = v_i + z_i$ and $v_{2i}^* = v + z_i^v$. As Lemma 4 should hold for node c_{2i}, we should pick a value for z_i^v such that: $|v_{2i}^* - v_{2i}| \leq \varepsilon$, i.e., $|v + z_i^v - v_i - z_i| \leq \varepsilon$ which leads to $|z_i - z_i^v| \leq \varepsilon - |v_i - v|$. □

Lemma 5 implies that the finite set of possible assigned values we have to examine at node c_i, for a given incoming value $v \in S_i$, consists of the multiples of δ in the interval $S_i^v = [z_i - (\varepsilon - |v_i - v|), z_i + (\varepsilon - |v_i - v|)]$.

By examining the Proof of Lemma 5, we observe some differences that exist in the multidimensional case. As we have said, a node c_i of a D-dimensional error tree contains $2^D - 1$ coefficients $z_{i,j}$ that all contribute to its 2^D children. Thus, for an incoming value v_i at node c_i, the incoming value at its j-th child is: $v_i + \sum_{k=0}^{2^D-2} \left(z_{i,k} sign(k,j)\right)$. Following a similar reasoning to the proof above, we get the inequalities:

$$|\sum_{k=0}^{2^D-2} \left(z_{i,k} - z_{i,k}^v\right) sign(k,j)| \leq \varepsilon - |v_i - v|, \forall j \in 0,..,2^D - 2 \quad (4)$$

A pairwise addition of the above inequalities leads to $2^D - 1$ more inequalities that place bounds to the candidate values of every individual coefficient $c_{i,k}$:

$$|z_{i,k} - z_{i,k}^v| \leq \varepsilon - |v_i - v|, \forall k \in 0,..,2^D - 2 \quad (5)$$

For a given incoming value v at node c_i, the possible assigned values for every coefficient $c_{i,k}, k = 0,..,2^D - 2$ comprise the finite set of the multiples of δ in the interval: $S_{i,k}^v = \left[z_{i.k} - (\varepsilon - |v_i - v|), z_{i,k} + (\varepsilon - |v_i - v|)\right]$ that also satisfy Inequalities 4. As $|S_{i,k}^v| = O\left(\frac{\varepsilon}{\delta}\right)$, and each node contains $2^D - 1$ coefficients, the number of examined values for node c_i is $O\left(\left(\frac{\varepsilon}{\delta}\right)^{2^D-1}\right)$.

The *MDMSpace* procedure works in a bottom-up left-to-right scan over the error tree. At each visited node c_i it calculates an array A of size $|S_i|$ from the precalculated arrays of its children nodes. A holds an entry $A[v]$ for each possible incoming value v at c_i. Such an entry contains: (i) the minimum number $A[v].s = S(i,v)$ of non-zero coefficients that need to be retained in the subtree rooted at c_i with incoming value v, so that the resulting synopsis satisfies the error bound ε, (ii) the δ-optimal values $A[v].\left(z_{i,0}^v, .., z_{i,2^D-2}^v\right)$ to assign at c_i, for incoming value v, and (iii) the actual minimized maximum error $A[v].e$ obtained in the scope of c_i. $S(i,v)$ is recursively expressed as:

$$S(i,v) = \min_{z_{i,k} \in S_{i,k}^v} \left(\sum_{j=i2^D}^{i2^{D+1}-1} S(j, v + \sum_{k=0}^{2^D-2} z_{i,k} sign(j,k)) + \sum_{k=0}^{2^D-2} (z_{i,k} \neq 0) \right)$$

$$S(0,0) = \min_{z_{0,0} \in S_{0,0}^0} (S(1, z_{0,0}) + (z_{0,0} \neq 0))$$

The above equations compute the smallest between (i) the minimum required space if a non-zero coefficient value $z_{i,k}$ is assigned at $c_{i,k}$; and (ii) the required space if a zero value is assigned at it. The latter case applies only if $0 \in S_{i,k}^v$. Let $S_{i,k}^v$ denote the set of those assigned values at $c_{i,k}$ for incoming value v that require the minimum space in order to achieve the error bound ε: The δ-optimal value to select is the one among these candidates that also minimizes, in a secondary priority, the obtained maximum absolute error in the scope of c_i. So, we also need the equations:

$$E(i,v) = \min_{z_{i,k} \in S_{i,k}^v} \left(\max_{j=i2^D}^{i2^{D+1}-1} E(j, v + \sum_{k=0}^{2^D-2} z_{i,k} sign(j,k)) \right)$$

$$E(0,0) = \min_{z_{0,0} \in S^0_{0,0}} (E(1,z_{0,0}))$$

Complexity Analysis. The result array A on each node c_i holds $|S_i|$ entries, one for each possible incoming value, hence its size is $O\left(\frac{\varepsilon}{\delta}\right)$. Moreover, at each node c_i and for each $v \in S_i$, we loop through all $\prod_{k=0}^{2^D-2} |S^v_{i,k}| = O\left(\left(\frac{\varepsilon}{\delta}\right)^{2^D-1}\right)$ possible assigned values. Thus, the runtime of *MDMSpace(0, ε)* is $O\left(\left(\frac{\varepsilon}{\delta}\right)^{2^D} N\right)$.

6.2 *MGreedyAbs*: D-dimensional *GreedyAbs*

For extending the algorithms of Section 5 to multiple dimensions, we first need to modify the centralized *GreedyAbs* algorithm.

As in the one-dimensional case, the greedy algorithm picks each time the coefficient c_{jk} with the lowest MA and discards it. According to Equation 3, the computation of MA_k for a node c_k demanded four values (max^l_k, min^l_k, max^r_k, min^r_k): the maximum and minimum error for each of the two subtrees of c_k. A node of a D-dimensional error tree has 2^D children. Thus, in order to compute MA_{jk}, we need to know the maximum and minimum error in each of the 2^D subtrees of c_{jk}, thus 2^{D+1} values are required. We can see that all the coefficients of a node support the same region of the original data, and so they should observe the same errors in the reconstruction of the corresponding data values. In that way, we do not need to store at each coefficient the maximum and minimum error observed in each subtree, but all the coefficients of a node can share the same 2^{D+1} values. The equation for the computation of MA_{jk} is:

$$MA_{jk} = \max_{0 \le s \le 2^D-1} \{|max^s_j - sign(s)\, c_{jk}|, |min^s_j - sign(s)\, c_{jk}|\} \quad (6)$$

where s is the index of each subtree of node j and $sign(s)$ is the sign of the error caused in subtree s when deleting coefficient c_{jk}. Similarly to the one-dimensional case, when a coefficient c_{jk} is discarded, its maximum and minimum errors need to be updated, as well as the MA-values of all coefficients in the subtrees of node j and if needed the coefficients in the ancestors of node j. Furthermore, this time we also need to update the MA-values of the remaining coefficients in node j that are not yet discarded. Algorithm 5 formally presents *MGreedyAbs*, the modified algorithm for handling multi-dimensional data.

Complexity Analysis. The initial heap H can be constructed in $O(N)$ time. The algorithm performs $O(N)$ discarding operations. A dropped coefficient c_{jk} at height h of the error tree has at most 2^{Dh} descendant nodes and each of them at most $2^D - 1$ non-deleted coefficients. Thus, each coefficient at height h has at most $2^{Dh}(2^D-1)$ non-deleted descendant coefficients which must be updated. Moreover, at height h of the error tree, there are $2^{D(logN-h)}(2^D-1)$ coefficients[3]. As all of them will eventually be discarded, the total number of updates in descendants for all coefficients is:

$$\sum_{h=1}^{logN} [2^{Dh}(2^D-1) \cdot 2^{D(logN-h)}(2^D-1)] = (2^D-1)^2 NlogN \quad (7)$$

A discarded coefficient c_{jk} has at most $logN$ ancestor nodes with at most 2^D-1 non-deleted coefficients each, and thus the total number of updates in ancestors for all deleted coefficients is also $O((2^D-1)^2 NlogN)$. Furthermore, for each dropped coefficient c_{jk}, we need to update at most 2^D-2 coefficients in node j, i.e., in the same node of the discarded coefficient. As there are $O(N)$ discarded nodes, the cost of updates in the same node is: $O((2^D-2)N)$ in total. Thus, the total update operations of the algorithm are: $O(NlogN + N)$. Moreover, each update in a coefficient costs its re-positioning in H which is an $O(logN)$ operation. The complexity of the algorithm is thus: $O(Nlog^2N + NlogN)$, that asymptotically remains to be $O(Nlog^2N)$ as in the one-dimensional case.

3. In this proof, it holds that $logN = log_{2^D}N$

Algorithm 5 *MGreedyAbs*
```
1:  Input: $W_A$ vector of N Haar wavelet coefficients
2:  H := create_heap($W_A$)
3:  while H not empty do
4:    discard $c_{jk}$ := H.top // coefficient with smallest $MA_{jk}$
5:    for $s = 0; s \le 2^D - 1; s{+}{+}$ do
6:      $max^s_j = max^s_j - sign(s)\, c_{jk}$; $minx^s_j = minx^s_j - sign(s)\, c_{jk}$
7:    for $i = 0; i \le 2^D - 2; i{+}{+}$ do
8:      if $c_{ji}$ not discarded then
9:        recalculate $MA_{ji}$;update $c_{ji}$'s position in H
10:   for each subtree $s \in [0, 2^D - 1]$ of node $j$ do
11:     for each coefficient $c_{mn} \in s$ do
12:       if $c_{mn}$ not discarded then
13:         Update all error measures in $c_{mn}$ by $c_{jk}$
14:         recalculate $MA_{mn}$;update $c_{mn}$'s position in H
15:   $max_{err} := \max_{0 \le s \le 2^D-1} \left(max^s_{jk}, max^s_{jk}\right)$;
16:   $min_{err} := \min_{0 \le s \le 2^D-1} \left(min^s_{jk}, min^s_{jk}\right)$; $node_i = node_j$.parent
17:   while $node_i \ne NULL$ do
18:     $max^l_i := max_{err}$; $\min^l_i := \min_{err}$
19:     if any of $\{max^s_i, min^s_i\}, s \in [0, 2^D - 1]$ changed then
20:       if $c_i$ not discarded then
21:         recalculate $MA_i$;update $c_i$'s position in H
22:       $max_{err} := \max_{0 \le s \le 2^D-1} \left(max^s_{jk}, max^s_{jk}\right)$
23:       $min_{err} := \min_{0 \le s \le 2^D-1} \left(min^s_{jk}, min^s_{jk}\right)$
24:       $node_i = node_i$.parent
25:     else break
```

6.3 Discussion

From an algorithmic perspective, the main difference in the construction of one- and D-dimensional wavelet synopses is the structure of the error tree. All the modifications on the proposed algorithms aim at handling error trees in which each node can have an arbitrary number of children.

We observe that the complexity of *MDMSpace* becomes prohibitive even for low dimensionalities. A dimension of $D = 4$ can lead to billions of iterations for the algorithm. On the other hand, the complexity analysis of the greedy algorithms shows that an error tree of a D-dimensional dataset incurs a computational overhead in the order of 2^{2D}. However, this is counterbalanced to some extent by the fact that along with the change in dimensionality, there is also a change in the basis of the logarithm in the complexity formula. Our experiments in Section 7 show that the synopsis construction for a 4-dimensional dataset is only 1.5 times slower than the synopsis construction for a one-dimensional same-sized dataset when the greedy algorithms are employed.

As our experiments indicate, it is more difficult to yield accurate wavelet synopses for datasets of high dimensionality. Intuitively, the higher the number of dimensions, the higher is the number of neighbors for a data-value in the input array. Depending on the distribution, this can lead to an increased number of discontinuities that should be captured by the synopsis.

In Table 2, we summarize the algorithms presented throughout the paper. *IndirectHaar* and *DIndirectHaar* can handle the multidimensional case only if they use *MDMSpace* instead of *MinHaarSpace*. Similarly, *DGreedyAbs* and *BUDGreedyAbs* should use *MGreedyAbs* for handling multiple dimensions.

TABLE 2: Summary of presented algorithms

Algorithm	Distributed	Multidimensional
MinHaarSpace	no	no
MDMSpace **(this work)**	no	**yes**
IndirectHaar	no	**yes**
DIndirectHaar **(this work)**	**yes**	**yes**
GreedyAbs	no	no
MGreedyAbs **(this work)**	no	**yes**
DGreedyAbs **(this work)**	**yes**	**yes**
BUDGreedyAbs **(this work)**	**yes**	**yes**

7 Experimental Evaluation

In the experimental Section we evaluate the proposed algorithms in terms of (i) synopsis construction time and (ii) achieved error. We show that our distributed solutions present linear scalability and we are able to run experiments on bigger datasets than any previous work. All algorithms are implemented in Java 1.8 and for the distributed ones we use the MapReduce programming model.

Datasets. The experiments are conducted using both synthetic and real datasets. As synthetic data, we use uniform and zipfian distributions with data values that lie between [0, 1000]. For the one-dimensional real-life datasets we utilize NYCT [37] and WD [38]. NYCT describes taxi trips in the New York City and contains records for the trip time in seconds. WD consists of observations on wind direction (azimuth degrees) captured during hurricanes in the USA. As multidimensional real-life datasets, we utilize NOAA [39] and NYCT2D [37]. NYCT2D is a 2-dimensional dataset of 1.5 billion records that contains trip distances and total fares for the taxi rides. For NOAA, we consider the following four dimensions: *Wind Direction*, *Wind speed*, *Temperature* and *Dew point*. All datasets are partitioned in order to test scalability over different sizes. The smallest partition comprises the first $1M$ records, while each subsequent partition is 2^D times the previous one, where D is the dataset's dimensionality. Our largest dataset consists of $268M$ datapoints.

Platform setup. For our deployment platform, we use a Hadoop 2.6.5 cluster of 9 machines, each featuring eight Intel Xeon CPU E5405 @ 2.00GHz cores and 8 GB of main memory. One machine is used as the master node and the remaining ones as slaves. Each slave is allowed to run simultaneously up to 5 map tasks and 1 reduce task. Each of these tasks is assigned 1 physical core and 1 GB of main memory. For all the remaining properties, we keep the default Hadoop configuration.

For experimenting with the centralized algorithms we use one machine with the same specifications as the ones listed above. Thus, centralized algorithms may have up to 8 GB of available main memory for their execution.

7.1 Scalability

In this Section, we use synthetic data to assess the scalability with respect to the available budget for the synopsis B, the number of datapoints N and the number of tasks running in parallel. We show that our algorithms can scale to data sizes that state-of-the-art centralized approaches are incapable of. For this Section, we use uniformly distributed values in the range of $[0, 1K]$.

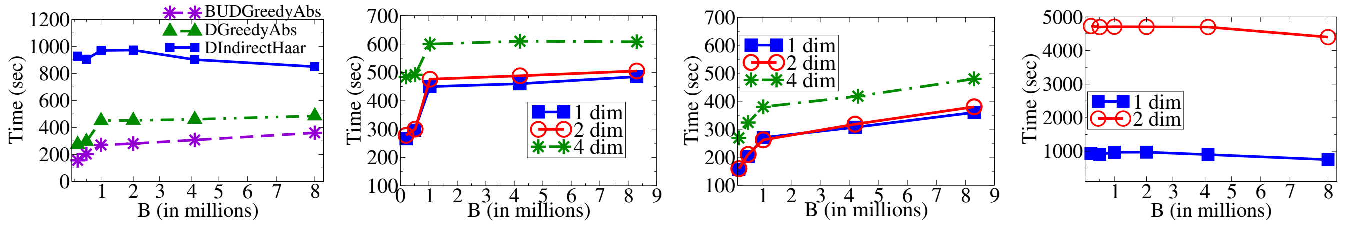
Varying space budget. In this experiment we examine the scalability with respect to the space budget B. We run *DGreedyAbs*, *BUDGreedyAbs* and *DIndirectHaar* for one-dimensional data of size $N = 17M$ and we vary B from $N/64$ to $N/2$. The results of Fig. 9a show that for *DGreedyAbs*, running-time is not considerably affected by the size of the synopsis. However, this is not true for *DIndirectHaar* and *BUDGreedyAbs*. For *DIndirectHaar*, a larger B is more probable to lead in a smaller error and decrease the $\frac{\varepsilon}{\delta}$ factor of its complexity formula. Thus, a larger budget may lead to faster execution of the algorithm. For *BUDGreedyAbs*, as the complexity of the reducer is $O\left(Rlog^2R + RB\right)$, running-time can linearly increase with B. We repeat the experiment, this time considering datasets of multiple dimensions. Fig. 9b, 9c and 9d show the results for *DGreedyAbs*, *BUDGreedyAbs* and *DIndirectHaar* respectively. As expected, for all algorithms, the higher the dataset dimension is, the higher is the running-time of the algorithm. For the greedy algorithms and for budget sizes smaller than the partition size of the distributed job (1M datapoints), we observe better running-times. This is due to an optimization where each worker emits only the B most important coefficients to the reduce stage. For Fig. 9d, we note that the complexity of *DIndirectHaar* for a 4-dimensional dataset is too high and the algorithm is not able to run.

Varying datasize and number of parallel tasks. Fig. 10 shows the scalability with respect to the number of datapoints (N) and tasks running in parallel for *DIndirectHaar*, *DGreedyAbs* and *BUDGreedyAbs* respectively. We set $B = 1M$ for all the experiments of this subsection and vary the datasize from 2M to 268M datapoints for all the algorithms and the number of parallel map tasks from 10 to 40. We also compare both algorithms with the corresponding centralized implementations in order to assess the difference in performance. We note that the y-axis in Fig. 10a, 10b and 10c follows a logarithmic scale.

All the 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 almost doubled for all the examined algorithms.

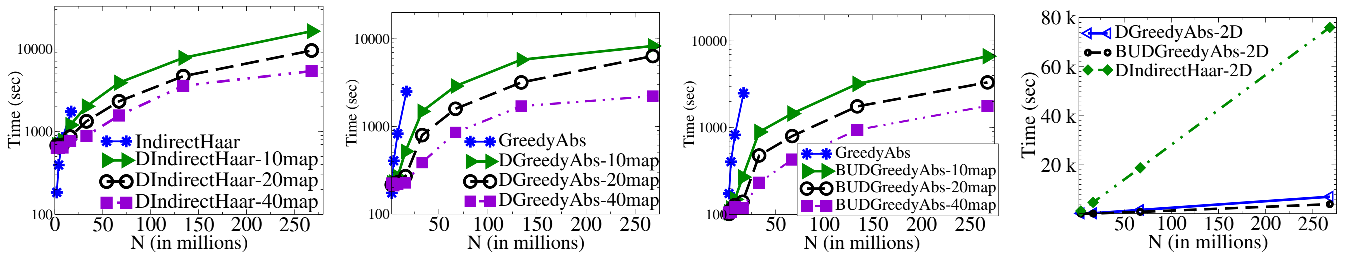
The centralized algorithms were not able to run for datasizes greater than $17M$ datapoints, as their execution demands more than the available main memory. Compared to the centralized *GreedyAbs*, *BUDGreedyAbs* appears to be $20\times$ faster for a dataset of 17M datapoints when all of its map tasks can run fully in parallel. In Fig. 10b, 10c we also observe that *BUDGreedyAbs* is twice as fast as *DGreedyAbs*. This is because *DGreedyAbs* needed to try two C_{root} sets in order to converge, while *BUDGreedyAbs* always needs a single MapReduce job. As we notice in Fig. 10a, even if *DIndirectHaar* scales linearly, it is slower than the greedy algorithms, being $1.5\times$ and $3\times$ slower than *DGreedyAbs* and *BUDGreedyAbs* respectively. Moreover, we see that the centralized *IndirectHaar* is faster than *DIndirectHaar* when the dataset size is small or few parallel tasks are running. That is because the centralized implementation loads the whole dataset in memory and the required multiple jobs do not need to perform I/O operations.

Fig. 10d presents scalability results when two-dimensional datasets of different sizes are used. Once again, all examined algorithms scale linearly with the dataset size. The important observation here is that the running-time gain of the greedy algorithms compared to *DIndirectHaar* increases along with the dimensional-



(a) 1-dimensional data (b) *DGreedyAbs* (c) *BUDGreedyAbs* (d) *DIndirectHaar*

Fig. 9: Scalability with the space budget B.



(a) *DIndirectHaar* (b) *DGreedyAbs* (c) *BUDGreedyAbs* (d) 2-dimensional datasets

Fig. 10: Scalability with the dataset size (N) and number of parallel tasks

ity. For the 2-dimensional datasets, the greedy algorithms present almost the same performance with the 1-dimensional case while *DIndirectHaar* becomes considerably slower.

The main results of this Section are that: (i) all distributed algorithms scale linearly with the datasize, (ii) greedy algorithms are much faster than the state-of-the-art DP, (ii) dimensionality positively affects running-time and (iv) the higher the dimensionality, the higher is the benefit of using a greedy algorithm.

7.2 Data Dimensionality and Maximum Absolute Error

In this Section, we investigate how dimensionality affects maximum absolute error and what tradeoffs *DIndirectHaar* offers for its high running-time. For this experiment, we consider synthetic datasets of 1, 2 and 4 dimensions and $B = \frac{N}{16}$. All datasets follow a zipfian-1.5 distribution, with a size of $N = 17M$ datapoints. The choice of distribution is inline with previous research [28], as it has been shown that wavelets can better capture skewed distributions.

In Fig. 11, we notice that the achieved accuracy is negatively affected by an increase in dimensionality. The higher the dimensionality, the higher is the observed error. This is probably an effect of the enhanced locality in high-dimensional spaces. Furthermore, *DIndirectHaar* compensates for its high computational complexity with an error 30% smaller than the one achieved by the greedy algorithms for both high dimensional datasets. However, once again it was not able to run for the 4-dimensional dataset.

7.3 Comparison for Real Datasets

In this Section, we compare *DGreedyAbs*, *BUDGreedyAbs* and *DIndirectHaar* with each other, as well as with their centralized counterparts using real-life one-dimensional datasets. Furthermore, we compare them against *CON* [31]: an algorithm that constructs a conventional synopsis (i.e., L_2-optimal). As *CON* is less compute-intensive, we want to investigate the tradeoffs in running-time and produced maximum error. For the approximation quality experiments, we do not include *IndirectHaar*, as it theoretically achieves the same results as *DIndirectHaar*.

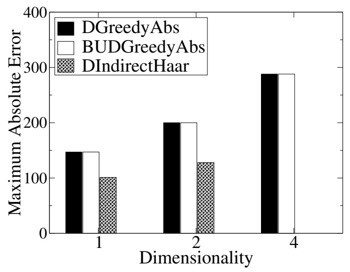
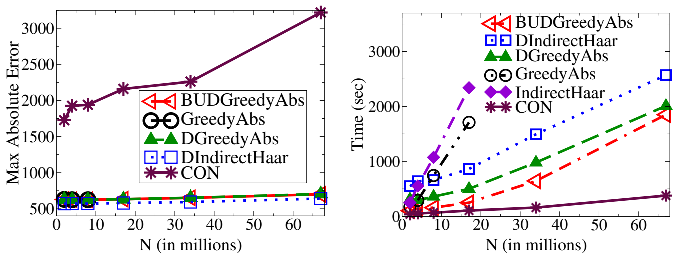
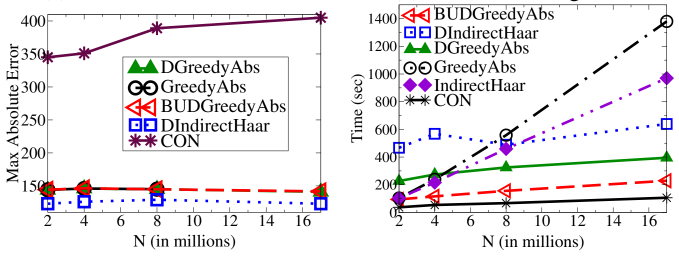


Fig. 11: Maximum Absolute Error for Zipfian data and $B = N/16$.



(a) NYCT-Max Abs Error (b) NYCT-Running-time



(c) WD-Max Abs Error (d) WD-Running-time

Fig. 12: 1-D real datasets. $B = N/8$

NYCT dataset. In Fig. 12a, we present the approximation quality results for the NYCT dataset and $B = \frac{N}{8}$. The construction of an accurate synopsis for this dataset is a difficult task to accomplish as it contains values of high magnitude and variance.

Two important observations are that: (i) scalability does not come with a cost; the distributed greedy algorithms achieve the same error with *GreedyAbs* and (ii) all algorithms targeting maximum error metrics outperform *CON* from 2 to 5 times.

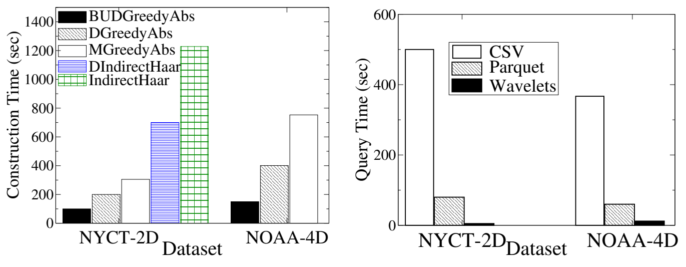
Fig. 12b presents the running-time results for the same dataset. With the maximum absolute error over 550 for all datasizes, the multiplicative factor $\left(\frac{\mathscr{E}}{\delta}\right)^2$ of the complexity formula of *DIndirectHaar* is equal to 121. As such, for this dataset, the execution of the DP algorithms is very compute-intensive. We observe that for datasizes smaller than 60M datapoints, *BUDGreedyAbs* is the most time-efficient algorithm among the ones that target maximum error metrics. In Section 7.1, we said that the available budget affects the running-time performance of *BUDGreedyAbs*. Since we have set $B = N/8$, an increase in the number of datapoints implies an increase in the synopsis' size which in turn increases the running-time of the algorithm. At this point, we observe a trade-off between *DGreedyAbs* and *BUDGreedyAbs*. On the one hand, *DGreedyAbs* needs multiple passes over the data in the map phase of the job, while *BUDGreedyAbs* needs only one. On the other hand, *DGreedyAbs* has a lightweight reducer, while the one of *BUDGreedyAbs* is compute-intensive and can become a bottleneck. Thus, when datasize is large, we suggest *BUDGreedyAbs* for datasets that can be easily approximated with a small available budget and *DGreedyAbs* when a higher budget is demanded. As the conventional synopsis is easier to be computed, we observe *CON* to be much faster than all the other algorithms.

WD dataset. Fig. 12c shows the approximation quality and running-time results for the WD dataset and $B = \frac{N}{8}$. The conclusions are similar to the ones for the NYCT dataset. In Fig. 12d we see that *IndirectHaar* outperforms *DIndirectHaar* for datasizes up to 8M datapoints. When data fits in main memory, *IndirectHaar* avoids the I/O overhead of the multiple MapReduce jobs, that *DIndirectHaar* requires. Still, the most efficient algorithm, that targets the minimization of maximum error metrics, is *BUDGreedyAbs* as it outperforms *GreedyAbs* by a factor of 6.7 and *DGreedyAbs* by a factor of 2 for a 17M dataset.

We now extend our evaluation to multidimensional real datasets. For the multidimensional experiments, we use the NYCT2D and NOAA datasets. Furthermore, in order to demonstrate the merits of wavelet thresholding in exploratory analysis tasks, we also present a query-time evaluation for the constructed synopses. For answering queries on wavelet synopses, we have implemented the work of [10]. As proposed there, instead of applying the wavelet transform directly on the data, we first construct a datacube of joint frequencies. After the synopsis is constructed, it can be loaded in main memory and provide in-memory query answering.

Fig. 13a presents the results of the construction time comparison when $B = N/16$. For both datasets, *BUDGreedyAbs* is the most time-efficient algorithm. DP algorithms are able to run only for the NYCT2D dataset, where *IndirectHaar* is 12× and *DIndirectHaar* 7× slower than *BUDGreedyAbs*. Despite the dimensionality of these datasets, we observe that all algorithms achieve lower running-times than the ones achieved in Fig. 12b. This may seem counter-intuitive, but the explanation lies behind the distribution of the wavelet transform. The transforms of NOAA and NYCT2D are sparse enough and the data that the thresholding algorithms actually process are fewer than the original dataset.

Regarding quality guarantees, all greedy algorithms produced a maximum absolute error of 1.8 and 0.9 for the NYCT2D and NOAA datasets respectively. As the errors are already small enough, the DP algorithms could not yield an interesting tradeoff for the high-running time they present.



(a) Synopsis Construction Time (b) Query time

Fig. 13: Synopsis Construction and Query Time for real-life datasets.

Fig. 13b shows the results of the query time experiment. We consider queries of the form ***select*** $\{sum, count, avg\}$ ***from T where*** $p_1 \wedge ... \wedge p_k$, where p_i is an inequality predicate. For each dataset, we run a workload of 10 random queries of that form and present the average query time. We compare query time on wavelet synopses against SparkSQL [40] queries on raw csv and Parquet [41] files. The csv text files do not fit in the aggregate memory of the cluster we have configured and thus they produce the worst query latencies. As Parquet enables lossless compression mechanisms, the corresponding Parquet files fit in our cluster's memory and improve a lot on the observed query time. However, our wavelet synopses with $B = N/16$ can fit in a single machine's main memory and thus present the best query times.

The main conclusions from the comparisons in this Section are that: (i) The proposed distributed approaches scale to datasizes that the traditional centralized algorithms are unable to process. (ii) The most time-efficient algorithms are *BUDGreedyAbs* and *DGreedyAbs* and each of these algorithms can be the most appropriate choice in a different use-case; when B is not too large, the *BUDGreedyAbs* algorithm is suggested. (iii) *DIndirectHaar* produces results of better quality but it presents the worse running-time and ends up to be impracticable in higher dimensions.

8 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 focus on designing algorithms with linear scalability over scale-out infrastructures. We first present a novel technique that allows the parallel execution of all the existing DP algorithms for the problem and show that it works for both one- and multi-dimensional datasets. Our results indicate that we can scale DP algorithms to data sizes that their centralized counterparts are incapable of processing. Moreover, in order to further improve on the running-time for the synopsis construction, we propose *DGreedyAbs* and *BUDGreedyAbs*, which are new heuristic-based algorithms based on *GreedyAbs*. These greedy algorithms are more time-efficient than the state-of-the-art DP algorithm and we also show that the performance gain they offer increases along with the dimensionality.

References

[1] 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*. ACM, 2013, pp. 29–42.

[2] B. Mozafari, "Verdict: A system for stochastic query planning." in *CIDR*, 2015.

[3] 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*. ACM, 2015, pp. 1941–1953.

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

[5] P. B. Gibbons and Y. Matias, "New sampling-based summary statistics for improving approximate query answers," in *ACM SIGMOD Record*, vol. 27, no. 2. ACM, 1998, pp. 331–342.

[6] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy, "Join synopses for approximate query answering," in *ACM SIGMOD Record*, vol. 28, no. 2. ACM, 1999, pp. 275–286.

[7] Y. E. Ioannidis and V. Poosala, "Histogram-based approximation of set-valued query-answers," in *VLDB*, vol. 99, 1999, pp. 174–185.

[8] P. B. Gibbons, Y. Matias, and V. Poosala, "Fast incremental maintenance of approximate histograms," in *VLDB*, vol. 97, 1997, pp. 466–475.

[9] H. V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. C. Sevcik, and T. Suel, "Optimal histograms with quality guarantees," in *VLDB*, vol. 98, 1998, pp. 275–286.

[10] 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*, vol. 10, no. 2-3, pp. 199–223, 2001.

[11] 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*. ACM, 2004, pp. 166–176.

[12] 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*. VLDB Endowment, 2005, pp. 421–432.

[13] 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*. ACM, 2007, pp. 380–389.

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

[15] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss, "Surfing wavelets on streams: One-pass summaries for approximate aggregate queries," in *VLDB*, vol. 1, 2001, pp. 79–88.

[16] 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*. ACM, 1996, pp. 20–29.

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

[18] T. Li, Q. Li, S. Zhu, and M. Ogihara, "A survey on wavelet applications in data mining," *ACM SIGKDD Explorations Newsletter*, vol. 4, no. 2, pp. 49–68, 2002.

[19] Y. Matias, J. S. Vitter, and M. Wang, "Wavelet-based histograms for selectivity estimation," in *ACM SIGMOD Record*, vol. 27, no. 2. ACM, 1998, pp. 448–459.

[20] J. S. Vitter and M. Wang, "Approximate computation of multidimensional aggregates of sparse data using wavelets," in *ACM SIGMOD Record*, vol. 28, no. 2, 1999.

[21] 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*, vol. 15, no. 3, pp. 541–554, 2003.

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

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

[24] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine, "Synopses for massive data: Samples, histograms, wavelets, sketches," *Foundations and Trends in Databases*, vol. 4, no. 1–3, pp. 1–294, 2012.

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

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

[27] 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*. ACM, 2009, pp. 732–743.

[28] M. Garofalakis and A. Kumar, "Wavelet synopses for general error metrics," *ACM Transactions on Database Systems (TODS)*, vol. 30, no. 4, pp. 888–928, 2005.

[29] S. Guha and B. Harb, "Wavelet synopsis for data streams: minimizing non-euclidean error," in *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*. ACM, 2005, pp. 88–97.

[30] Y. Matias and L. Portman, "Workload-based wavelet synopses," Technical report, Department of Computer Science, Tel Aviv University, Tech. Rep., 2003.

[31] I. Mytilinis, D. Tsoumakos, and N. Koziris, "Distributed wavelet thresholding for maximum error metrics," in *Proceedings of the 2016 International Conference on Management of Data*. ACM SIGMOD, 2016, pp. 663–677.

[32] J. Jestes, K. Yi, and F. Li, "Building wavelet histograms on large data in mapreduce," *Proceedings of the VLDB Endowment*, vol. 5, no. 2, pp. 109–120, 2011.

[33] P. Karras and N. Mamoulis, "Hierarchical synopses with optimal error guarantees," *ACM Transactions on Database Systems (TODS)*, vol. 33, no. 3, p. 18, 2008.

[34] Q. Zhang, C. Pang, and D. Hansen, "On multidimensional wavelet synopses for maximum error bounds," in *International Conference on Database Systems for Advanced Applications*. Springer, 2009, pp. 646–661.

[35] X. Li, S. Huang, H. Zhao, X. Guo, L. Xu, X. Li, and Y. Li, "Image compression based on restcted wavelet synopses with maximum error bound," in *Proceedings of the 9th International Conference on Utility and Cloud Computing*, ser. UCC '16. New York, NY, USA: ACM, 2016, pp. 333–338. [Online]. Available: http://doi.acm.org/10.1145/2996890.3007880

[36] A. Wasay, X. Wei, N. Dayan, and S. Idreos, "Data canopy: Accelerating exploratory statistical analysis," in *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 2017, pp. 557–572.

[37] "Nyct," http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml.

[38] "Linked sensor data," https://wiki.knoesis.org/index.php/SSW_Datasets.

[39] "National oceanic and atmospheric administration," https://www1.ncdc.noaa.gov/pub/data/noaa/.

[40] "Spark SQL," https://spark.apache.org/sql/.

[41] "Parquet," https://parquet.apache.org/.

Ioannis Mytilinis is a Ph.D. candidate at the Computing Systems Laboratory of the National Technical University of Athens (NTUA). His research lies in the fields of Large Scale Data Management, Distributed Systems and Cloud Computing. He received his Diploma in Electrical and Computer Engineering from NTUA in 2012. He is a member of ACM SIGMOD and of Technical Chamber of Greece. For more: http://www.cslab.ece.ntua.gr/~gmytil.

Dimitrios Tsoumakos is an Associate Professor in the Department of Informatics of the Ionian University. He is also collaborating with the Computing Systems Laboratory of the National Technical University of Athens (NTUA). He received his Diploma in Electrical and Computer Engineering from NTUA in 1999, then joined the graduate program in Computer Sciences at the University of Maryland in 2000, where he received his M.Sc. (2002) and Ph.D. (2006). For more: http://www.cslab.ece.ntua.gr/~dtsouma.

Nectarios Koziris is a Professor of Computer Science in the School of Electrical and Computer Engineering at the National Technical University of Athens. His research interests include parallel and distributed systems, interaction between compilers, OS and architectures, scalable data management and large scale storage systems. He is a member of the IEEE Computer Society, senior member of the ACM, elected chair of the IEEE Greece Section and started the IEEE Computer Society Greece. For more: http://www.cslab.ece.ntua.gr/~nkoziris.