The VLDB Journal manuscript No.
(will be inserted by the editor)

Hierarchically Compressed Wavelet Synopses

Dimitris Sacharidis · Antonios Deligiannakis · Timos Sellis

the date of receipt and acceptance should be inserted later

Abstract The wavelet decomposition is a proven tool for constructing concise synopses of large data sets that can be used to obtain fast approximate answers. Existing research studies focus on selecting an optimal set of wavelet coefficients to store so as to minimize some error metric, without however seeking to reduce the size of the wavelet coefficients themselves. In many real data sets the existence of large spikes in the data values results in many large coefficient values lying on paths of a conceptual tree structure known as the error tree. To exploit this fact, we introduce in this paper a novel compression scheme for wavelet synopses, termed Hierarchically Compressed Wavelet Synopses, that fully exploits hierarchical relationships among coefficients in order to reduce their storage. Our proposed compression scheme allows for a larger number of coefficients to be stored for a given space constraint thus resulting in increased accuracy of the produced synopsis. We propose optimal, approximate and greedy algorithms for constructing hierarchically compressed wavelet synopses that minimize the sum squared error while not exceeding a given space budget. Extensive experimental results on both synthetic and real-world data sets validate our novel compression scheme and demonstrate the effectiveness of our algorithms against existing synopsis construction algorithms.

This work has been funded by the project PENED 2003. The project is co-financed 75% of public expenditure through EC - European Social Fund, 25% of public expenditure through Ministry of Development - General Secretariat of Research and Technology and through private sector, under measure 8.3 of OPERATIONAL PROGRAMME "COMPETITIVENESS" in the 3rd Community Support Programme.

D. Sacharidis · T. Sellis
National Technical University of Athens
E-mail: {dsachar,timos}@dblab.ntua.gr

A. Deligiannakis
University of Athens
E-mail: adeli@di.uoa.gr

Keywords wavelet synopsis · data streams · compression

1 Introduction

Approximate query processing over compact precomputed data synopses has attracted a lot of attention recently as an effective means of dealing with massive data sets in interactive decision support and data exploration environments. In such settings, users typically pose complex queries, which require considerable amounts of time to produce exact answers, over large parts of the stored data. In exploratory queries of such nature, users can often tolerate small imprecisions in query results, as long as these results are quickly generated and fairly accurate.

The wavelet decomposition is a mathematical tool for the hierarchical decomposition of functions with a long history of successful applications in signal and image processing [17,24,26]. Several studies have demonstrated the applicability of wavelets as a data reduction tool for a variety of database problems. Briefly, the key idea is to first apply the decomposition process over an input data set, thus producing a set of wavelet coefficients. We then retain only a subset, composing the *wavelet synopsis*, of the coefficients by performing a thresholding procedure. This thresholding process depends on the desired minimization metric. The bulk of past research [26,28,29], both in databases as well as in image and signal processing applications, has focused on the minimization of the sum squared error of the produced approximation. Recent approaches have targeted the minimization of additional error metrics, such as the maximum absolute/relative error [8,9], or the weighted sum squared relative error [12,23,27] of the approximation.

Independently of the targeted minimization metric, the selected coefficients are stored as pairs $\langle Coords, Value \rangle$, where the first element (*Coords*) is the *coordinates/index* of the coefficient and determines the data that this coefficient helps reconstruct (also termed as the *support region* of the coefficient), while the second element (*Value*) denotes the magnitude/value of the coefficient. Depending on the actual storage representation for these elements (i.e., integer values for the coordinates and floating point numbers for the coefficient value) and the data dimensionality, the fraction of the available storage for the synopsis that is used for storing coefficient coordinates can be significant. If sizeof$(Coord)$ and sizeof$(Value)$ denote the storage requirements for the coefficient coordinates[1] and the coefficient value, correspondingly, then the storage of the coefficient coordinates will occupy a fraction $\frac{\text{sizeof}(Coord)}{\text{sizeof}(Coord)+\text{sizeof}(Value)}$ of the overall synopsis size (see Section 2).

While reducing the storage overhead of the wavelet coordinates would allow for a larger number of coefficient values to be stored, and would thus result in increased accuracy of the synopsis, to our knowledge none of the above techniques tries to exploit this fact and incorporate it in the coefficient thresholding process. A past suggestion [1] has proposed constructing a linear approximation method with respect to the wavelet orthonormal basis [19] by selecting for storage only the top coefficient values (i.e., the ones with the largest support regions). Using such an approach, no coordinates need to be stored. However, such an approach does not give any guarantee on whether the selected coefficients can significantly reduce the desired error metric. Finally, techniques that target, possibly multi-dimensional, data sets with multiple measures [6, 15] exploit storage dependencies among only coefficient values that correspond to the same coordinates, but for different measures.

To address the drawbacks of existing techniques, in this paper we propose a novel, flexible, compression scheme, termed *Hierarchically Compressed Wavelet Synopses* (denoted as HCWS), for storing wavelet coefficients. In a nutshell, instead of individually storing wavelet coefficients, our compression scheme allows for storing sets of coefficient values. These stored sets are not arbitrary, but are rather composed by coefficients that lie on a path of a conceptual tree-like structure, known as the *error tree*, that captures the hierarchical relationship amongst wavelet coefficients. While its formal description is deferred for Section 2, a sample error tree is depicted in Figure 1. Each path of coefficient values stored as a hierarchically compressed wavelet coefficient (HCC) can be uniquely identified by (i) the coordinates of the path's lowest, in the error-tree, stored coefficient *LC*; and (ii) a bitmap that reveals how many ancestors of *LC* are also stored in the same HCC. Utilizing such an index-sharing setting leverages better space allocation, since the coordinates of a single coefficient need to be stored in each path, which can result to increased accuracy of the obtained approximation.

A question that naturally arises is whether "important", for the desired error metric, coefficient values can frequently occur within such a path and would, thus, be beneficial to store using a HCC. As we explain in Section 2, due to the nature of the wavelet decomposition process, this behavior is expected to be frequently observed, and only, in data sets with frequent spikes and discontinuities in neighboring domain regions. These discontinuities are often due to large spikes in the collected data values, such as the ones observed in network monitoring applications where the number of network packets may often exhibit a bursting behavior. A similar behavior also occurs in sparse regions over large domain sizes, where either few non-zero data values may occur in an otherwise empty region, or where dense regions neighbor empty regions of the data.

One could argue that wavelets are ill-suited for such data sets, and that other competitive approximation techniques, such as compressed histograms [25], might be in some cases more appropriate. Our proposed techniques seek to improve the accuracy of the obtained data synopsis in such data sets, without requiring any a-priori knowledge on the overall data distribution, or on the existence, and frequency, of spikes and discontinuities in the collected data. When such spikes and discontinuities occur frequently, our techniques manage to improve the storage utilization of wavelet coefficients and, thus, the quality of the obtained approximation. Moreover, our techniques can be adapted to multi-dimensional data sets, where prior studies [2, 29] have demonstrated that the wavelet transform can lead to more accurate data representations than competitive techniques, such as Histograms.

To briefly illustrate the benefits of our approach, consider the sample error tree depicted in Figure 1. In this figure, the values of 16 coefficients are depicted, using the symbol c_i to denote the coefficient at coordinate i. Without formally introducing the internals of the conventional thresholding process, assuming a space budget of 41 bytes, and using 8 bytes for storing the$\langle Coord, Value\rangle$ pairs, the optimal conventional wavelet synopsis would simply store the coefficients c_0, c_1, c_7, c_8 and c_{15} shown in gray. On the other hand, our hierarchically compressed wavelet synopsis, given the same space budget, would store the two paths shown in Figure 1 — that is, it would manage to also store coefficients c_2, c_3, c_5 and c_{11} in comparison to the coefficient c_8 selected by the conventional wavelet synopsis. The effect of including these coefficients is the reduction of the sum squared error (SSE) of the approximation by 60% (SSE of 294 instead of 752).

While the notion of HCWS can be used as a storage technique by optimization algorithms and incorporated in their

[1] While for a D-dimensional data set, the D coefficient coordinates could be stored uncompressed, alternative encodings can be utilized to limit their size. For example, utilizing a location function for arrays, the D-dimensional coefficient coordinates can be encoded with space that depends on the product of the dimension cardinalities.

operation for any of the proposed error metrics, in this paper we simply focus on minimizing the commonly used sum squared (absolute or relative) error metrics. The contributions of our work can be summarized as:

1. We introduce the concept of HCWS, a novel compression scheme that fully exploits the hierarchical relationships among wavelet coefficients, and that may lead to significant accuracy gains.
2. We propose a novel, optimal dynamic programming algorithm, HCDynL2, for selecting the HCWS that minimizes the sum of squared errors under a given synopsis size budget. We then propose a streaming variant of the optimal algorithm that can operate in one pass over the data using limited memory.
3. We present an approximation algorithm, HCApprL2, with tunable guarantees, for the *benefit* of the obtained solution, for the same optimization problem. Further, we present a streaming variant of the algorithm.
4. Due to the large running time and space requirements of our DP solution, we introduce a fast greedy, HCGreedyL2, algorithm with space and time requirements on par with conventional synopsis techniques. We then also present a streaming variant, the HCGreedyL2-Str algorithm, of the greedy algorithm.
5. We sketch useful extensions for multi-dimensional data sets and running time improvements for large domain sizes.
6. We present extensive experimental results of our algorithms on both synthetic and real-life data sets. Our experimental study demonstrates that (i) the use of HCWS can lead to wavelet synopses with significantly reduced errors; (ii) HCApprL2 constructs HCWS with tunable accuracy guarantees; (iii) although HCGreedyL2 cannot provide guarantees in the quality of the obtained synopsis, it always provides near-optimal solutions, while exhibiting very fast running times; and (iv) The HCGreedyL2-Str algorithm consistently provides results comparable to those of the HCGreedyL2 algorithm.

Outline. The remainder of this paper is organized as follows. Section 2 builds the necessary background on wavelet decomposition, introduces the concept of Hierarchically Compressed Wavelet Synopses and formally presents our optimization problem. In Section 3 we formulate a dynamic programming recurrence and use it to optimally solve this optimization problem. Next, in Section 4 we present an approximation algorithm with tunable guarantees, whereas, in Section 5 we present a faster greedy algorithm. In Section 6 we provide a streaming version of our greedy algorithm. In Section 7 we sketch some useful extensions of our algorithms and in Section 8 we describe the results of our empirical study. Section 9 presents related work and, finally, Section 10 provides some concluding remarks and future directions.

2 Preliminaries

In this section, we provide a quick introduction to the simplest of wavelet decompositions, the Haar wavelet decomposition. We also discuss the existing strategies for coefficient thresholding and demonstrate some of their important shortcomings. Finally, we introduce the notion of *hierarchically compressed wavelet coefficients and synopses*, which form the basis for our proposed approach and data-reduction algorithms.

2.1 One-Dimensional Haar Wavelets

Wavelets are a useful mathematical tool for hierarchically decomposing functions in ways that are both efficient and theoretically sound. Broadly speaking, the wavelet decomposition of a function consists of a coarse overall approximation along with detail coefficients that influence the function at various scales [26]. Suppose we are given the one-dimensional data vector A containing the $N = 16$ data values $A = [17, 41, 32, 30, 36, 36, 35, 57, 0, 0, 0, 0, 0, 0, 0, 36]$. The Haar wavelet transform of A can be computed as follows. We first average the values together pairwise to get a new "lower-resolution" representation of the data with the following average values [29, 31, 36, 46, 0, 0, 0, 18]. In other words, the average of the first two values (that is, 17 and 41) is 29, that of the next two values (that is, 32 and 30) is 31, and so on. Obviously, some information has been lost in this averaging process. To be able to restore the original values of the data array, we store some *detail coefficients* that capture the missing information. In Haar wavelets, these detail coefficients are simply the differences of the (second of the) averaged values from the computed pairwise average. Thus, in our simple example, for the first pair of averaged values, the detail coefficient is -12 since $29 - 41 = -12$, for the second we again need to store 1 since $31 - 30 = 1$. Recursively applying the above pairwise averaging and differencing process on the lower-resolution array containing the averages, we get the following full decomposition: The

Resolution	Averages	Detail Coefficients
4	[17, 41, 32, 30, 36, 36, 35, 57, 0, 0, 0, 0, 0, 0, 0, 36]	—
3	[29, 31, 36, 46, 0, 0, 0, 18]	[-12, 1, 0, -11, 0, 0, 0, -18]
2	[30, 41, 0, 9]	[-1, -5, 0, -9]
1	[35.5, 4.5]	[-5.5, -4.5]
0	[20]	[15.5]

wavelet transform (also known as the *wavelet decomposition*) of A consists of the single coefficient representing the

Symbol	Description $(i \in \{0,\ldots,N-1\})$
N	Number of data-array cells
D	Data-array dimensionality
B	Space budget for synopsis
A, W_A	Input data and wavelet transform arrays
d_i	Data value for i^{th} cell of data array
$\hat{d_i}$	Reconstructed data value for i^{th} cell
c_i, c_i^*	Un-normalized/normalized Haar coefficient coordinate i
$\text{path}(u)$	Set of non-zero proper ancestors of u in the error tree
$\text{level}(c_i)$	The level of the error tree c_i belongs to
HCC	A hierarchically compressed wavelet coefficient
$\text{bottom}(HCC)$	The bottommost coefficient that belongs to HCC
$\text{top}(HCC)$	The topmost coefficient that belongs to HCC
$\text{parent}(HCC)$	The parent of the topmost coefficient that belongs to HCC

Table 1 Notation.

overall average of the data values, followed by the detail coefficients in the order of increasing resolution. Thus, the one-dimensional Haar wavelet transform of A is given by $W_A = [20, 15.5, -5.5, -4.5, -1, -5, 0, -9, -12, 1, 0, -11, 0, 0, 0, -18]$. Each entry in W_A is called a *wavelet coefficient*. The main advantage of using W_A instead of the original data vector A is that for vectors containing similar values most of the detail coefficients tend to have very small values. Thus, eliminating such small coefficients from the wavelet transform (i.e., treating them as zeros) introduces only small errors when reconstructing the original data, resulting in a very effective form of lossy data compression [26].

The Haar Coefficient Error Tree. A helpful tool for exploring and understanding the key properties of the Haar wavelet decomposition is the *error tree* structure [21]. The error tree is a hierarchical structure built based on the wavelet transform process (even though it is primarily used as a conceptual tool, an error tree can be easily constructed in linear $O(N)$ time). Figure 1 depicts the error tree for our example data vector A. Each internal node c_i $(i = 0,\ldots,15)$ is associated with a wavelet coefficient value, and each leaf d_i $(i = 0,\ldots,15)$ is associated with a value in the original data array; in both cases, the index/coordinate i denotes the positions in the data array or error tree. For example, c_0 corresponds to the overall average of A. Note that average coefficients are shown with square nodes (data values can be considered as averages at level $\log N$), whereas details are shown with circular nodes. The resolution levels l for the coefficients (corresponding to levels in the tree) are also depicted. (We use the terms "node" and "coefficient" interchangeably in what follows.) Table 1 summarizes some of the key notational conventions used in this paper; additional notation is introduced when necessary. Detailed symbol definitions are provided at the appropriate locations in the text. Given a node u in an error tree T, let $\text{path}(u)$ denote the set of all proper ancestors of u in T (i.e., the nodes on the path from u to the root of T, including the root but not u) with non-zero coefficients. A key property of the Haar wavelet decomposition is that the reconstruction of any data value d_i depends only on the values of coefficients on $\text{path}(d_i)$; more specifically, we have $d_i = \sum_{c_j \in \text{path}(d_i)} \delta_{ij} \cdot c_j$, where $\delta_{ij} = +1$ if d_i is in the left child subtree of c_j or $j = 0$, and $\delta_{ij} = -1$ otherwise. Reconstructing any data value involves summing at most $\log N + 1$ coefficients. For example, in Figure 1, $d_5 = c_0 + c_1 - c_2 + c_5 = 20 + 15.5 - (-5.5) + (-5) = 36$. Note that, intuitively, wavelet coefficients carry different weights with respect to their importance in rebuilding the original data values. For example, the overall average and its corresponding detail coefficient are obviously more important than any other coefficient since they affect the reconstruction of all entries in the data array. In order to weigh the importance of all wavelet coefficients, we need to appropriately *normalize* the final entries of W_A. A common normalization scheme [26] is to multiply each wavelet coefficient c_i by $\sqrt{2^{\log N - \text{level}(c_i)}}$, where $\text{level}(c_i)$ denotes the *level of resolution* at which the coefficient appears (with 0 corresponding to the "coarsest" resolution level and $\log N$ to the "finest"). Given this normalization procedure, the normalized values of the wavelet coefficients of our example data array A are: $[80, 62, -11\sqrt{2}, -9\sqrt{2}, -2, -10, 0, -18, -12\sqrt{2}, \sqrt{2}, 0, -11\sqrt{2}, 0, 0, 0, -18\sqrt{2}]$.

2.2 Conventional Wavelet Synopses

Given a limited amount of storage for building a *wavelet synopsis* of the input data array A, a thresholding procedure retains a certain number $B_C \ll N$ of the coefficients in W_A as a highly-compressed approximate representation of the original data (the remaining coefficients are implicitly set to 0). The goal of coefficient thresholding is to determine the "best" subset of B_C coefficients to retain, so that some overall error measure in the approximation is minimized. The method of choice for the vast majority of studies on wavelet-based data reduction and approximation [2, 21,22] is *conventional coefficient thresholding* that greedily retains the B_C largest Haar-wavelet coefficients in *absolute normalized value*. This thresholding method *provably* minimizes the sum squared error (SSE). Indeed, in a mathematical view point, the process of computing the wavelet transform and normalizing the coefficients is actually the orthonormal transformation of the data vector with respect to the Haar basis. Parseval's formula guarantees that choosing the B_C largest coefficients is optimal with respect to the SSE. Consider our example array A and assume that we have a space budget of 41 bytes. In conventional synopses we require to store each coefficient as a $\langle i, c_i \rangle$ pair, where i denotes the index/coordinate of the coefficient and c_i denotes its value. Thus, our budget translates to 5 coefficients, if we further assume that a coordinate and a coefficient value cost 4 bytes each. Optimizing for the sum of squared errors, leads to choosing the 5 largest (in absolute normalized value) coefficients. These retained coefficients c_0, c_1, c_7, c_8 and c_{15}

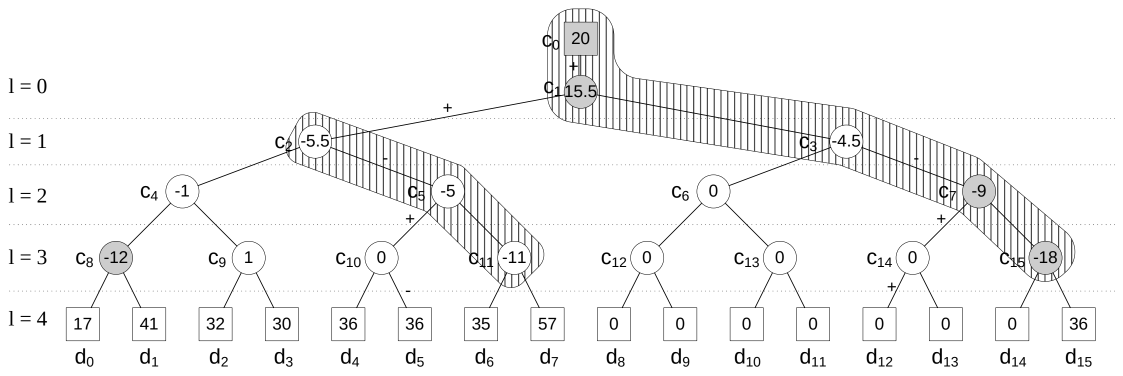


Fig. 1 Error-tree structure for example data vector A.

Coordinate	Bitmap	Set of Coefficient Values
11	11110	$\{-11, -5, -5.5, 15.5, 20\}$
15	110	$\{-18, -9, -4.5\}$

Table 2 HCWS for data vector A and $B = 41$ bytes.

are shown in gray in Figure 1. Note that in D-dimensional data sets the stored coefficients consist of the D dimension coordinates (which, as mentioned in Section 1, can be stored in less space than explicitly storing them as D integer values) and of the coefficients value.

As discussed in Section 1, the main drawback of conventional wavelet synopses for minimizing the SSE of the approximation is that not only is there no effort to reduce the storage overhead of the selected coefficients but, more importantly, that this objective is not incorporated in the operation of the algorithm. The same drawback also occurs in thresholding algorithms that try to minimize other error metrics, such as the maximum or weighted sum squared absolute/relative error of the approximation [8,9,12,23,27]. Due to the differencing process employed by the wavelet decomposition between average values of neighboring regions, multiple large coefficient values may exhibit hierarchical relationships (i.e., belong in the same path) only when spikes over some regions of the data are large enough[2] to significantly impact the values of coefficients (and, thus, generate coefficients with large values) in multiple (and potentially all) resolution levels. Data sets which include multiple spikes with the aforementioned property (i.e., can generate multiple large coefficients in their path), present great opportunity for exploiting the hierarchical relationships among important coefficient values and also provide better opportunities for our presented techniques to be most effective.

[2] Besides its magnitude, the impact of a spike may also depend, in the case of the $\mathcal{L}_2^w$ error metric discussed in Section 7.3, on the weight specified for each data point.

2.3 Hierarchically Compressed Synopses

Given the shortcomings of the existing wavelet thresholding algorithms we now introduce the notion of a *hierarchically compressed wavelet coefficient* (HCC). For ease of presentation, we initially focus on the one-dimensional case. The extensions to multi-dimensional data sets are presented in Section 7.

Definition 1 *A hierarchically compressed (HCC) wavelet coefficient is a triplet* $\langle \textsc{Bit}, C, V \rangle$ *consisting of:*

- *A bitmap* Bit *of size* $|\textsc{Bit}| \geq 1$, *denoting the storage of exactly* $|\textsc{Bit}|$ *coefficient values.*
- *The coordinate/index* C *of the bottommost stored coefficient.*
- *The set* V *of* $|\textsc{Bit}|$ *stored coefficient values.*

□

The bitmap of a *HCC* can help determine how many coefficient values have actually been stored. By representing the number of stored coefficients in unary format, as a series of $(|V| - 1)$ 1-bits and utilizing a 0-bit as the last bit (also acting as a stop bit), any hierarchically compressed wavelet coefficient that stores $|V|$ coefficient values requires a bitmap of just $|V|$ bits. A *hierarchically compressed wavelet synopsis* (HCWS) consists of a set of *HCC*s, in analogy to a conventional synopsis that comprises $\langle Coords, Value \rangle$ pairs.

Returning to our example array A, for a space budget of 41 bytes, or 328 bits, optimizing for the SSE metric results in storing two hierarchically compressed coefficients. These *HCC*s are essentially the two paths illustrated in Figure 1 and are depicted in Table 2. Assuming, as before, that a coordinate and a coefficient value each require 32 bits, the first hierarchical coefficient requires $32 + 5 + 5 \cdot 32 = 197$ bits, whereas the second one requires $32 + 3 + 3 \cdot 32 = 131$ bits.

It is easy to see how a hierarchically compressed synopsis better utilizes the available space, and in doing so manages to store 3 more coefficients than the conventional synopsis retains. In terms of SSE, the conventional synopsis loses 752, whereas the HCWS just 294 — an improvement of over 60%. It is important though to emphasize that the coefficient values stored in HCWS are not necessarily a superset of the coefficients selected by the conventional thresholding algorithm, since it is often more beneficial to exploit storage dependencies and store multiple coefficient values that lie on a common path, than storing a slightly larger individual value, as shown in Section 8. In our example, note that the c_8 coefficient, selected by a conventional synopsis, is not included in the optimal HCWS.

2.4 Problem Definition

The selection of which hierarchically compressed wavelet coefficients to store is based on the optimization problem we are trying to solve. To simplify notation, in our discussion hereafter the unit of space is set equal to 1 bit, and all space requirements are expressed in terms of this unit. The bulk of the work in wavelet-based compression of data tries to minimize the sum of squared absolute errors (SSE) of the overall approximation. We focus on the same problem, here; extensions to the sum of squared relative errors, or any weighted $\mathcal{L}_2^w$ norm, can be found in Section 7. More formally, the optimization problem can be posed as follows:

Problem 1 [Sum of Squared Errors Minimization for Hierarchically Compressed Coefficients] Given a collection W_A of wavelet coefficients and a storage constraint B select a synopsis $\mathcal{S}$ of hierarchically compressed wavelet coefficients HCC's that minimizes the sum of squared errors; that is, minimize $\sum_{i=0}^{N-1}(d_i - \hat{d}_i)^2$ subject to the constraint $\sum_{HCC \in \mathcal{S}} |HCC| \leq B$, where $|HCC|$ denotes the space requirement for storing HCC. □

Based on Parseval's theorem and the discussion in Section 2.2, using c_i^* to denote the normalized value for the i^{th} wavelet coefficient, we can restate the above optimization problem in the following equivalent (but easier to process) form.

Problem 2 [Benefit Maximization for Hierarchically Compressed Coefficients] Given a collection W_A of wavelet coefficients and a storage constraint B, select a synopsis $\mathcal{S}$ of hierarchically compressed wavelet coefficients that maximizes the sum $\sum_{i=0}^{N-1}(c_i^*)^2$ of the squared retained normalized coefficient values, subject to the constraint $\sum_{HCC \in \mathcal{S}} |HCC| \leq B$, where $|HCC|$ denotes the space requirement for storing HCC. □

Symbol	Description
S_1	$\text{sizeof}(Coords) + \text{sizeof}(Value) + 1$
S_2	$\text{sizeof}(Value) + 1$
$\text{M}[i,B]$	The optimal benefit acquired when assigning at most B space to the subtree of coefficient c_i
$\text{F}[i,B]$	The optimal benefit acquired when assigning at most B space to the subtree of coefficient c_i and when c_i is forced to be stored.

Table 3 Notation used in HCDynL2 Algorithm.

3 HCDynL2: An Optimal Dynamic-Programming Algorithm

We now propose a thresholding algorithm (termed HCDynL2) based on Dynamic-Programming (DP) ideas, that optimally solves the optimization problem described above. Our HCDynL2 algorithm takes as input a set of input coefficient values W_A and a space constraint B. HCDynL2 then selects an optimal set of hierarchically compressed coefficients for Problem 2. Before explaining the operation of our HCDynL2 algorithm, we need to introduce the notion of *overlapping* paths.

Definition 2 *Two paths are* overlapping *if they both store the value of at least one common coefficient.*

It is important to note that the benefit of storing two overlapping paths is not equal to the sum of benefits of these two paths, since the storage of at least one coefficient value is duplicated. Thus, the benefit of each path depends on which other overlapping paths are included in the optimal solution. The possibly varying benefit of each candidate path is the main difficulty in formulating an optimal algorithm. To make matters worse, the number of candidate paths that may be part of the solution is quite large ($O(N \log N)$), as is the number of overlapping paths. In particular, any coefficient value c_i belonging at level level(c_i) may be stored in up to $\sum_{\substack{0 \leq k \leq \min\{j, \text{level}(c_i)\} \\ \text{level}(c_i) - k + j \leq \log N}} 2^k$ paths of length $1 \leq j \leq \log N + 1$ (i.e., paths originating from nodes in its subtree with distance at most j from c_i). Fortunately, the following lemma helps reduce the search space of our algorithm, by considering the structure of the error tree.

Lemma 1 *The optimal solution for Problem 2 (and equivalently for Problem 1) never needs to consider overlapping paths.*

The proof of Lemma 1 is simple and is based on the observation that for any solution that includes a pair of overlapping paths (the extension to having multiple overlapping paths is straightforward), there exists an alternative solution with non-overlapping paths that stores exactly the same coefficient values and, thus, has the same benefit while requiring less space. This solution is produced by simply removing from one of the overlapping paths its intersection with

$$\mathrm{F}[i,B] = \begin{cases} -\infty & \text{, if } i \geq N \text{ or } B < S_1 \\ \max \begin{Bmatrix} \max\limits_{0 \leq b_L \leq B-S_1} (c_i^*)^2 + \mathrm{M}[2i,b_L] + \mathrm{M}[2i+1,B-b_L-S_1] \\ \max\limits_{0 \leq b_L \leq B-S_2} (c_i^*)^2 + \mathrm{F}[2i,b_L] + \mathrm{M}[2i+1,B-b_L-S_2] \\ \max\limits_{0 \leq b_L \leq B-S_2} (c_i^*)^2 + \mathrm{M}[2i,b_L] + \mathrm{F}[2i+1,B-b_L-S_2] \end{Bmatrix} & \text{, otherwise} \end{cases} \qquad \mathrm{M}[i,B] = \begin{cases} 0 & \text{, if } i \geq N \text{ or } B < S_1 \\ \max \begin{Bmatrix} \max\limits_{0 \leq b_L \leq B} \mathrm{M}[2i,b_L] + \mathrm{M}[2i+1,B-b_L] \\ \mathrm{F}[i,B] \end{Bmatrix} & \text{, otherwise} \end{cases} \quad (1)$$

the other path. Let the storage overhead cost of the coefficient coordinate be assigned to the lowest coefficient of each path. Thus, the required space for this coefficient (i.e., the "start-up" cost for any HCC) is $S_1 = \text{sizeof}(Coord) + \text{sizeof}(Value) + 1$ and the corresponding space for all other coefficient values in its path is simply: $S_2 = \text{sizeof}(Value) + 1$. Then, when considering the optimal solution at any node $i \geq 1$ (The extension to node 0 that has just one subtree is straightforward) of the error tree, given any space constraint B, the following cases may arise:

1. Coefficient c_i is not part of the optimal solution. The optimal solution arises from the best allotment of the space B to the two subtrees of c_i.
2. Coefficient c_i is part of the optimal solution but is not a part of any hierarchically compressed path originating from any of its descendants in the error tree. The optimal solution arises from storing c_i in a new hierarchically compressed path and considering the best allotment of the space $B - S_1$ to the two subtrees of c_i.
3. Coefficient c_i is part of the optimal solution and is part of a single hierarchically compressed path originating from one of its descendants that may reside in its left (right) subtree. The optimal solution arises from attaching c_i to the hierarchically compressed path of the left (right) subtree and considering the best allotment of the space $B - S_2$ to the two subtrees of c_i. However, for this space distribution process to be valid, we need to make sure that the solution that is produced by allocating space $0 \leq b \leq B - S_2$ to the left (right) subtree stores the coefficient c_{2i} (c_{2i+1}) — otherwise, c_i cannot be attached to a path originating from that subtree.

Cases 1 and 2 are pretty straightforward, since they introduce a recursive procedure that can be used to calculate the optimal solution at node i. This recursive procedure will check all possible allocations of space to the two subtrees of i and calculate the optimal solutions in these subtrees, given the space allocated to them. The optimal solution arises from the space allocation that results in the largest benefit. Note that in these two cases there are no dependencies or requirements from the solutions sought in the two subtrees, other that they result in the largest possible benefit, given the space allocated to them (and thus seeking the optimal solutions in these subtrees suffices).

On the contrary, in Case 3 coefficient c_i, for any space allocation to its two subtrees, needs to be attached to a solution that is produced at one of its subtree **and** where this solution stores the coefficient value at the root of the subtree. Given this requirement, the solution for this subtree is not necessarily the optimal one, but only the optimal solution, given that the root of the subtree is stored. This implies that our algorithm will need to also keep track of some suboptimal solutions, similarly to the dynamic programming algorithm in [6], which seeks to exploit storage dependencies in data sets with multiple measures *only* among coefficient values of different measures that share the same coefficient coordinates (and, thus, cannot be used for the problem addressed in this paper). On the other hand, the goal of our algorithm is to explore hierarchical relationships among coefficient values of different coordinates in order to reduce their storage overhead and improve their storage utilization in single-measure data sets. This requires properly utilizing the error-tree structure to identify these storage dependencies and processing the nodes in the error tree using an appropriate ordering. Neither of these restrictions was present in [6].

3.1 Our Solution

We now formulate a dynamic programming (DP) solution for the optimization problem of Section 2.4; the notation used is shown in Table 3. Let $\mathrm{M}[i,B]$ denote the maximum benefit acquired when assigning at most space B to the subtree of coefficient c_i. Also, let $\mathrm{F}[i,B]$ denote the optimal benefit acquired when assigning at most space B to the subtree of coefficient c_i and when c_i is *forced* to be stored. Equation 1 depicts the recurrences employed by our HCDynL2 algorithm in order to calculate these values. Case 2, discussed above, corresponds to the first clause of the *max* calculation for $\mathrm{F}[i,B]$, while Case 3 is covered by the next two clauses of the same *max* calculation. Of course, when the remaining space is less than S_1 or $i \geq N$, it is infeasible [3] to store the coefficient value c_i, thus returning a benefit of $-\infty$. For the calculation of the $\mathrm{M}[i,B]$ value, Case 1 is covered in the first clause of the *max* quantity, while Cases 2 and 3 are covered

[3] Even though c_i can be stored for $S_2 \leq B < S_1$, there will be insufficient space ($< S_1$) to allocate to the lowest node of the path that c_i is attached to.

in the second clause (F$[i,B]$). Of course, if the remaining space is less than S_1 or $i \geq N$, no coefficient value can be stored, thus returning a benefit of 0 for M$[i,B]$.

Given Equation 1, our HCDynL2 algorithm starts at the root of the error tree and seeks to calculate the value of M$[0,B]$. In this process, various M[] and F[] values are calculated. For each of these calculations we also record which clause of the formulas helped determine these values, and the corresponding allotments b_L to the left subtree of the nodes (see Equation 1). This step helps to quickly trace the steps of the algorithm when reconstructing the optimal solution.

After the value M$[0,B]$ has been calculated, we can reconstruct the optimal solution as follows. We start from the root node with a space constraint B. Based on which clause determined the value of M$[i,B]$, we recurse to the two subtrees with the appropriate space allocation (recall that this information was recorded in the calculation of the M[] and F[] values) and a list of *hanging* coefficient values. These coefficient values belong to the hierarchically compressed path that passes through c_i. This path needs to be included in the recursion process because it can only be stored when all of its the coefficient values have been identified. Based on Cases 1-3 described above, at each node we may either: (i) Not store c_i; then store the input hanging path if it is non-empty; (ii) Attach c_i to the received hanging path (Case 2) and store the resulting hierarchically compressed coefficient; or (iii) Attach c_i to the received hanging path (Case 3) and recurse to the two subtrees. In this recursion, the resulting hanging path needs to be input to the appropriate subtree, while the other subtree will receive an empty hanging path.

Theorem 1 *The* HCDynL2 *algorithm computes the optimal* M$[i,B]$ *and* F$[i,B]$ *values at each node of the error tree and for any space constraint B correctly.*

Proof We will prove Theorem 1 by induction on the height of each coefficient from the bottom of the error tree (i.e., leaf nodes correspond to height 1).

Base Case: Leaf nodes (height = 1). If coefficient c_i belongs at the leaf level, then the possible set of paths in the subtree of c_i degenerates to simply storing c_i. Thus, the optimal benefit of a solution M$[i,B]$ is equal to $(c_i^*)^2$ for $B \geq S_1$ and 0, otherwise. Similarly, for $B \geq S_1$, F$[i,B]$ = M$[i,B]$ = $(c_i^*)^2$. Otherwise, c_i cannot be stored because of space constraints (thus the benefit is set $-\infty$ to represent this). Notice that in all cases the formulas for calculating M$[i,B]$ and F$[i,B]$ correctly compute the optimal solution and its benefit for any leaf node c_i and for any space constraint B assigned to the subtree of the node.

Inductive Step. Assume that the HCDynL2 algorithm correctly computes the optimal M$[i,B]$ and F$[i,B]$ values at each node of the error tree up to height j and for any space constraint B. We will show that the HCDynL2 algorithm also correctly computes the optimal M$[i,B]$ (the proof for F$[i,B]$ is similar) values at each node at height $j+1$.

Note that the HCDynL2 algorithm considers all combinations of storing (or not) the root coefficient at each subtree and attaching this coefficient (or not) to optimal solutions calculated by the node's two subtrees. Thus, a case of suboptimality may occur only if the optimal solution at node i needs to be computed by using a suboptimal solution (other than the computed M$[2i,B]$ and M$[2i+1,B]$ values, or the F$[2i,B]$ and F$[2i+1,B]$ values when c_i is stored) at (at least) one of its two subtrees.

Let the suboptimal solution needed to be considered is over a solution computed over the left subtree of c_i (i.e., the subtree of coefficient c_{2i}). Situations where the suboptimal solution is over the right subtree (or over both subtrees) are handled in a similar way.

First Case: M$[i,B]$ **does not store** c_i**.** Consider that the optimal solution at a coefficient c_i that lies at height $j+1$ of the error tree for a space constraint B is produced by not storing c_i, but by considering solutions LSOL and RSOL at the left and right subtrees, respectively, of c_i with corresponding maximum space b_L and b_R. Let the solution M$'[2i,b_L]$ at the left subtree be a suboptimal one, meaning that M$'[2i,b_L] <$ M$[2i,b_L]$. Then, a solution that would consider RSOL and the solution of M$[2i,b_L]$ requires at most space $b_L + b_R$ and has a larger benefit than the optimal solution of LSOL and RSOL. We therefore reached a contradiction.

Second Case: M$[i,B]$ **stores** c_i **and does not attach it in paths of the solutions of any subtree.** In this case, if the optimal solution needs to consider a sub-optimal solution LSOL at the left subtree of c_i with space b_L, then obviously HCDynL2 examines the solution that stores M$[2i,b_L]$ instead of LSOL, and which results in a larger benefit. We therefore reached a contradiction.

Third Case: M$[i,B]$ **stores** c_i **and attaches it to suboptimal solution** LSOL (RSOL) **at left (right) subtree.** In this case, note that c_{2i} must be stored in the suboptimal solution LSOL (RSOL) considered at the left (right) subtree (and thus c_i requires space S_2 to be stored). Note that the solution that stores c_i and attaches it to the solution of F$[2i,b_L]$ (F$[2i+1,b_R]$), where b_L (b_R) denotes the space of the suboptimal solution LSOL (RSOL), while also storing M$[2i+1,B-b_L-S_2]$ (M$[2i,B-b_R-S_2]$) will result in a larger benefit, due to the inductive hypothesis. We therefore reached a contradiction.

□

$$\mathrm{F}[i,B]=\begin{cases}-\infty & \text{, if } i\geq N \text{ or } B<S_1-1\\ \max\begin{Bmatrix}\max\limits_{0\leq b_L\leq B-S_2-1}(c_i^*)^2+\mathrm{G}[2i,b_L]+\mathrm{M}[2i+1,B-b_L-S_2-1]\\ \max\limits_{0\leq b_L\leq B-S_2-1}(c_i^*)^2+\mathrm{M}[2i,b_L]+\mathrm{G}[2i+1,B-b_L-S_2-1]\\ \max\limits_{0\leq b_L\leq B-S_2}(c_i^*)^2+\mathrm{F}[2i,b_L]+\mathrm{M}[2i+1,B-b_L-S_2]\\ \max\limits_{0\leq b_L\leq B-S_2}(c_i^*)^2+\mathrm{M}[2i,b_L]+\mathrm{F}[2i+1,B-b_L-S_2]\end{Bmatrix} & \text{, otherwise}\end{cases}$$

$$\mathrm{G}[i,B]=\begin{cases}-\infty & \text{, if } i\geq N \text{ or } B<S_1-1\\ \max\limits_{0\leq b_L\leq B-S_1+1}(c_i^*)^2+\mathrm{M}[2i,b_L]+\mathrm{M}[2i+1,B-b_L-S_1+1] & \text{, otherwise}\end{cases}$$

$$\mathrm{M}[i,B]=\begin{cases}0 & \text{, if } i\geq N \text{ or } B<S_1-1\\ \max\begin{Bmatrix}\max\limits_{0\leq b_L\leq B}\mathrm{M}[2i,b_L]+\mathrm{M}[2i+1,B-b_L]\\ \mathrm{F}[i,B]\\ \mathrm{G}[i,B]\end{Bmatrix} & \text{, otherwise}\end{cases} \tag{2}$$

3.2 Running Time and Space Complexities

Consider a node c_i at height j in the error tree. Since there can be at most 2^j-1 coefficients below the subtree rooted at node c_i, the total budget allocated cannot exceed $2^j\cdot S_1$. Therefore, at any node c_i, HCDynL2 must calculate at most $\min\{B,2^jS_1\}$ entries (if $2^jS_1<B$, all space allotments larger than 2^jS_1 result in the same benefit as that of the allotment for 2^jS_1 and need not be computed), where each requires time $\min\{B,\,2^jS_1\}$ to consider all possible space allotments to the children nodes. Given that there are $N/2^j$ nodes at height j and summing across all $\log N$ heights we obtain (note that $B=2^jS_1$ when $j=\log\frac{B}{S_1}$):

$$\sum_{j=1}^{\log N}\frac{N}{2^j}\left(\min\{B,2^jS_1\}\right)^2=\sum_{j=1}^{\log\frac{B}{S_1}}\frac{N}{2^j}2^{2j}S_1^2+\sum_{j=\log\frac{B}{S_1}+1}^{\log N}\frac{N}{2^j}B^2$$
$$=NS_1^2\sum_{j=1}^{\log\frac{B}{S_1}}2^j+NB^2\sum_{j=\log\frac{B}{S_1}+1}^{\log N}\frac{1}{2^j}$$
$$=NS_1^2\cdot O(\frac{B}{S_1})+NB^2\cdot O(\frac{S_1}{B})=O(S_1NB)=O(NB).$$

Note that the reconstruction process simply requires a top-down traversal of the error tree. Therefore, the total running time remains $O(NB)$. Using similar arguments, we obtain that the space complexity is:

$$\sum_{j=1}^{\log\frac{B}{S_1}}\frac{N}{2^j}2^jS_1+\sum_{j=\log\frac{B}{S_1}+1}^{\log N}\frac{N}{2^j}B=NS_1\log B+NB\cdot O(\frac{S_1}{B})$$
$$=O(N\log B).$$

Theorem 2 *The* HCDynL2 *algorithm constructs the optimal HCWS, given a space budget of B, in $O(NB)$ time using $O(N\log B)$ space.*

A Streaming Variant. The HCDynL2 algorithm can be easily modified to operate in one pass over the data using limited memory — i.e., in a data stream setting. Recall that HCDynL2 requires two passes over the data, one bottom-up for computing the optimal benefit while marking the decisions made, and another top-down for constructing the optimal HCWS. The streaming variation, denoted as HCDynL2-Str, makes two important observations: (i) not all the entries of the dynamic programming array are needed for the construction of the optimal HCWS; and (ii) in order to reconstruct the solution, the selected coefficients must be carried at each node along each M[] and F[] entry. Thus, in principle our HCDynL2-Str algorithm follows the same observations made in [14]. However, as explained later in this section, our main technical contribution in the HCDynL2-Str algorithm involves the ability to calculate the M[] and F[] entries and perform the memory cleanup in an efficient way, through some careful book-keeping. This step was not present in [14]. The following definition is helpful in our remaining discussion.

Definition 3 *A wavelet coefficient is termed as* closed *if we have observed all the data values in its support region. A wavelet coefficient is termed as* active *if it is not* closed *and for which we have already observed at least one data value in its support region. A wavelet coefficient is termed as* inactive *if it is neither active nor closed.*

For the first observation, notice that at any time a new data value d_i is read, then the *active* wavelet coefficients, which lie in $\mathrm{path}(d_i)$, need to be updated and their corresponding M[] and F[] entries need to be calculated again. Each such *active* node will also require in its operation the corresponding M[] and F[] entries of its child node that does not lie in $\mathrm{path}(d_i)$. Thus, for any space allotment b, only $O(\log N)$ (rather than $O(N)$) entries $\mathrm{M}[\cdot,b]$ and $\mathrm{F}[\cdot,b]$ need to be in memory, for any b — the remaining entries are only required for the second pass over the error tree.

For the algorithm to operate in one pass, the price that has to be paid is that of increased space requirements per M[] and F[] entry. Namely, following the second observation, we need a factor of $O(\min\{B,2^j\})$ more space to store the HCCs calculated so far at each node that belongs at height j of the error tree (again, this space is required only for the aforementioned $O(\log N)$ active nodes). An important observation is that through some careful book-keeping for each entry M$[\cdot,b]$, F$[\cdot,b]$ we only require $O(1)$ time to calculate the HCCs involved. To achieve this, we maintain the selected HCCs at each of the aforementioned $O(\log N)$ nodes as a list, where the first element is always the HCC that includes the root coefficient of the node's subtree (note that such a HCC may not exist for the M[] entry). The selected HCCs of a node are updated when data values in its support region are observed. However, as we explain later in this section, for all nodes that become *closed* we need to perform a cleanup operation that removes from main memory certain HCCs of these nodes. This cleanup operation is thus performed only once, and not per observed data value in their support region.

For each allotment b to the node's subtree the HCCs for the M[] and F[] entries can be computed as follows (assuming that b_L (b_R) space is allocated to the node's left (right) subtree):

1. If c_i is stored and attached to a path originating from the node's left (right) subtree, then create a new HCC that is the result of adding c_i to the first HCC that corresponds to the solution of the F$[2i,b_L]$ (F$[2i+1,b_R]$). We then link this new HCC to the second element of the list of F$[2i,b_L]$ (F$[2i+1,b_R]$) and with the corresponding list of the M$[2i+1,b_R]$ (M$[2i,b_L]$) entry.
2. If c_i is stored but is not attached to any path originating from the node's left (right) subtree, then create a new HCC containing only c_i. Then link to this HCC the lists of HCCs that correspond to M$[2i,b_L]$ and M$[2i+1,b_R]$.
3. If c_i is not stored then simply link the lists of HCCs that correspond to M$[2i,b_L]$ and M$[2i+1,b_R]$.

All the above operations can be completed in $O(1)$ time, along with the removal of the HCCs that were created at node c_i (and linked to the HCC lists calculated at the children nodes of c_i) at the observation of previous data values (see the above 3 cases). Please note that when we compute the final list of HCCs for any node c_i (after the node becomes *closed*), then any HCCs of its children nodes c_{2i} and c_{2i+1} that are not part of the HCCs stored at node c_i are no longer needed and need to be deleted. This can be easily detected by examining how the M$[i,b]$ and F$[i,b]$ entries at node c_i were calculated. Assuming that c_i belongs at height j of the error tree, this can be achieved in $O(\min\{B,2^j\})$ time per each space allotment $b \le B$ to c_i.

Thus, the overall running time requirements of the algorithm become (since the HCCs of each node are calculated continuously, but the memory cleanup is performed just once per node and per space allotment $b \le B$):

$$\begin{aligned}
&\sum_{j=1}^{\log N} \frac{N}{2^j}\left(\left(\min\{B,2^j\}\right)^2 + \log N \cdot \min\{B,2^j\}\right) \\
&= \sum_{j=1}^{\log B} \frac{N}{2^j}(2^{2j} + 2^j \log N) + \sum_{j=\log B+1}^{\log N} \frac{N}{2^j}(B^2 + B\log N) \\
&= N\sum_{j=1}^{\log B}(2^j + \log N) + NB(B+\log N)\sum_{j=\log B+1}^{\log N}\frac{1}{2^j} \\
&= O(NB + N\log N\log B) + N(B+\log N) \\
&= O(NB + N\log N\log B).
\end{aligned}$$

To summarize, HCDynL2-Str operates in one pass over the data and gains in space by storing only $B\log N$ entries, which, however, each requires $O(B)$ space for the storage of its HCCs. Moreover, at any specific moment the currently selected HCWS can be accessed directly from the root node of the error tree.

Theorem 3 *The* HCDynL2-Str *algorithm constructs the optimal HCWS in one pass, given a space budget of B, in $O(B + \log N \log B)$ amortized time per processed data value using $O(B^2 \log N)$ space.*

3.3 Achieved Benefit vs. Classic Method

A question that naturally arises is how does the benefit of the solution achieved by the HCDynL2 algorithm compare to the one achieved by a traditional technique (Classic) that individually stores the coefficients with the largest absolute normalized values. Consider a set $\mathcal{S} = S_1,\ldots,S_B$ of B stored coefficient values, sorted in non-increasing order of their absolute normalized values, by the traditional thresholding algorithm. Consider the case where these coefficient values lie in distant regions of the error tree. Using a space constraint equal to $B \times (sizeof(Coord) + sizeof(Value)) = B \times (S_1 - 1)$ the benefit of the HCDynL2 algorithm cannot be smaller than the benefit of storing the first $m = \lfloor \frac{B\times(S_1-1)}{S_1} \rfloor$ coefficient values of $\mathcal{S}$ as hierarchically compressed wavelet coefficients, each storing exactly one coefficient value. Thus, in the worst case the ratio of benefits of the HCDynL2 algorithm over the Classic algorithm, as described above, may be as low as $\frac{Benefit(\text{HCDynL2})}{Benefit(\text{Classic})} = \frac{Benefit(S_1,\ldots,S_m)}{Benefit(S_1,\ldots,S_B)} \ge \frac{m}{B}$, since the coefficients in $\mathcal{S}$ are sorted.

On the other hand, the best case for the benefit of the HCDynL2 algorithm may occur for a storage constraint of $B' = S_1 + (\log N + 1) \times S_2$. In this case if the $\log N + 1$ coefficient values with the largest absolute normalized values lie on the same root-to-leaf path of the error tree, then the ratio of benefits of the HCDynL2 algorithm over the Classic algorithm will be as high as (for $m' = \lfloor \frac{B'}{S_1 - 1} \rfloor$) : $\frac{Benefit(\text{HCDynL2})}{Benefit(\text{Classic})} = \frac{Benefit(S_1,\ldots,S_{\log N+1})}{Benefit(S_1,\ldots,S_{m'})} \le \frac{\log N+1}{m'}$. Therefore, the following theorem holds.

Theorem 4 *The* HCDynL2 *algorithm, when compared to the* Classic *algorithm, given the same space budget, exhibits a benefit ratio of*

$$\frac{\left\lfloor B \times \frac{S_1-1}{S_1} \right\rfloor}{B} \leq \frac{Benefit(\mathsf{HCDynL2})}{Benefit(\mathsf{Classic})} \leq \frac{\log N + 1}{\left\lfloor \frac{S_1+(\log N+1)S_2}{S_1-1} \right\rfloor}.$$

An Improved-Benefit Variant. It is important to emphasize that the HCDynL2 algorithm can be easily modified to guarantee that its produced solution has a benefit at least equal to the one of the traditional approach. This can be achieved by allowing HCCs with a single stored coefficient value to drop the very small overhead of the single bit and be stored in a separate storage. In this case, the first stored coefficient in a HCC requires space $S_1 - 1$, the second coefficient value in the same HCC requires additional space equal to $S_2 + 1$, while any additional coefficient values in the same HCC require space S_2 to be stored. This results in constructing a modified HCWS* synopsis.

The main difference of the modified algorithm, denoted as HCDynL2*, compared to the discussion of Section 3 is that now, due to the different space needed for the second and third coefficient values of each HCC, two suboptimal solutions need to be maintained (see Equation 2): (i) $\mathrm{F}[i,B]$, the benefit of the optimal solution when assigning space at most equal to B to the subtree of coefficient c_i and when both c_i and one of its children (c_{2i} or c_{2i+1}) are stored; and (ii) $\mathrm{G}[i,B]$, the benefit of the optimal solution when assigning space at most equal to B to the subtree of coefficient c_i and c_i is stored as the bottom-most coefficient in a path. Note, that for the second suboptimal solution the space required is $S_1 - 1$, as discussed. For the first suboptimal solution two cases exist: (a) c_i is the second coefficient in a path, hence, the space required is $S_2 + 1$ and further, a suboptimal solution G[] in one of its children must be combined with an optimal solution M[] in the other child (the first two non-trivial cases of $\mathrm{F}[i,B]$ in Equation 2); and (b) c_i is not the second coefficient (it could be the third or more), hence, the space required is S_2 and further, a suboptimal solution F[] in one of its children must be combined with an optimal solution M[] in the other child (the last two non-trivial cases of $\mathrm{F}[i,B]$ in Equation 2. Therefore, the following theorem holds.

Theorem 5 *The* HCDynL2* *algorithm constructs a modified HCWS* synopsis such that the obtained benefit is never less than that of the* Classic *algorithm, given the same space budget:*

$$Benefit(\mathsf{HCDynL2}^*) \geq Benefit(\mathsf{Classic}).$$

It is important to note that the asymptotic running time and space requirements of the HCDynL2* algorithm are the same as those of the HCDynL2 algorithm. However, since its implementation requires the evaluation of three DP recurrences, its actual running time and space requirements are about 50% increased over the ones of HCDynL2. Finally, a streaming variant of the HCDynL2* algorithm can be obtained in a manner analogous to that of HCDynL2. Similarly, an approximation algorithm for the HCDynL2* algorithm can be obtained in a manner analogous to the approximation algorithm of HCDynL2 (presented in Section 4).

Symbol	Description
$\mathrm{APPRM}[i,x]$	Approximate value for optimal benefit when assigning at most space x to the subtree rooted at coefficient c_i
$\mathrm{APPRF}[i,x]$	Approximate value for optimal benefit when assigning at most space x to the subtree rooted at coefficient c_i and when c_i is forced to be stored
$\{p^i_j\}$	Set of breakpoints for $\mathrm{APPRM}[i,\cdot]$
$\{q^i_k\}$	Set of breakpoints for $\mathrm{APPRF}[i,\cdot]$
ε	Approximation factor
δ	Degradation factor incurred at each level

Table 4 Notation used in HCApprL2 Algorithm.

4 HCApprL2: An Approximation Algorithm

In this section we propose an approximation algorithm for efficiently constructing hierarchically compressed wavelet synopses. Our algorithm, termed HCApprL2, offers significant improvements in time and space requirements over HCDynL2 while providing tunable error guarantees. The HCApprL2 algorithm constructs a HCWS that has a benefit that does not exceed the optimal synopsis, but definitely not less than $\frac{1}{1+\varepsilon}$ of the optimal benefit, for some given parameter ε. Clearly, smaller values for ε lead to more accurate synopses; HCApprL2 solves Problem 2 optimally for $\varepsilon = 0$.

The HCApprL2 algorithm constructs functions APPRM[], APPRF[] and computes their values at some space allotment in a similar manner to how HCDynL2 computes M[] and F[] values (i.e., the values at a non-leaf node depend on the values of its children) but does so for a sparse set of space allotments, termed breakpoints, rather than for all possible allotments.

The HCApprL2 algorithm operates on the error tree in a bottom-up manner. At each node it creates a set of *candidate* breakpoints by combining breakpoints from the children of the node. Then, in a two-phase trimming process it eliminates some of these candidates to obtain the actual breakpoints of the node. This trimming process is responsible for bounding the error incurred by not examining all space allotments, as it will become apparent in the next section.

4.1 Breakpoint Calculation

The crux of the HCApprL2 algorithm lies in the calculation of the breakpoints and their corresponding benefit values for functions APPRM[] and APPRF[] at each node. The algorithm proceeds in a bottom-up manner, starting from the leaf nodes at height 1.

$$\left.\begin{aligned}\text{APPRF}[i,p_j^L+p_k^R+S_1] &= (c_i^*)^2+\text{APPRM}[2i,p_j^L]+\text{APPRM}[2i+1,p_k^R]\\ \text{APPRF}[i,q_j^L+p_k^R+S_2] &= (c_i^*)^2+\text{APPRF}[2i,q_j^L]+\text{APPRM}[2i+1,p_k^R]\\ \text{APPRF}[i,p_j^L+q_k^R+S_2] &= (c_i^*)^2+\text{APPRM}[2i,p_j^L]+\text{APPRF}[2i+1,q_k^R]\end{aligned}\right\} \quad (3)$$

$$\left.\begin{aligned}\text{APPRM}[i,p_j^L+p_k^R+S_1] &= \text{APPRF}[i,p_j^L+p_k^R+S_1]\\ \text{APPRM}[i,q_j^L+p_k^R+S_2] &= \text{APPRF}[i,q_j^L+p_k^R+S_2]\\ \text{APPRM}[i,p_j^L+q_k^R+S_2] &= \text{APPRF}[i,p_j^L+q_k^R+S_2]\\ \text{APPRM}[i,p_j^L+p_k^R] &= \text{APPRM}[2i,p_j^L]+\text{APPRM}[2i+1,p_k^R]\end{aligned}\right\} \quad (4)$$

Assume that node c_i, at height 1, is a non-zero leaf coefficient. In this case there are two breakpoints 0 and S_1 with approximate benefits 0 and $(c_i^*)^2$ respectively for both approximation functions. In the case of a zero valued leaf coefficient APPRM$[i,\cdot]$ has only breakpoint 0 with zero benefit, whereas APPRF$[i,\cdot]$ has breakpoints $0, S_1$ with benefits $-\infty$ and 0 respectively.

For all non-leaf nodes the breakpoint calculation proceeds following the same steps: (i) a set of candidate breakpoints is created by combining all breakpoints of the children; (ii) a trimming process reduces this set to the actual breakpoints to be used as input for the first step in the parent node.

Consider a non-leaf node c_i at height l; since HCApprL2 proceeds bottom up all breakpoints for nodes lower in the tree have been calculated. Thus, let $\{p_j^L\}$, $\{q_k^L\}$ denote the set of breakpoints for APPRM$[2i,\cdot]$ and APPRF$[2i,\cdot]$ functions for the left child of c_i and let $\{p_j^R\}$, $\{q_k^R\}$ denote the set of breakpoints for APPRM$[2i+1,\cdot]$ and APPRF$[2i+1,\cdot]$ functions for the right child of c_i.

The candidate breakpoints for APPRF$[i,\cdot]$ and the corresponding benefit values are calculated combining all breakpoints from sets $\{p_j^L\}$, $\{q_k^L\}$, $\{p_j^R\}$, $\{q_k^R\}$ as shown in Equation 3 — candidate breakpoints of space more than B are easily identified and rejected. Observe that these equations correspond to the non-trivial cases of the defining recurrence for F$[i,\cdot]$ (Equation 1). The algorithm considers the following cases for all j,k:

- Store c_i using space S_1 and combine all (approximately) optimal solutions APPRM$[2i,p_j^L]$, APPRM$[2i+1,p_j^R]$.
- Store c_i using space S_2 and combine all (approximately) optimal when forced to store c_{2i} solutions APPRF$[2i,q_k^L]$ with all (approximately) optimal solutions APPRM$[2i+1,p_j^R]$.
- Store c_i using space S_2 and combine all (approximately) optimal solutions APPRM$[2i,p_j^L]$ with all (approximately) optimal when forced to store c_{2i+1} solutions APPRF$[2i+1,q_k^R]$.

Similarly, the candidate breakpoints for APPRM$[i,\cdot]$ and their corresponding benefit values are also calculated combining all breakpoints from sets $\{p_j^L\}$, $\{q_k^L\}$, $\{p_j^R\}$, $\{q_k^R\}$ as shown in Equation 4 — candidate breakpoints of space more than B are easily identified and rejected. Again, observe that these equations correspond to the non-trivial cases of the defining recurrence for M$[i,\cdot]$ (see Equation 1), which in addition to the candidate breakpoints considered for APPRF$[i,\cdot]$ considers the following case, for all j,k: Do not store c_i and combine all (approximately) optimal solutions, i.e., the pairs APPRM$[2i,p_j^L]$, APPRM$[2i+1,p_j^R]$.

Once all candidate breakpoints have been calculated we perform a two-phase trimming process for each approximation function, to reduce the number of breakpoints.

First Phase. We remove the useless configurations — those that cost more but have less benefit than others. This can be done by a simple ordering of the configurations increasingly by their approximate benefit values and a subsequent linear scan.

Second Phase. The final breakpoints $\{p_j^i\}$, $\{q_k^i\}$ are set as follows. Consider the case of the approximate benefit function APPRM$[i,\cdot]$. Set p_1^i equal to the first candidate breakpoint (after sorting); it is easy to see that this breakpoint always corresponds to space 0. The rest of the breakpoints are discovered iteratively: assuming breakpoint p_{k-1}^i has been found, breakpoint p_k^i is the smallest candidate breakpoint such that APPRM$[i,p_k^i] > (1+\delta)$APPRM$[i,p_{k-1}^i]$, for some parameter δ which depends on the desired approximation factor ε and whose value will be determined later in this section.

The following lemmas are a direct result of the trimming process.

Lemma 2 *For any node that belongs at height j of the error tree, there can be at most $R_j = O\left(\min\{B, 2^j, \frac{1}{\delta}\log||W_A||\}\right)$ breakpoints.*

Proof Certainly, there can be no more than B breakpoints for each approximation function. Similarly, since there can be at most 2^j-1 coefficients in the subtree rooted at each node that belongs at height j, the total number of space entries, and thus breakpoints, cannot exceed $2^jS_1 = O(2^j)$. Additionally, there can be no more than $\log_{1+\delta}$ M$[i,B]$ breakpoints for APPRM$[i,\cdot]$ (and no more than $\log_{1+\delta}$ F$[i,B]$ for APPRF$[i,\cdot]$), as M$[i,B]$ (resp. F$[i,B]$) is the highest possible benefit that can be attained at node c_i for space B. Since this benefit cannot be more than the energy of the wavelet transform $||W_A||^2$, the lemma easily follows for small δ values. □

Lemma 3 *Let $\{p_j^i\}$ be the set of breakpoints for approximation benefit function* APPRM$[i,\cdot]$. *If b is a candidate breakpoint such that $b \in [p_k^i, p_{k+1}^i)$, then* APPRM$[i,b] \leq (1+\delta)\cdot$ APPRM$[i,p_k^i]$ *— i.e., b is covered by p_k^i within a $(1+\delta)$ degradation. Analogous result holds for function* APPRF$[]$.

Proof If b is not discarded in the first phase of the trimming process it is straightforward to see that the lemma holds. Now, assume that b was discarded in the first phase. Therefore, there must exist a candidate breakpoint $b' < b$ not discarded in the first phase with $\text{APPRM}[i,b'] \geq \text{APPRM}[i,b]$ such that b' is the highest non-discarded breakpoint smaller than b. Observe that b' and b are covered by the same breakpoint p^i_k (b' might be the breakpoint p^i_k): $b,b' \in [p^i_k, p^i_{k+1})$ and that the lemma holds for b'. Therefore, $\text{APPRM}[i,b] \leq \text{APPRM}[i,b'] \leq (1+\delta)\text{APPRM}[i,p^i_k]$ and the lemma holds for b. □

By aggregating the degradation occurred at all descendants of a node we obtain the following.

Lemma 4 *Assume node c_i is at height h of the error tree (equivalently at level $\log N - h$), and let $\{p^i_j\}$ and $\{q^i_j\}$ be the set of breakpoints for $\text{APPRM}[i,\cdot]$ and $\text{APPRF}[i,\cdot]$ respectively. Also let x,y be some arbitrary space allotments and let breakpoints p^i_k, q^i_k be such that $x \in [p^i_k, p^i_{k+1})$ (or $x \geq p^i_k$, if p^i_k is the last breakpoint) and $y \in [q^i_k, q^i_{k+1})$ (or $y \geq q^i_k$, if q^i_k is the last breakpoint). The approximate benefit value computed at node c_i (the approximate benefit value when c_i is forced to be stored) for space p^i_k (resp. q^i_k) is not less than $\frac{1}{(1+\delta)^{h-1}}$ of the optimal value (resp. optimal value when c_i is forced to be stored). That is, $\text{M}[i,x] \leq (1+\delta)^{h-1}\text{APPRM}[i,p^i_k]$ and $\text{F}[i,y] \leq (1+\delta)^{h-1}\text{APPRF}[i,q^i_k]$.*

Proof We prove the lemma for $\text{APPRF}[i,\cdot]$ and $\text{APPRM}[i,\cdot]$, by induction on the height h of the error tree node c_i belongs to. The base case $h{=}1$ holds by construction: Assume coefficient c_i is non-zero; thus only breakpoints $p^i_1{=}0$, $p^i_2{=}S_1$ and $q^i_1{=}0$, $q^i_2{=}S_1$ exist for approximation functions $\text{APPRM}[i,\cdot]$ and $\text{APPRF}[i,\cdot]$ respectively. Clearly, (i) when $x \in [p^i_1, p^i_2)$, $\text{APPRM}[i,p^i_1]{=}\text{M}[i,x]$; (ii) when $y \in [q^i_1, q^i_2)$, $\text{APPRF}[i,q^i_1]{=}\text{F}[i,y]$; (iii) when $x \geq p^i_2$, $\text{APPRM}[i,p^i_2]{=}\text{M}[i,x]$; and (iv) when $y \geq q^i_2$, $\text{APPRF}[i,q^i_2]{=}\text{F}[i,y]$. In the case of a zero valued coefficient c_i, only breakpoint p^i_1 exists and the reasoning is similar.

Assuming the hypothesis holds for all nodes at height h we will show that it holds for nodes at height $h+1$. We will only consider the approximation function $\text{APPRF}[i,\cdot]$ for node c_i at height $h+1$, as the proof for $\text{APPRM}[i,\cdot]$ is similar. Further, assume that the optimal benefit $\text{F}[i,y]$ when c_i is forced to be stored for a space budget of y is constructed from the second non-trivial clause of Equation 1 by allotting space S_2 to coefficient c_i, y_L to the left subtree and $y{-}y_L{-}S_2$ to the right subtree; that is, $\text{F}[i,y] = (c^*_i)^2 + \text{F}[2i,y_L] + \text{M}[2i{+}1, y{-}y_L{-}S_2]$. The proof is similar for the other clauses and thus omitted.

If $\{q^L_j\}$ and $\{p^R_j\}$ denote the sets of breakpoints for functions $\text{APPRF}[2i,\cdot]$ and $\text{APPRM}[2i+1,\cdot]$ respectively, let q^L_k be the highest breakpoint not exceeding y_L and let p^R_k be the highest breakpoint less than $y{-}y_L{-}S_2$. By the induction hypothesis

$$\text{F}[2i,y_L] \leq (1+\delta)^{h-1}\text{APPRF}[2i,q^L_k] \text{ and}$$
$$\text{M}[2i+1, y-y_L-S_1] \leq (1+\delta)^{h-1}\text{APPRM}[2i+1,p^R_k].$$

Define $b = q^L_k + p^R_k + S_1$. Certainly, b was a candidate breakpoint for function $\text{APPRF}[i,\cdot]$ and considered by our HCApprL2 algorithm (see Equation 3):

$$\text{APPRF}[i,b] = (c^*_i)^2 + \text{APPRF}[2i,q^L_k] + \text{APPRM}[2i+1,p^R_k].$$

Using the above equation and the induction hypothesis, optimal value $\text{F}[i,y]$ is bounded as follows.

$$\begin{aligned}\text{F}[i,y] &= (c^*_i)^2 + \text{F}[2i,y_L] + \text{M}[2i+1,y-y_L-S_2]\\ &\leq (c^*_i)^2 + (1{+}\delta)^{h-1}\left(\text{APPRF}[2i,q^L_k] + \text{APPRM}[2i{+}1,p^R_k]\right)\\ &\leq (1+\delta)^{h-1}\text{APPRF}[i,b]\end{aligned}$$

Now, either b belongs to $[q^i_k, q^i_{k+1})$ or not. Consider the first case. By Lemma 3 $\text{APPRF}[i,b] \leq (1+\delta)\text{APPRF}[i,q^i_k]$ and thus:

$$\text{F}[i,y] \leq (1+\delta)^{h-1}\text{APPRF}[i,b] \leq (1+\delta)^{h}\text{APPRF}[i,q^i_k].$$

In the other case, observe that b must be smaller than q^i_k, because $b \leq y \in [q^i_k, q^i_{k+1})$. Therefore, since $\text{APPRF}[i,b] \leq \text{APPRF}[i,q^i_k]$:

$$\text{F}[i,y] \leq (1+\delta)^{h-1}\text{APPRF}[i,b] \leq (1+\delta)^{h-1}\text{APPRF}[i,q^i_k].$$

Thus, in either case $\text{F}[i,y] \leq (1+\delta)^{h}\text{APPRF}[i,q^i_k]$. □

Finally, we obtain the following.

Theorem 6 *The HCApprL2 algorithm provides a HC synopsis to Problem 2 that needs space not more than B and has benefit not less than $\frac{1}{1+\varepsilon}$ of the optimal benefit. Assuming p^0_k is the highest breakpoint of function $\text{APPRM}[0,\cdot]$ not exceeding B, we have $\text{M}[0,B] \leq (1+\varepsilon)\text{APPRM}[0,p^0_k]$.*

Proof Apply Lemma 4 for $\text{M}[0,B]$ setting $\delta = \frac{\varepsilon}{\log N}$. □

4.2 Space and Running Time Complexities

The space and time complexity of the HCApprL2 algorithm depend on the number of breakpoints R_j (rather than solely on B) for each approximation function at each node that belongs at height j of the error tree. Lemma 2 provides a bound for this number, if one sets $\delta = \frac{\varepsilon}{\log N}$: $R_{max} = O\left(\min\{B, 2^j, \frac{1}{\varepsilon}\log N \log \|W_A\|\}\right)$.

At each node and for each approximation function, the HCApprL2 algorithm first computes candidate breakpoints by combining all breakpoints from the children nodes (in $O(R_j^2)$ time and space), sorts them (in $O(R_j^2 \log R_j)$ time) and performs the trimming process (in $O(R_j^2)$ time and space). Thus, the time requirement is $O(R_j^2 \log R_j)$ per node at height j of the error tree. HCApprL2 requires a temporary space of $O(R_j^2)$ to perform the trimming process, but registers only $O(R_j)$ space per node. Using similar reasoning with the complexity analysis of the HCDynL2 algorithm we derive the following running time complexity our algorithm (by setting $K = \min\{B, \frac{1}{\varepsilon}\log N \log ||W_A||\}$ - observe the time requirement increases by a factor of $\log R_j$ due to the sorting involved during the breakpoint calculation):

$$O(\sum_{j=1}^{\log N} \frac{N}{2^j} \left((\min\{2^j, K\})^2 \log \min\{2^j, K\} \right)) =$$
$$O(\sum_{j=1}^{\log K} \frac{N}{2^j} j 2^{2j} + \sum_{j=\log K+1}^{\log N} \frac{N}{2^j} K^2 \log K)$$
$$= O(N \sum_{j=1}^{\log K} j 2^j + N K^2 \log K \sum_{j=\log K+1}^{\log N} \frac{1}{2^j})$$
$$= O(NK \log K).$$

Using a similar calculation for the space requirements of the algorithm, the following Theorem easily follows.

Theorem 7 *Given space budget B, the* HCApprL2 *algorithm constructs a HCWS, in $O(NK \log K)$ time using $O(N \log K)$ space, where $K = \min\{B, \frac{1}{\varepsilon}\log N \log ||W_A||\}$. The streaming variant of this algorithm requires only $O(K^2 \log N)$ space.*

Note that the streaming variant of the algorithm is analogous to the corresponding variant of the optimal DP algorithm, and is thus omitted from our presentation.

5 HCGreedyL2: A Greedy Heuristic

Due to the large space and running time requirements of the HCDynL2 and HCApprL2 algorithms, we now seek to devise a more efficient greedy solution for the same optimization problem. At first sight our optimization problem looks similar to the classical knapsack problem. However, our optimization problem is much more difficult for two reasons. First, even though the benefit of including any given coefficient in the synopsis is fixed, its space requirement depends on the position of the coefficient it the hierarchical path; it may require either S_1 or S_2 space. Second, considering the search space of all possible HCCs, observe that once a HCC is chosen, there is a large number of HCCs which become invalid and cannot be part of the solution; these are the hierarchically compressed coefficients that overlap with the chosen HCC. This dependency amongst the candidate HCCs is not typical in knapsack-like problems for which there exist greedy algorithms with tight approximation bounds.

In analogy to most greedy heuristics for knapsack-like problems, we try to formulate candidate solutions and utilize a per-space benefit heuristic at each step of the algorithm. In particular, our proposed HCGreedyL2 algorithm greedily allocates its available space by continuously selecting (until the space budget is exhausted) for storage the candidate path that (i) does not overlap any of the already selected for storage paths; and (ii) is estimated to exhibit the largest *per space benefit*, if included in the solution. To increase the effectiveness of the algorithm, it is crucial that, whenever possible, candidate paths be combined with paths already selected for storage, and that such storage dependencies be exploited. As we will explain shortly, this can be achieved by some careful book-keeping.

Symbol	Description
GrM_i	Non-stored candidate path in c_i's subtree with the estimated maximum per space benefit
GrF_i	Non-stored candidate path in c_i's subtree with the estimated maximum per space benefit when storing c_i
$Owner_i$	The hierarchically compressed coefficient in which c_i belongs to (Ø if c_i has not been stored)
$GrM_i.b$	Benefit of GrM_i
$GrM_i.sp$	Needed space for GrM_i
$GrF_i.b$	Benefit of GrF_i
$GrF_i.sp$	Needed space for GrF_i
$State_i[0..2]$	Bitmap of node i, consisting of 3 bits: - If $State(0)$ is set, c_i has already been selected for storage - If $State(0)$ and $State(1)$ are set, c_i = bottom($Owner_i$) - Otherwise, if $State(0)$ is set, $State(2)$ denotes if set (not set) that c_i is part of a path through its left (right) subtree
chM_i, chF_i	2-bit bitmaps for retracing the algorithm choices (determine through which action the paths GrM_i and GrF_i were formed)

Table 5 Notation used in HCGreedyL2 Algorithm.

The operation of the algorithm is based on two main steps, that are repeated several times, and that we will detail shortly: (i) Selecting good candidate paths per subtree; and (ii) Marking candidate paths for storage and properly adjusting the benefits of non-stored candidate paths. The first of these phases first occurs at the initialization phase of the algorithm by visiting all the nodes of the error tree, in order to setup the values of several variables at each node. Table 5 provides a synopsis of these variables, and of the notation used in the entire HCGreedyL2 algorithm. Appropriate definitions will be provided in our discussion whenever necessary. After this initialization phase, the coefficients in the path that is estimated at the root node to exhibit the best per space benefit are visited and marked for inclusion in the final solution (by modifying the *State* bitmap of these nodes). This is achieved by the second phase. Following each such marking process, the first phase needs to be called for each visited node of the error tree. Observe that calls to the second phase and all subsequent calls to the first phase only visit nodes in the currently selected path.

Before proceeding to our discussion, it is important to emphasize that the paths GrM_i, GrF_i and $Owner_i$ (referenced in Table 5) are not stored at each node, but can rather be reconstructed by an appropriate traversal of the error tree.

5.1 Candidate Path Selection

The computation of the best candidate path in a subtree of the error-tree structure is a bottom-up procedure. At each step of the algorithm, at each node c_i of the error tree we store the benefit and the corresponding space of two candidate paths: (i) the candidate path GrM_i in the subtree of c_i that is estimated to achieve, if stored, the best per space benefit; and (ii) the candidate path GrF_i of c_i's subtree that is estimated to achieve the best per space benefit while storing the coefficient c_i. This implies that GrM_i might be any path of the subtree rooted at c_i, whereas GrF_i has to be a path containing c_i.

In order to compute these two candidate paths along with their corresponding benefits and their needed space, the HC-GreedyL2 algorithm considers combining the coefficient value c_i with the candidate paths computed at c_i's two subtrees. This process utilizes some information that is produced during the operation of the algorithm and is stored as a bitmap $State$ in each node, whereas the choices made are stored in chF, chM (see Table 5).

In the following, we omit discussion on what happens in the case of the root node for exposition purposes; the required changes due to the root having a single child are straightforward.

5.1.1 Computing GrF_i

The computation of GrF_i depends on whether c_i has been stored (i.e., whether $State_i(0)$ is set).

Coefficient c_i has been stored. In this case there is no candidate path that can store c_i. Thus, in this case we have $GrF_i = \emptyset$ and we set $GrF_i.b = GrF_i.sp = 0$ and $chF_i = 00$.

Coefficient c_i has not been stored. The following choices should be considered and the one with the highest per space benefit is selected (by appropriately setting the value of chF_i):

1. Storing simply c_i ($chF_i = 01$). The space requirements of this solution depends on whether c_i can be attached to an already selected path. If c_i is a non-leaf node in the error tree and either $State_{2i}(0)$ or $State_{2i+1}(0)$ is set, then c_i can be attached to such a path (through the corresponding subtree) and $GrF_i.sp = S_2$. Otherwise, we set $GrF_i.sp = S_1$. This solution has a benefit equal to $(c_i^*)^2$ if the available space (at the step of the algorithm when this computation is performed) is at least $GrF_i.sp$, or 0 otherwise.
2. Storing c_i and combining it with GrF_{2i} ($chF_i = 10$) (or combining it with GrF_{2i+1} (chF_i=11)). This solution has an overall space requirement of S_2+$GrF_{2i}.sp$ (resp., $S_2+ GrF_{2i+1}.sp$) and a benefit of $(c_i^*)^2$+$GrF_i.b$ (resp., $(c_i^*)^2 + GrF_{2i+1}.b$) if the available space (at the step of the algorithm when this computation is performed) is at least $GrF_i.sp$, or $-\infty$ otherwise.

Node	*$GrM_i.b$*	*$GrM_i.sp$*	*$GrF_i.b$*	*$GrF_i.sp$*	*$State_i$*	*chM_i*	*chF_i*
8	288	65	288	65	000	11	01
9	2	65	2	65	000	11	01
10	0	65	0	65	000	11	01
11	242	65	242	65	000	11	01
12	0	65	0	65	000	11	01
13	0	65	0	65	000	11	01
14	0	65	0	65	000	11	01
15	648	65	648	65	000	11	01
4	288	65	292	98	000	01	10
5	242	65	342	98	000	10	11
6	0	65	0	65	000	11	01
7	648	65	972	98	000	10	11
2	584	131	584	131	000	11	11
3	648	65	1134	131	000	10	11
1	3844	65	3844	65	000	11	01
0	10244	98	10244	98	000	11	11

Table 6 Computed Values at Initialization Phase.

Moreover, in all three cases presented above, we decrease the $GrF.sp$ values by $S_1 - S_2$ if the parent node of c_i has been marked for storage and is also the bottom-most coefficient in its HCC. This is because GrF_i can help reduce, if selected for storage, the storage overhead for the parent node of c_i.

5.1.2 Computing GrM_i

The computation of GrM_i also depends on whether c_i has been stored (i.e., whether $State_i(0)$ is set).

Coefficient c_i has not been stored. The following choices should be considered and the one with the highest per space benefit is selected (by appropriately setting the chM_i value):

1. The candidate path of solution GrF_i (chM_i = 11).
2. For non-leaf nodes, GrM_i copies a candidate path from one of its children, either GrM_{2i} (chM_i=01) or GrM_{2i+1} (chM_i=10), selecting the one with the highest per space benefit.

Coefficient c_i has been stored. If c_i is a leaf node, then $GrM_i = \emptyset$ and we set $GrM_i.b = GrM_i.sp = 0$ and $chM_i = 00$. For non-leaf nodes, GrM_i examines the candidate paths GrM_{2i} and GrM_{2i+1} from its children nodes and copies the one that exhibits the largest per space benefit.

Example 8 In Table 6 we depict the calculated GrM_i, GrF_i, $State_i$, chM_i and chF_i values and bitmaps computed at each node of Figure 1 during the initialization phase of the HC-GreedyL2 algorithm. Based on the normalized coefficient values presented in Section 2, the benefit of storing each of the 16 coefficients is: [6400, 3844, 242, 162, 4, 100, 0, 324, 288, 2, 0, 242, 0, 0, 0, 648]. In this example, the size required to store a coefficient coordinate or a coefficient value has been set to 32 bits. The nodes in Table 6 have been ordered according to their resolution level. Details on the selected HCCs are provided later in this section. However, it is interesting to note that, even though the final selection of the HCCs has been presented in Section 2, the stored HCCs are produced by successive steps where smaller HCCs are merged. For example, by examining the GrF_1 values we observe that the HCC that is estimated to achieve the best per space benefit at node c_1 while also storing c_1 contains only the node c_1, and not the entire path c_{15}, c_7, c_3, c_1. This path will gradually be formed by the algorithm.

5.2 Marking Paths for Storage

After the path with the overall per space benefit has been estimated (GrM_0), and its space $GrM_0.sp$ is subtracted from the available space, the process of traversing the error tree to mark the coefficients in GrM_0 for storage is simple, due to the storage of the *chM* and *chF* bitmaps at each node. This top-down recursive process starts at the root node and descends the path that leads to the node bottom(GrM_0). The steps of this process are:

1. At each node c_i of this path, we are asked to reconstruct either the path GrM_i or the path GrF_i. Notice that reconstructing GrM_i may lead to reconstructing GrF_i if chM_i = 11.
2. This process will never visit a node where the corresponding chM_i or chF_i values are equal to ’00’.
3. If reconstructing GrF_i and chF_i = 01, then c_i is marked as stored by setting the bit $State_i(0)$ to 1. If in this case $GrF_i.sp$ = S_1, then $State_i(1)$ is set and we recomputed the *GrF* and *GrM* values at the two children nodes of c_i, as described in Section 5.1. Otherwise, we reset $State(1)$ and assign the value of $State_i(2)$ depending on which path c_i can be attached to (if it can be attached to paths from both subtrees, pick any one of them randomly).
4. If reconstructing GrF_i and chF_i = 10 (11), we mark c_i for storage by setting $State_i(0)$ to 1, resetting the value of $State_i(1)$ and setting the value of $State_i(2)$ to 1 (0, respectively). We also recurse to reconstruct GrF_{2i} (GrF_{2i+1}).
5. If reconstructing GrM_i and chM_i = 01 (10), then we recurse to reconstruct GrM_{2i} (GrM_{2i+1}). After this recursion, we need to check if the newly stored path in the subtree of c_{2i} (c_{2i+1}) can be attached to c_i. By following the process described in Section 5.1, if this is detected the value of $State_i(1)$ is reset and the value of $State_i(2)$ is set to 1 (0, correspondingly). Also, in this case, the *GrF* and *GrM* values of c_{2i+1} (c_{2i}) need to be recalculated, since any path containing c_{2i+1} (c_{2i}) cannot lower, any more, the storage cost of c_i.

Node	$GrM_i.b$	$GrM_i.sp$	$GrF_i.b$	$GrF_i.sp$	$State_i$	chM_i	chF_i
8	288	65	288	65	000	11	01
9	2	65	2	65	000	11	01
10	0	65	0	65	000	11	01
11	242	65	242	65	000	11	01
12	0	65	0	65	000	11	01
13	0	65	0	65	000	11	01
14	0	65	0	65	000	11	01
15	648	65	648	65	000	11	01
4	288	65	292	98	000	01	10
5	242	65	342	98	000	10	11
6	0	65	0	65	000	11	01
7	648	65	972	98	000	10	11
2	242	33	242	33	000	11	01
3	1134	99	1134	99	000	11	11
1	1134	99	0	0	110	10	00
0	1134	99	0	0	100	10	00

Table 7 Computed Values after Marking the first Selected HCC for Storage.

Node	$GrM_i.b$	$GrM_i.sp$	$GrF_i.b$	$GrF_i.sp$	$State_i$	chM_i	chF_i
8	288	65	288	65	000	11	01
9	2	65	2	65	000	11	01
10	0	65	0	65	000	11	01
11	242	65	242	65	000	11	01
12	0	65	0	65	000	11	01
13	0	65	0	65	000	11	01
14	0	65	0	65	000	11	01
15	0	0	0	0	110	00	00
4	288	65	292	98	000	01	10
5	242	65	342	98	000	10	11
6	0	65	0	65	000	11	01
7	0	65	0	0	100	01	00
2	584	131	584	131	000	11	11
3	0	65	0	0	100	01	00
1	584	131	0	0	100	01	00
0	584	131	0	0	100	10	00

Table 8 Computed Values after Marking the Second Selected HCC for Storage.

6. After possibly recursing to solutions in subtrees of c_i, the algorithm needs to recalculate the values of GrM_i and GrF_i, and all the corresponding chM_i and chF_i variables by executing the Candidate Path Selection phase on the visited nodes.

The only detail that we have not discussed is what happens if the selected path does not fit within the remaining space budget. In this case we simply traverse the selected path but mark for inclusion in the final solution only the highest coefficients in the path, such that the space constraint is not violated (we thus omit coefficients at the bottom of the path).

Example 9 Returning our attention to Table 6, we notice that based on the chM_0 and chF_0 bitmaps, the selected solution will need to store the coefficient c_0 and combine it with an HCC at its subtree (since chF_0=11). The bit $State_0(0)$ is thus set, while the bit $State_0(1)$ remains unset since this coefficient will surely not be the bottom-most coefficient in its HCC. Since node 0 has only one child node in the error tree, we must decide whether to consider that node 1 lies in its left or right subtree. We have selected the latter option and, thus, do not set the $State_0(2)$ bit. By recursing at node 1, we see based on the chM_1 and chF_1 bitmaps, that the coefficient c_1 needs to be stored, and that we do not need to recurse to children nodes. In this case, the bits $State_1(0)$ and $State_1(1)$ need to be set. Since c_1 became a new bottom-most coefficient at a new HCC, we recompute the GrF and GrM values at its two children nodes, in order to take into account that GrF paths from these subtrees could help lower the storage cost of c_1. Please note that the GrF values at nodes c_2 and c_3 both change (see Table 7), compared to the values in Table 6. Then, moving bottom-up we need to compute the GrF_1, GrM_1, GrF_0 and GrM_0 values, while properly setting the chM and chF bitmaps at nodes c_1 and c_0. The calculated entries at each node after marking for storage nodes c_0 and c_1 are depicted in Table 7.

In Table 8 we depict the calculated entries after the algorithm stores the next HCC, which contains the coefficients c_{15}, c_7 and c_3, and combines it with the first selected HCC. This can be easily identified by examining the $State$ bitmaps. The 5 entries that are set at the first bit (from the left) of these bitmaps translate to 5 stored coefficient values. The 1 entry that is set at the second bit of these bitmaps translates to 1 different HCCs. Since c_{15} does not have any children nodes, we do not needs to recompute the GrF and GrM at any of its descendant nodes. However, since c_1 seizes to be the bottom-most coefficient at a HCC, the GrF_2 and GrM_2 values are recalculated to take into account that no path storing c_2 can lower the storage cost of c_1.

At this stage of the algorithm, the last HCC, containing nodes c_{11}, c_5 and c_2 can be stored.

5.3 Storing the Selected Solution

The process of storing the selected HCCs follows a preorder traversal of the nodes in the error tree. At each visited node c_i, its input is a set (possibly empty) of *straddling* coefficient values. This set corresponds to coefficient values that belong to the same HCC, but where the lowest node in that HCC has not yet been visited. Any time the algorithms reaches a node c_i where the two bits $State_i(0)$ and $State_i(1)$ are both set, then the index/coordinate of c_i and its coefficient value along with the straddling coefficient values form a HCC. In this case, the input list to the both subtrees of c_i will be empty.

If only the $State_i(0)$ is set, but not the bit $State_i(1)$, then depending on the value of $State_i(2)$ the value c_i is attached to the list of straddling coefficient values for the appropriate subtree of c_i (the input list to the other subtree will be empty). If, finally, $State_i(0)$ is not set, then we simply recurse to the two subtrees with their inputs being empty lists of straddling coefficients.

5.4 Space and Running Time Complexities

For each node of the error tree there are $O(1)$ stored variables. Thus, the needed space is $O(N)$. At the initialization step, the calculation of the GrM_i, GrF_i, chM_i and chF_i variables requires $O(1)$ time. Then, the algorithm repeatedly marks at least one coefficient for storage. Thus, at most $O(\frac{B}{S_2})$ steps can be performed. At each step a path originating at the root of the error tree is traversed in order to mark for storage the nodes in GrM_0. This process visits at most $O(\log N)$ nodes. At each node, the recalculation of the GrM_i, GrF_i, chM_i and chF_i variables requires $O(1)$ time. Finally, the storage of the marked coefficients can be achieved in a single pass of the error tree. Thus, the overall running time complexity is $O(N+\frac{B}{S_2}\log N) = O(N+B\log N)$. Note that the running time complexity are on par with that of constructing a conventional synopsis — hence, no significant increase in data processing time is expected (see also Section 8).

Theorem 10 *The* HCGreedyL2 *algorithm constructs a HCWS given a space budget of B, in $O(N+B\log N)$ time using $O(N)$ space.*

6 HCGreedyL2-Str: A Streaming Greedy Algorithm

In order for our algorithms to adapt to streaming environments, we propose a streaming greedy algorithm, termed as HCGreedyL2-Str in our discussion, for our optimization problem. As expected, the HCGreedyL2-Str algorithm shares some common characteristics with the HCGreedyL2 algorithm in the way that it constructs candidate HCCs for storage.

6.1 Order of Processed Wavelet Coefficients

The algorithms proceeds by reading the data values one by one and by updating the (normalized) values of the wavelet coefficients. Note that the total number of data values to be read does not need to be known in advance, since the normalized value of a coefficient depends only on the number of data values that lie beneath it in the error tree (and, thus, from the difference in levels between the node and leaf coefficient values in the error tree). This process has well been documented in prior work [15].

When reading the n-th data value, the values of the wavelet coefficients that lie in path(n) are updated. According to Definition 3, a wavelet coefficient is *closed* only when all the data values that beneath it in its error tree have been read. Depending on the value of n, the number of coefficients that become closed due to a new data value ranges from 0 to $\log n+1$. These newly closed coefficients all belong to the bottom portion of $path(n)$ that originates from the last read data value and proceeds upwards in the error tree until $path(n)$ reaches the last error tree node for which the data value belongs to its right subtree. Our HCGreedyL2-Str algorithm processes these newly closed nodes of the error tree in a bottom-up fashion.

6.2 Used Data Structures

At each step of the algorithm, the current selection of HCCs is stored in a min-heap structure where the HCCs are ordered based on their per space benefit.[4] Each HCC is identified by its bottommost coefficient. We defer a detailed description and the implementation of this min-heap structure until later in this section.

The min-heap does not store each HCC explicitly, but rather a pointer to a structure containing: (i) The HCC; (ii) The benefit of the HCC; and (iii) The required space for the HCC. Please note that in order to guarantee that swapping any pair of HCCs in the min-heap can be performed in $O(1)$ time (and thus guarantee the worst time complexity of the First(), Pop() and Insert() operations, described in Section 6.4), we cannot simply store the HCCs in the min-heap, due to their variable size. We finally note that the number of different HCCs stored in the min-heap is obviously $O(\frac{B}{S_2}) = O(B)$.

Another important characteristic of our HCGreedyL2-Str algorithm is that it does not fully combine the stored HCCs, even though it accurately estimates their space requirements. This means that there may exist pairs of HCCs (i.e., HCC h_A and HCC h_B) in the min-heap such that parent(top(h_A)) = bottom(h_B). In such a case, even though h_A and h_B are not combined in one HCC, the storage overhead for bottom(h_B) is correctly set to S_2 in our algorithm. We explain in Section 6.5 why our HCGreedyL2-Str algorithm utilizes such an approach of storing HCCs.

Besides the min-heap structure our HCGreedyL2-Str algorithm also utilizes two hash tables, termed as *TopCoeff* and *BottomCoeff*, with a maximum of $O(\frac{B}{S_2})$ entries each. The *TopCoeff* (*BottomCoeff*) hash table maps the coordinate c_i of a coefficient to the stored HCC h_A in the min-heap, such that c_i = top(h_A) (c_i = bottom(h_A)). If the coordinate c_i is not the top (bottom) coefficient value stored in any HCC, then the *TopCoeff* (*BottomCoeff*) hash table does not contain an entry for it.

6.3 Operations at each node

For each processed node c_i our HCGreedyL2-Str algorithm generates a straddling candidate HCC, termed as $SGrF_i$. This straddling HCC is similar to GrF_i, in that it corresponds to the non-stored candidate path in the node's subtree with the estimated maximum per space benefit when storing c_i. Thus, its computation is similar, with the only difference that due to the streaming nature of the algorithm and the bottom-up way of processing closed coefficients, there is no way that c_i has already been stored in a HCC. Thus, the only choices considered for generating $SGrF_i$ are restricted to:

- Simply storing c_i. The space requirements of this choice is S_2 if either c_{2i} or c_{2i+1} has been stored, or S_1, otherwise. Please note that if c_{2i} or c_{2i+1} has been stored, then these coefficients must be the top coefficients in a stored HCC. This can be checked in $O(1)$ time by looking at the *TopCoeff* hash table. Let *Ben1* denote the per space benefit of this choice.
- Combining c_i with $SGrF_{2i}$ ($SGrF_{2i+1}$). The space requirements for $SGrF_i$ in this case is S_2+$SGrF_{2i}.sp$ (resp., S_2+ $SGrF_{2i+1}.sp$). Let *Ben2* (resp., *Ben3*) denote the per space benefit of this combination.

Given the aforementioned choices, $SGrF_i$ is set to:

1. $c_i \cup SGrF_{2i}$, if $Ben2 = \max\{Ben1, Ben2, Ben3\}$ and $Ben2$ is larger or equal to the per space benefit of $SGrF_{2i}$. In this case, $SGrF_{2i+1}$ cannot be of any further use in upper levels of the error tree. Thus, it is checked for insertion to the min-heap, by comparing its per space benefit to that of the stored HCC with the minimum per space benefit (see Section 6.4).
2. $c_i \cup SGrF_{2i+1}$, if $Ben3 = \max\{Ben1, Ben2, Ben3\}$ and, further, $Ben3$ is larger or equal to the per space benefit of $SGrF_{2i+1}$. In this case, $SGrF_{2i}$ cannot be of any further use in upper levels of the error tree. Thus, it is checked for insertion to the min-heap, by comparing its per space benefit to that of the stored HCC with the minimum per space benefit.
3. c_i, otherwise. In this case, $SGrF_{2i}$, $SGrF_{2i+1}$ are checked in succession for insertion to the min-heap, by comparing their per space benefit to that of the stored HCC with the minimum per space benefit.

Please note that, in the HCGreedyL2-Str algorithm, once we have computed the $SGrF_i$ coefficient for any node c_i, we no longer need to keep in main memory the straddling paths of its two subtrees.

6.4 Detailing the Operations of the Min-Heap

We now present the basic operations of the Min-Heap structure.

1. First(): Returns the stored HCC with the minimum per space benefit. This is straightforward. The operation requires $O(1)$ time.

[4] We can alternatively use any data structure, such as an AVL-tree, which provides a worst case cost of $O(\log B)$ for the (i) search of the stored item with the minimum per space benefit; (ii) the insertion of an item; and (iii) the deletion of an item.

2. Pop(): Removes the First() item. The operation adjusts the size of the Min-Heap, based on two factors:
   – The size of the removed HCC, termed as h_A in our discussion. This is available in the third field of the item (see Section 6.2 on how HCCs are stored).
   – Whether removing this item requires adjusting the space of some other HCC h_B. This case occurs when $\text{parent}(\text{top}(h_A)) = \text{bottom}(h_B)$ and the other child coefficient of $\text{bottom}(h_B)$ is not currently stored in the Min-Heap. The former can be tested by first probing the *BottomCoeff* hash table to see if $\text{parent}(\text{top}(h_A))$ exists as the bottom-most coefficient in a stored HCC. The latter can be tested by then probing the *TopCoeff* hash table for the other child of $\text{bottom}(h_B)$. If both conditions are satisfied, then the space requirements of h_B are adjusted and the standard heap procedure *heapifyUp()* is invoked in order to make sure that no conditions are violated in the path of the heap between the updated node and the root of the heap. This *heapifyUp()* operation requires $O(\log B)$ time.

   Thus. the Pop() operation requires a total of $O(\log B)$ time.
3. Insert(h_A): Inserts the given HCC h_A in the Min-Heap. This operation is presented in Algorithm 2. The running time requirements of the Insert() operation depend on the size of the inserted HCC and the number of popped HCCs (Lines 6-10). In the worst case, for a HCC containing $O(\log n)$ coefficient values, the operation may require $O(\log n \times \log B)$ time. However, an interesting observation is that for any HCC containing more than one coefficient values, the insert operation is performed only for the top coefficient value of the HCC. Thus, the amortized cost of the insert operation per processed wavelet coefficient remains $O(\log B)$.
4. Parse(): Scans the min-heap and extracts the stored HCCs in a compact form with size at most B. In order to perform this step we need to combine the HCCs stored in the Min-Heap. When checking each stored HCC h_A, we also check to see if there exists another unprocessed HCC h_B that needs to be processed before h_A, and such that h_A can be attached on top of h_B (so that their bitmaps are combined). This requires checking the *TopCoeff* hash table for the two children of $\text{bottom}(h_A)$. This step essentially creates a recursive processing of the HCCs similarly to a topological sort. Since the min-heap cannot store more than $O(\frac{B}{S_2})$ entries, this operation requires a total of $O(B)$ time.

6.5 Details and Remarks

A question that naturally arises is why we chose to store the current selection of the HCCs in a way that does not aggressively combine them, even though storage dependencies are indeed exploited. If we had pursued to aggressively merge stored HCCs, coefficient values with large benefits might end up in HCCs with several other small coefficient values, e.g., a HCC containing the coefficient values $\langle 800, 10, 20, 5\rangle$. This could potentially lead to HCCs with small to medium overall per space benefit, even though a part of them exhibits a large per space benefit. Please note that in the HCGreedyL2 algorithm, such a problem did not exist, as HCCs were attached to existing HCCs after exhibiting globally the best estimated per space benefit. Due to the streaming nature of the HCGreedyL2-Str algorithm, this global estimate cannot be achieved since future parts of the error tree have not been unveiled yet. Thus, we need to be careful in our decisions to aggressively merge HCCs.

```
procedure Insert($h_A$)
Input: HCC $h_A$ to insert into the Min-Heap.
1.  A min-heap structure $hcs$ is used to maintain the currently
    selected HCCs for storage
2.  Each entry in $hcs$ has 3 fields: (1) hc: the stored HCC,
    (2) ben: benefit of the HCC,
    (3) sp: space needed for storing the HCC.
3.  $UsedB \le B$ denotes the true space required to compactly store
    the HCCs of the Min-Heap.
4.  tophc = hcs.First()
5.  lastPopped = $\emptyset$
6.  while $UsedB + h_A.sp > B$ AND $\frac{tophc.ben}{tophc.sp} < \frac{h_A.sp}{h_A.ben}$ do
7.      lastPopped = tophc
8.      hcs.Pop(). Also update TopCoeff and BottomCoeff hash tables
9.      Update $UsedB$ based on discussion in Section 6.4
10. endwhile
11. Insert $h_A$ in the heap using standard heap operation.
    Update TopCoeff, BottomCoeff and $UsedB$.
12. if $UsedB < B$ then
        Trim sufficient coefficient values from lastPopped and reinsert
        it in the Max-Heap.
end
```

Fig. 2 Sketch of Insert algorithm.

6.6 Running Time and Space Requirements

Based on the analysis presented in Section 6.4, the operations associated with inserting a HCC in the Min-Heap cost a total of $O(\log B)$ time. The insert operation at some nodes may exhibit a higher cost but, as we explained in Section 6.4, this cost is amortized over the coefficient values that comprise the HCC. The space requirements are those of the Min-Heap, the two hash tables and the straddling coefficients. The Min-Heap and each hash table requires $O(B)$ space. Parsing the Min-Heap to extract the synopsis also requires $O(B)$ time. There can be at most $O(\log n)$ straddling coefficients, of total size $O(\log^2 n)$. Thus, the amortized running time requirements per processed data item are $O(\log B)$, while the space requirements are $O(B + \log^2 n)$.

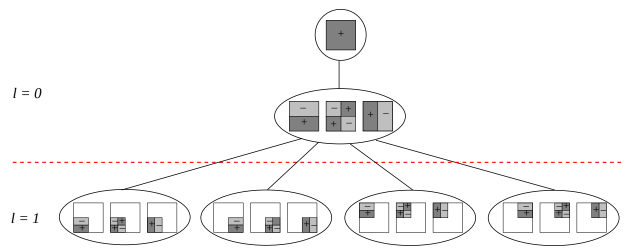


Fig. 3 Error-tree structure for the sixteen two-dimensional Haar coefficients for a 4×4 data array (data values omitted for clarity).

7 Extensions and Remarks

7.1 Multiple Dimensions

The Haar decomposition of a D-dimensional data array A results in a D-dimensional wavelet-coefficient array W_A with the same dimension ranges and number of entries. (The full details as well as efficient decomposition algorithms can be found in [2,28].) Consider a D-dimensional wavelet coefficient W in the wavelet-coefficient array W_A. W contributes to the reconstruction of a D-dimensional rectangular region of cells in the data array A (i.e., W's *support region*). Further, the sign of W's contribution ($+W$ or $-W$) can vary along the quadrants of its support region. The blank areas for each coefficient correspond to regions of A whose reconstruction is independent of the coefficient, i.e., the coefficient's contribution is 0. Each data cell in A can be accurately reconstructed by adding up the contributions (with the appropriate signs) of those coefficients whose support regions include the cell.

Error-tree structures for multi-dimensional Haar wavelets can be constructed (in linear time) in a manner similar to those for the one-dimensional case, but their semantics and structure are somewhat more complex. A major difference is that, in a D-dimensional error tree, each node (except for the root, i.e., the overall average) actually corresponds to a *set* of $2^D - 1$ wavelet coefficients that have the same support region but different quadrant signs and magnitudes for their contribution. Furthermore, each (non-root) node t in a D-dimensional error tree has 2^D children corresponding to the quadrants of the (common) support region of all coefficients in t.[5] If the maximum domain size amongst all dimensions is N_{max}, the height of the error tree will be equal to $\log N_{max}$. Note that the total domain size N can be as high as $N = N_{max}^D$ when all dimensions have equal domain size. Figure 3 depicts an example error-tree structure for a two-dimensional 4×4 data set.

[5] The number of children (coefficients) for an internal error-tree node can actually be less than 2^D (respectively, $2^D - 1$) when the sizes of the data dimensions are not all equal. In these situations, the exponent for 2 is determined by the number of dimensions that are *"active"* at the current level of the decomposition (i.e., those dimensions that are still being recursively split by averaging/differencing).

7.1.1 Multi-dimensional Hierarchically Compressed Wavelet Synopses

A *multidimensional hierarchically compressed wavelet synopsis* (MHCWS) groups nodes (not coefficients) into paths and thus requires additional information as to which coefficients of each node are included in the synopsis.

Definition 4 *The composite value NV of some node in the multidimensional error-tree is a pair* $\langle \text{NVBIT}, V \rangle$ *consisting of:*

- *A bitmap* NVBIT *of size* $2^D - 1$ *identifying which coefficient values are stored. The number of stored coefficient values is equal to the bits of* NVBIT *that are set.*
- *The set* V *of stored coefficient values.*

Having properly defined the composite value of a node we can now define a multidimensional hierarchically compressed wavelet coefficient as follows.

Definition 5 *A multidimensional hierarchically compressed (MHCC) wavelet coefficient is a triplet* $\langle \text{BIT}, C, \mathcal{NV} \rangle$ *consisting of:*

- *A bitmap* BIT *of size* $|\text{BIT}| \geq 1$*, denoting the storage of exactly* $|\text{BIT}|$ *node values.*
- *The coordinate/index* C *of any stored coefficient in the bottommost stored node.*
- *A set* $\mathcal{NV}$ *of* $|\text{BIT}|$ *stored composite values.*

We must note here that at any MHCC the coordinate of any stored coefficient in its bottommost stored node can be used, since the bitmap of that node's composite value can help determine which other coefficient values from the same node have also been stored.

7.1.2 Changes to the Algorithms

We now describe the necessary changes to the HCDynL2 and HCGreedyL2 algorithms for multi-dimensional data sets. The modifications to HCApprL2 are similar to the ones of HCDynL2.

Changes to HCDynL2. The extensions to the HCDynL2 algorithm are analogous to the corresponding extensions of prior DP techniques [9] to multi-dimensional data sets. In particular, when obtaining an optimal MHCWS given a space budget B, the algorithm given budget B should consider (i) the optimal benefit $\text{M}[i,B]$ assigning space B to the subtree rooted at node i; and (ii) the optimal benefit $\text{F}[i,B]$ assigning space B to the subtree rooted at node i *when at least one of the coefficients* of node i is forced to be stored (i.e., a composite value of the node is stored). The principle of optimality also holds in this case for $\text{M}[i,B]$ and $\text{F}[i,B]$, implying that optimal benefits at a node can be computed from optimal solutions of the node's subtrees.

At each node of the error tree, the optimal algorithm needs to decide how many coefficients, if any, of this node should be stored, whether they should be attached to some path of its children subtrees, and how much space to allocate to each child subtree. It should be noted that we only need to decide how many coefficients (from 1 to $2^D - 1$) of each node should be stored, as it can be easily shown that among all coefficient sets of k values, the set containing the coefficients with the k highest absolute normalized values exhibits the best benefit.

When the algorithm checks if a node should be included in the optimal solution but cannot be attached to any path of the children subtrees, the space requirement for this node is a function of the number $k \leq 2^D - 1$ of coefficients to be included (a choice to be made): $S_1(k) = \text{sizeof}(Coords) + 2^D + k \cdot \text{sizeof}(Value)$. Similarly, when the node at question can be attached to some path the space requirement is again a function of the number k of selected coefficients: $S_2(k) = 2^D + k \cdot \text{sizeof}(Value)$. Note that only in the first case the node "pays" for the overhead $\text{sizeof}(Coords)$ of creating a new $MHCC$.

At each node of the error tree the algorithm must perform two tasks: (i) sort the $2^D - 1$ coefficients of this node in $O(D2^D)$ time and $O(2^D)$ space; and (ii) for each space budget $0 \leq b \leq B^*$ choose the optimal split of space among the coefficients of this node and the 2^D children nodes. Note that, because a subtree rooted at a node at height l of the error tree can have up to $O(2^{Dl})$ nodes, the maximum alloted space at such a node is $B^* = \min\{B, O(2^{Dl})\}$. The second task can be performed in $O(2^D B^{*2})$, by solving a dynamic programming recurrence on a binary tree of height D constructed over the children nodes — for details refer to [9]. Using similar analysis with Section 3 and since there are at most $\frac{N_{max}^D}{2^{Dl}} = \frac{N}{2^{Dl}}$ nodes at height l it follows that the space complexity becomes $O(2^D N \log B)$, whereas the time complexity becomes $O(2^D NB)$.

Finally, note that the ratio of benefits between the HCDynL2 algorithm and the traditional technique can become as high as $\frac{1+\log N_{max} \times (2^D-1)}{m}$ for $m = \lfloor \frac{S_1 + \log N_{max} \times (2^D-1) \times S_2}{S_1 - 1} \rfloor$. The increased maximum value of the above ratio, when compared to the one-dimensional case, is not surprising, as in multi-dimensional data sets the existence of multiple coefficient values within each node of the error tree provides far more opportunities to exploit hierarchical relationships amongst stored coefficients, in order to reduce the storage overhead of their coordinates. Also, note that in the multidimensional case this storage overhead (and thus the size of S_1) increases with the number of dimensions, due to the increase in the number of the coefficient coordinates.

Changes to HCGreedyL2. For the HCGreedyL2 algorithm, when considering whether to include a node in a MHCC, or to attach it to a MHCC originating from one of the node's subtrees, we utilize the node's composite value that results in the best per space benefit. This can be accomplished by (i) sorting the node's coefficient values based on their normalized value; (ii) for $1 \leq j \leq 2^D - 1$ computing the per space benefit of the composite value that stores the node's j largest normalized values; and (iii) selecting the composite value with the overall best per space benefit. For nodes where, at some point of the algorithm's execution, some coefficient values have already been selected for storage, we only need to consider in the above case coefficient values that have not already been included in the solution and properly determine the space needed for their storage. The HCGreedyL2 algorithm, given a budget of B, requires $O(2^D N)$ space and only $O(D2^D N + 2^D B \log N_{max})$ time.

7.2 Dealing with Massive Data Sets

In order to improve the running time and space requirements of our algorithms for massive data sets, we can employ an initial thresholding step to discard coefficients with small values and apply our algorithms to the remaining $N_z \ll N$ coefficients. Such an approach is commonly followed for constructing wavelet synopses; the work in [28], for example, maintains only N_z coefficients after the decomposition to deal with sparse data sets of $N_z \ll N$ tuples. Preserving only N_z coefficients means that there can be at most N_z "important" nodes in the wavelet tree (in practice much fewer, as many large coefficients usually reside in a single node), which is a significant decrease compared to $N/2^D$, the total number of nodes.

More precisely, it is easy to see that all of our algorithms need to perform some computations to nodes that either (i) contain a non-zero coefficient value; or (ii) contain non-zero coefficient values at (at least) two of their subtrees. Thus, the total number of nodes where some computation needs to be performed is $O(2N_z - 1) = O(N_z)$. By sorting these nodes using a pre-order traversal it is easy to mark for each node: (i) the closest ancestor $anc(i)$ of i where computation needs to be performed; (ii) the subtree of $anc(i)$ that follows i; and (iii) the first descendant of i where computation needs to be performed. This process requires $O(N_z \log N_z)$ time, but allows for the execution of the algorithms with complexities that depend on N_z rather than N. Of course, some care is needed because the children of each node in the above "sparse" error-tree are not direct descendants, thus requiring proper calculation of the space needed when storing a node's composite value and combining it with a MHCC originating from one of the node's subtrees. Thus, when attaching a composite value to a MHCC that lies j levels below it in the sparse error tree, the value of S_2 must be set as follows: $S_2(k) = j \times (2^D - 1) + j + k \cdot \text{sizeof}(Value)$. The first summand in the above formula is due to the storage of the

NVBIT bitmaps for both the current node and all the intermediate, missing nodes until reaching the MHCC of the descendant node. The second summand determines the number of these bitmaps, while the third summand is due to the storage of k coefficient values in the node. Please note that each node of the sparse error-tree may exhibit different S_2 values for each of its subtrees, due to the potentially different resolution levels of each subtree's root node.

7.3 Optimizing for Other Error Metrics

All algorithms presented here can be made to optimize for any weighted $\mathcal{L}_2^w$ error metric. These error metrics include the sum squared relative error with sanity bound s (set $w_i = \frac{1}{\max\{d_i,s\}}$), and the expected sum squared error when queries are drawn from a workload distribution, in which case the weights correspond to the probability of occurrence for each query (set $w_i = p_i$).

For the weighted $\mathcal{L}_2^w$ metric and using the standard Haar decomposition process the Parseval theorem does not apply and hence Problem 2 does not follow from Problem 1. However the recent work of [27] demonstrated that the Parseval theorem applies when the decomposition process is altered to incorporate the weights. The result is a modified Haar basis for which the Parseval applies and, therefore, an analogous to Problem 2 formulation exists and our algorithms require no additional changes.

7.4 Query Performance Issues

For a synopsis size of B, due to the use of a variable-length header for the stored HCC coefficients, the retrieval of a single coefficient value requires $O(B)$ time, in contrast to $O(\min\{B, \log N\})$ time for the conventional wavelet synopses, where binary search is employed if the stored coefficients are sorted based on their coordinates. While this may seem as a potentially large increase in the resulting query time, we need to make two important observations: (i) The used synopses are typically memory resident and of small size ($B \ll N$); and (ii) To answer even point queries, $O(\log N)$ coefficients need to be retrieved. The number of retrieved coefficients is increased even more if a query that requires the evaluation of multiple individual data values (or data values in multiple areas of the data) is issued. This has the effect that a linear scan of the synopsis, to retrieve at batch all the desired coefficients, even in conventional wavelet synopses, is often as efficient as performing a logarithmic (or larger) number of binary searches in the synopsis. Thus, we expect that any potential running time deterioration due to the use of our proposed technique will be minimal. On the other hand, the improvements in the obtained accuracy achieved by the use of HCWS can be significant, as shown in Section 8.

8 Experimental Study

In this section, we present an extensive experimental study of our proposed algorithms for constructing hierarchically compressed wavelet synopses over large data sets. Our objective is to evaluate the scalability and the obtained accuracy of our algorithms when compared to conventional synopses. Our main findings include:

- **Improved Space Utilization.** The algorithms presented in this work create HCWS that consistently exhibit significant reductions in terms of the sum squared error of the approximation due to the improved storage utilization of the selected wavelet coefficients.
- **Efficient, Near-Optimal Greedy HCWS Construction.** Even though the HCGreedyL2 algorithm does not provide any guarantees on the quality of the obtained solution, in all of our experiments it provided near optimal results. At the same time, the HCGreedyL2 algorithm exhibits running time and space requirements on par with the conventional synopsis construction method. Moreover, our proposed HCGreedyL2-Str algorithm consistently produces HCWS with errors very close to those of the HCGreedyL2 algorithm.

8.1 Testbed and Methodology

Techniques and Implementation Details. We compare the algorithms HCDynL2, HCApprL2, HCGreedyL2, HCGreedyL2-Str introduced in this paper against the conventional synopsis construction algorithm denoted as Classic. The Classic algorithm utilizes a heap to identify the coefficients with the largest absolute normalized values, while not exceeding the available space budget. All algorithms were implemented in C++ and the experiments reported here were performed on a 2.4 GHz machine.

Data Sets. We have performed an extensive experimental study with several one-dimensional synthetic and real-life data sets; we present here the most significant findings. Each synthetic data set, termed `Zipfian`, is produced by generating 50 different zipfian distributions with the same skew parameter (where the values are placed in random locations of the data) and then summing up these 50 smaller data sets. We vary the domain size from $N = 2^{14}$ up to $2^{24} = 16,777,216$ and examine two values of the zipfian parameter, $z = 0.7$ and $z = 1.2$, i.e., average and high skew respectively. The first real data set, denoted as `Weather`[6], contains $N = 65,536$

[6] Data available at:
`http://www-k12.atmos.washington.edu/k12/grayskies/`

solar irradiance measurements obtained from a station at the University of Washington. The second real data set, denoted as Light, consists of light measurements from the Intel Labs data set [7]. In all experiments involving Light, we use the measurements of the sixth mote (sensor) of this data set.

Performance Metrics. We first investigate the running time scalability of our algorithms when varying the available synopsis budget, the data domain size and the ε parameter for the HCApprL2 algorithm. In order to assess the quality of the constructed HCWS we measure the sum squared error (SSE). To emphasize on the effectiveness over conventional synopses: (i) we explicitly measure the SSE increase of Classic relative to HCGreedyL2; and (ii) show how much more space (*space savings*) we would need to allocate to a conventional synopsis in order for it to become as accurate as our constructed HCWS. In a graph depicting the resulting SSE by all algorithms when varying the synopsis size, the SSE increase in absolute value can be measured at each point by the vertical distance between the graph of the Classic technique from the graph of either the HCDynL2, the HCApprL2, the HCGreedyL2 or the HCGreedyL2-Str algorithm. Correspondingly, in the same graph, the *space savings* of our algorithms can be (roughly) measured, for any space budget assigned to our algorithms, by the horizontal distance to the right, starting of course at the point of the graph corresponding to our technique and for the desired space budget, until we meet the graph (error) of the Classic algorithm. Recall that the goal of deploying a HCWS is to achieve better storage utilization and to improve the accuracy of the synopsis by storing, within a given space budget, a larger number of "important" coefficient values than a traditional wavelet synopsis. The space savings essentially provide us with an insight on how many "important" wavelet coefficients the HCWS contains, in addition to the ones selected by the Classic algorithm, that are responsible for the achieved SSE reduction (and, thus, how much can our algorithms exploit hierarchical relationships amongst coefficient values selected for storage). The combination of the two performance metrics also reveals some helpful characteristics on the distribution of the coefficient values. For example, assume that our algorithms consistently result in half the error achieved by the Classic algorithm, but that the space savings increase (decrease) as the synopsis size increases. This implies that as the synopsis size increases, and more coefficient values are stored, the number of non-stored coefficient values that are responsible for half of the *remaining* SSE also increases (decreases), since the Classic algorithm requires increasingly more (less) space to reduce its SSE by 50%.

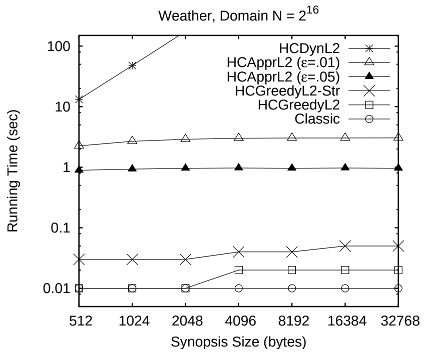
Further, we explicitly measure the deviation of the error exhibited by the solution of our HCGreedyL2 algorithm, when compared to the corresponding optimal error exhibited by the solution of our HCDynL2 algorithm, when varying either the available synopsis budget, or the data domain size. We also measure the errors achieved by our HCGreedyL2-Str algorithm, when compared to the corresponding errors of our HCGreedyL2 algorithm. Finally, we plot the approximation ratio achieved by the HCApprL2 algorithm against the theoretical bound.

8.2 Experimental Results

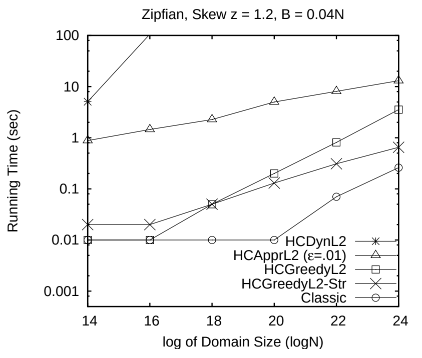
Scalability. Figure 4 investigates the scalability, in terms of the total running time, for all methods while the synopsis size and the domain size is varied. For the HCApprL2 algorithm we also plot its running time when varying the approximation parameter. Figure 4(a) presents the running time for the Weather data set when the available synopsis size increases from 512 to 32,768 bytes. The approximation parameter for the HCApprL2 algorithm was set to $\varepsilon = 0.05$ and 0.01. Please note that logarithmic axes are used for both the resulting running time and the synopsis size. In this experiment, the HCGreedyL2 and HCGreedyL2-Str algorithms consistently construct a HCWS within a few hundredths of a second, and almost as fast (with an increase in running time by a factor between 2 and 5) as Classic constructs a conventional synopsis. The HCDynL2 algorithm could not construct large HCWSs within a reasonable time, as depicted on Figure 4(a), due to its linear dependency on B. Similar trends were observed for all data sets and, thus, the graphs for the HCDynL2 algorithm are often omitted.

Figure 4(b) illustrates the scalability of the algorithms as the domain size increases from 2^{14} up to 2^{24} for the Zipfian data set with a skew parameter of 1.2. The synopsis size is set to a fixed percentage (4%) of the original data size. Therefore, the time complexity of HCDynL2 essentially becomes quadratic on the domain size. This is depicted on Figure 4(b), as the running time of HCDynL2 for domains larger than 2^{16} becomes prohibitive, while HCGreedyL2 can construct a HCWS in about 3.5 seconds, even for a domain size of 2^{24}. The running time of the streaming variant HCGreedyL2-Str increases at a lower rate than that of HCGreedyL2, as the domain size increases. This is attributed to the fact that the running time complexity for the HCGreedyL2-Str algorithm is based on a pessimistic case where every HCC tested for insertion in the min-heap requires $O(\log B)$ time. In practice, most of the HCCs in large domains do not have a sufficiently large per space benefit to be inserted into the min-heap, thus requiring only $O(1)$ time for them. Finally, note that even if it exhibits running times that are up to 2 orders of magnitude larger than the ones of HCGreedyL2, the HCApprL2 algorithm scales significantly better than the HCDynL2 algorithm.

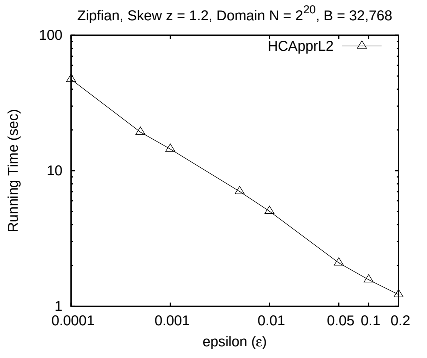
Figure 4(c) plots the running time of HCApprL2 as the approximation parameter ranges from $\varepsilon = 0.0001$ to 0.2 for the Zipfian data set with a skew parameter of 1.2, $N = 2^{20}$ data values and a fixed value of $B = 32768$. As the approximation



(a) Running Time vs Synopsis Size

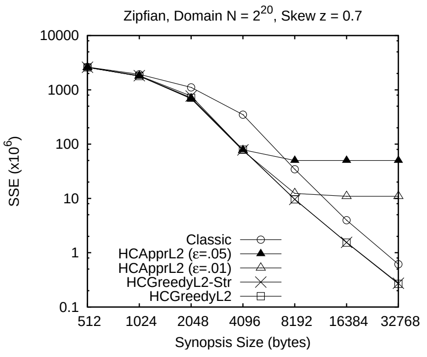


(b) Running Time vs Domain Size

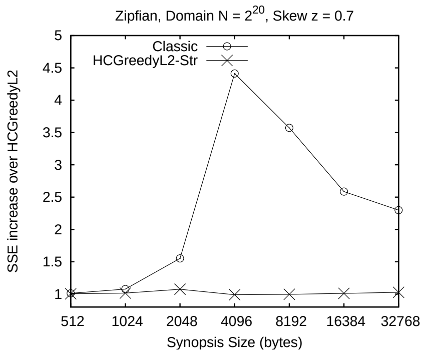


(c) Running Time vs ε

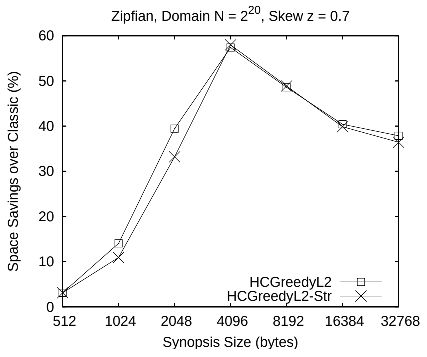
Fig. 4 Running Time Performance of all Algorithms



(a) SSE vs Synopsis Size



(b) SSE Increase vs Synopsis Size



(c) Space Savings vs Synopsis Size

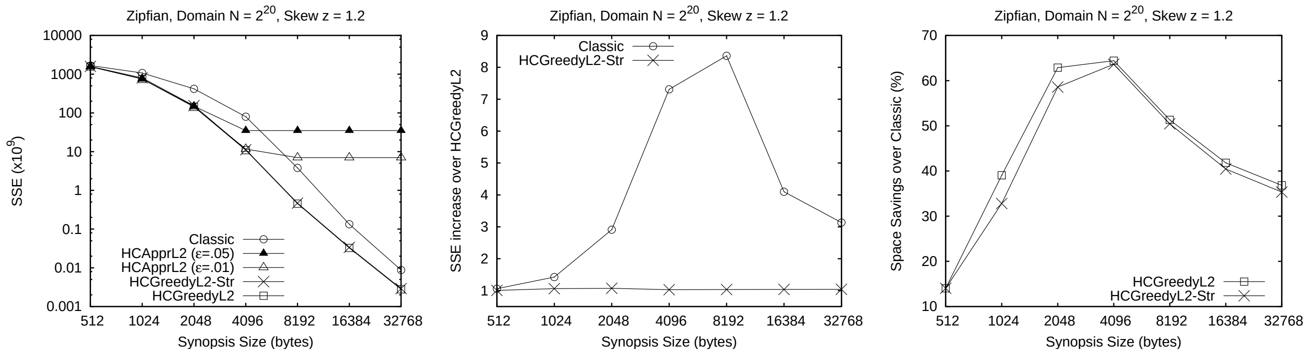
Fig. 5 HCWS Quality vs Synopsis Size for `Zipfian` $z = 0.7, N = 2^{20}$

requirements relax, the running time of HCApprL2 decreases exponentially.

HCWS Quality. In Figures 5, 6, 7 and 8 we investigate the quality of the HCWS synopses for the four data sets described in Section 8.1, as we vary the synopsis size from 512 to 32,768 bytes. For all data sets, we measure the SSE of the resulting synopses.

Figure 5(a) plots the SSE for all methods on the `Zipfian` data set with the average skew value. The HCGreedyL2 algorithm consistently constructs a synopsis with significantly smaller errors compared to a conventional synopsis. Moreover, the HCGreedyL2-Str algorithm achieves similar benefits, as its performance closely matches that of HCGreedyL2. On the other hand, the accuracy of the HCApprL2 algorithm quickly approaches the point where the algorithm manages to construct a synopsis that has captured a sufficiently large fraction $1/(1+\varepsilon)$ of the data's energy (and it is, thus, certainly also within the same $1/(1+\varepsilon)$ factor from the optimal algorithm) — hence, further increasing the budget leads to the HCApprL2 algorithm constructing the same synopsis. Figure 5(b) plots the *SSE increase* (i.e., the ratio of the SSE errors) of Classic and HCGreedyL2-Str over HCGreedyL2. We first observe that for a space budget of $B = 4096$, HCGreedyL2 constructs an HCWS that has almost 4.5 times less SSE than a conventional synopsis. HCGreedyL2-Str constructs synopses with similar SSE compared to HCGreedyL2. Comparing the two greedy heuristics, HCGreedyL2-Str achieves 2% lower SSE in the best case ($B = 4096$), and 7.4% larger SSE in the worst case ($B = 2048$), than HCGreedyL2. Figure 5(b) illustrates the space savings of the two greedy algorithms compared to a conventional synopsis that would achieve the same SSE. As the synopsis size increases, the space savings of our algorithms in absolute values (i.e., in bytes) increase as well. In relative terms (i.e., as a percentage to the synopsis size), the best case for our methods appears for $B = 4096$, where a HCWS requires 57.4% less space than a conventional synopsis. The space savings of HCGreedyL2-Str show a similar trend with a maximum savings of 58% for $B = 4096$.

Figure 6 repeats the above setup using the `Zipfian` data set with high skew ($z = 1.2$). The higher skew results in a more compressible data set with the SSE decreasing rapidly with B, as depicted on Figure 6(a). In this data set, constructing hierarchically compressed synopses proves highly beneficial as shown in Figures 6(b) and 6(c). HCGreedyL2 construct a synopsis with up to 8.3 times lower SSE than Classic (for $B = 8192$). Furthermore, the space savings of the HCGreedyL2 algorithm are significant (up to 64% for a synopsis size of $B = 4096$). Note that HCGreedyL2-Str constructs



(a) SSE vs Synopsis Size (b) SSE Increase vs Synopsis Size (c) Space Savings vs Synopsis Size

Fig. 6 HCWS Quality vs Synopsis Size for `Zipfian` $z = 1.2$, $N = 2^{20}$

synopses with marginally increased SSE compared to HCGreedyL2 (up to 7% increase, with an average increase of 2%).

Figures 7 and 8 repeat the previous experimental setup for the real-life data sets, `Weather` and `Light`, respectively. For both data sets, the benefits, in terms of the reduction in the SSE, increase with the synopsis size. For the `Weather` data set, the HCGreedyL2 algorithm results in up to 2.36 times lower SSE (for $B = 32768$), as shown in Figure 7(b). On the other hand, Figure 8(b) shows that in the `Light` data set, the HCGreedyL2 algorithm achieves a reduction in SSE of up to 4.7 times (for $B = 32768$). For both real data sets, and for synopsis sizes larger than 1024 bytes, the space savings of our methods are consistently high (please note our earlier discussion that the benefits in absolute terms continuously increase in these cases as well, even though the relative space savings start decreasing at some point), as shown in Figures 7(c) and 8(c).

The effect of the domain size in the performance of our algorithms is illustrated in Figure 9. In this setup we use the `Zipfian` data set with the high skew value ($z = 1.2$) and vary the domain size from $N = 2^{14}$ up 2^{24}, while maintaining the synopsis size to 4% of N. Similar findings hold for other space ratios as well as for the average skew data set. As seen in Figure 9(a), both greedy variants consistently construct synopses with lower SSE (up to 7.4 times) than Classic. Similarly, our greedy heuristics are able to achieve significant space savings (up to 69% for the HCGreedyL2 algorithm and up to 66% for the HCGreedyL2-Str algorithm), compared to the Classic algorithm.

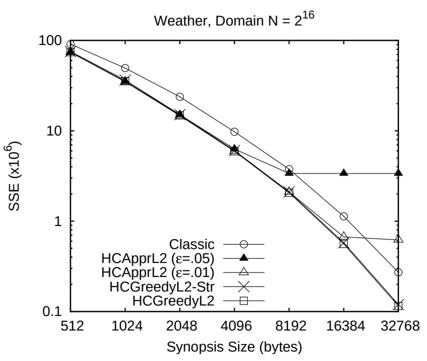
HCGreedyL2, HCGreedyL2-Str and HCApprL2 Accuracy. The HCGreedyL2 and HCGreedyL2-Str algorithms, as we have seen, require only frugal time and space in order to construct a wavelet synopsis when compared to the optimal HCDynL2 algorithm. A question that naturally arises is how close is the error of a HCWS constructed by the greedy algorithms to the one of the optimal HCWS. Thus, in the following set of experiments we measure the SSE increase incurred by HCGreedyL2 and HCGreedyL2-Str when constructing a HCWS — this is, essentially, the ratio between the errors of the greedy variants and the HCDynL2 algorithms.

Figure 10(a) shows the SSE increase ratio for the `Weather` data set as the space budget is varied from 512 to 4096 bytes. It is easy to see that the error of the HCWS obtained by HCGreedyL2 (HCGreedyL2-Str) is always within 1.6% (4.6%) of the error achieved by the optimal HCWS. Figure 10(b) shows the SSE increase for the `Zipfian` data set as the domain size varies from 2^{10} to 2^{15}, while the synopsis size is set to 1% of the original data. Such a setup is chosen so that the HCDynL2 algorithm, which provides the optimal HCWS, can execute within the available memory and within a time window of one hour. Again, the error of the HCWS obtained by HCGreedyL2 is within 2.2% of the error achieved by the optimal HCWS, while in 3 cases the HCGreedyL2 algorithm produced the optimal solution. Regarding the accuracy of HCGreedyL2-Str, note that in the worst case it produces HCWS with error which is within 12% (and with an average value of 4%) of the optimal.

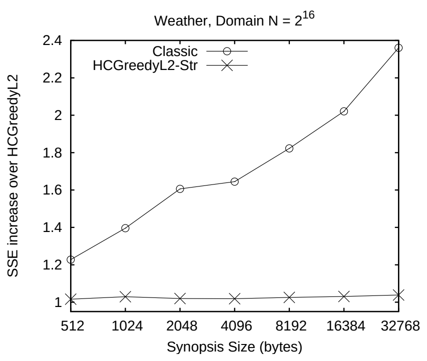
To measure the quality of HCApprL2, we plot the approximation ratio (benefit of constructed HCWS over the benefit of the optimal HCWS) for HCApprL2 as ε varies in Figure 10(c). Further, we also plot the theoretical bound of $\frac{1}{1+\varepsilon}$ for reference. Observe that HCApprL2 consistently achieves a HCWS with approximation ratio significantly larger than the theoretical bound.

9 Related Work

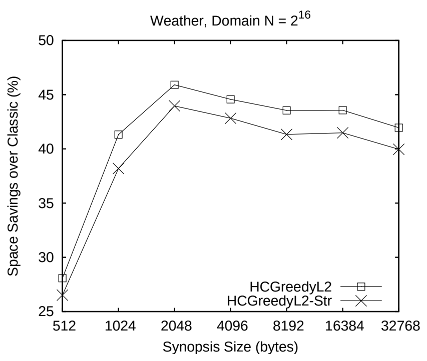
The wavelet decomposition has been applied successfully as a data reduction mechanism in a wide variety of applications. Wavelets have been used in answering range-sum aggregate queries over data cubes [29,28] and in selectivity estimation [21]. The effectiveness of Haar wavelets as a general-purpose approximate query processing tool was demonstrated in [2]. For the case of data sets with multiple measures the authors in [5,6] introduce the notion of



(a) SSE vs Synopsis Size

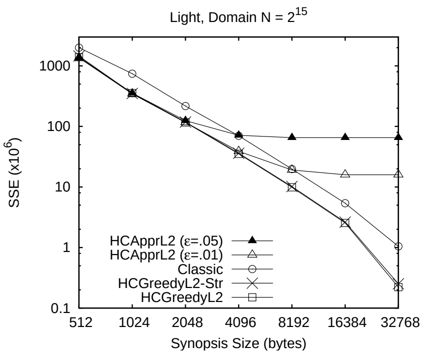


(b) SSE Increase vs Synopsis Size

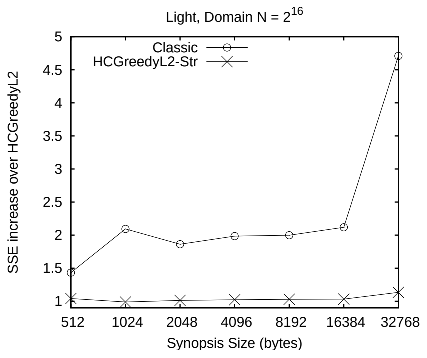


(c) Space Savings vs Synopsis Size

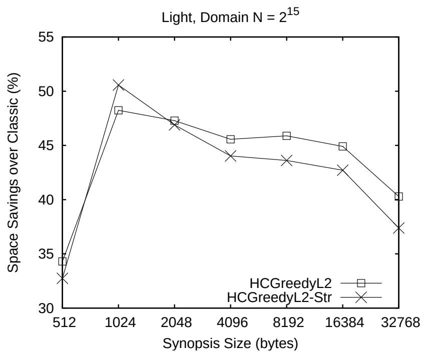
Fig. 7 HCWS Quality vs Synopsis Size for `Weather`, $N = 2^{16}$



(a) SSE vs Synopsis Size

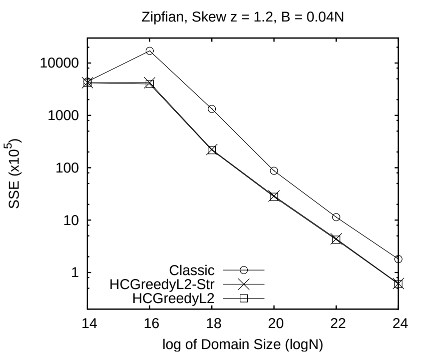


(b) SSE Increase vs Synopsis Size

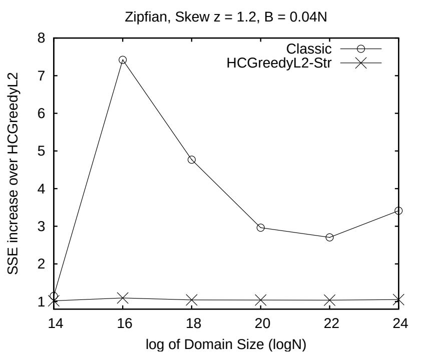


(c) Space Savings vs Synopsis Size

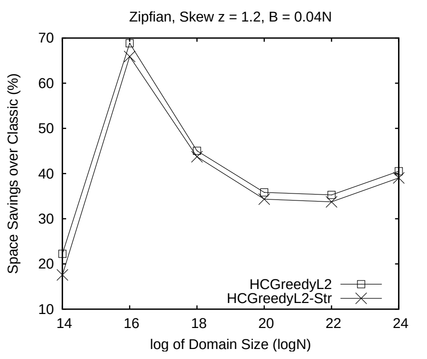
Fig. 8 HCWS Quality vs Synopsis Size for `Light`, $N = 2^{15}$



(a) SSE vs Domain Size



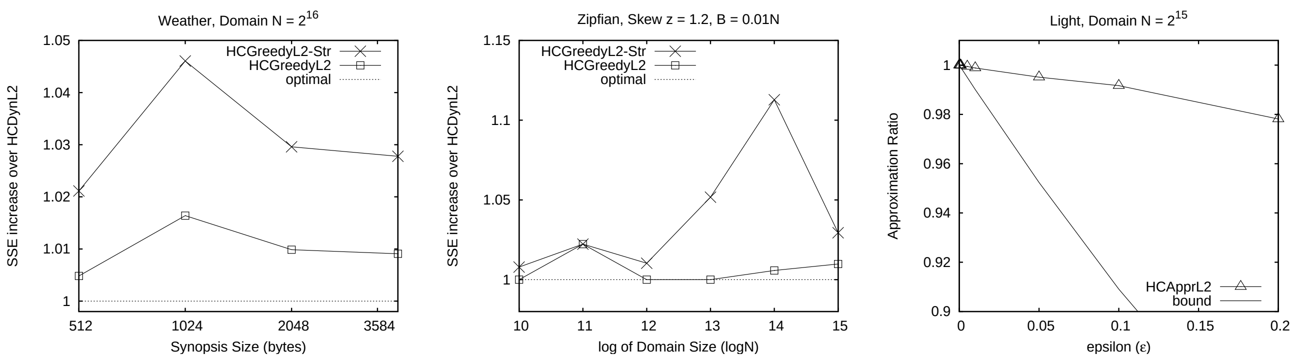
(b) SSE Increase vs Domain Size



(c) Space Savings vs Domain Size

Fig. 9 HCWS Quality vs Domain Size for `Zipfian` $z = 1.2$, $B = 0.04N$

extended wavelets; some further improvements were presented in [15], where a streaming algorithm for the above problem is also introduced. A common characteristic of the work in [5,6,15] with this paper is that all of these papers seek to exploit storage dependencies amongst stored coefficient values. However, these storage dependencies are only amongst coefficient values, of different measures, that correspond to the same coefficient coordinates. Thus, the storage overhead of a coefficient value is not influenced by whether other coefficient values in the path towards the root of the error tree have also been stored. This observation implied that the error tree structure does not need to be taken into account at all. Due to this crucial difference with the problem tackled in this paper, the techniques in [5,6,15] cannot be used to solve our optimization problem, and are in fact completely different than the techniques that we propose here. Similarly, extending our proposed algorithms of this paper to multi-measure data sets requires significant modifications and is an interesting topic of future work. I/O efficient algorithms for maintenance tasks were presented in [16].



(a) HCGreedyL2 Accuracy vs Synopsis Size (b) HCGreedyL2 Accuracy vs Domain Size (c) HCApprL2 Accuracy vs ε

Fig. 10 HCGreedyL2, HCGreedyL2-Str, and HCApprL2 Accuracy

In a previous work [1], the authors proposed the storage of coefficient values forming a rooted subtree of the error tree. While such an approach was guaranteed to provide a worse benefit than the conventional thresholding process, their techniques performed well for signal de-noising purposes. However, this work neither considered reducing the storage overhead of the wavelet coefficients' coordinates, nor did it incorporate such an objective in the thresholding process. Moreover, the requirement that rooted subtrees be stored, rather than arbitrary paths of coefficient values, often required the storage of many small coefficient values that simply happened to lie on root-to-leaf paths of other large coefficient values.

A lot of recent work focus on constructing wavelet synopses that minimize error metrics other than SSE. The work in [8] constructs wavelet synopses that probabilistically minimize the maximum relative or absolute error incurred for reconstructing any data value. The work in [4] provides a sparse approximation scheme for the same problem. While solving entirely an entirely different problem, our HCApprL2 algorithm shares in fact several common characteristics in its operation with the algorithm in [4]. However, the HCApprL2 algorithm is slightly more complicated due to the two mutually recursive functions that it needs to approximate, and the increased number of breakpoint combinations of children nodes that it needs to consider in its operation. Such details also lead to a more tedious proof of its correctness. The work in [9] showed that it is possible to deterministically construct wavelet synopses for the same problem as in [8] and provided a novel dynamic programming recurrence, extensible [10] to any distributive error metric. Similar ideas were employed in [23] to construct optimal synopses in sub-quadratic time for a particular class of error metrics. Further, the work in [12] improves the space requirements of the aforementioned dynamic programming algorithms. For the same problem of optimal weighted synopses, the work in [27] constructs a wavelet-like basis so that the Parseval's theorem applies and, thus, the conventional greedy thresholding technique can be used. Assuming all range-sum queries are of equal importance, the authors in [20] proved that the heuristics employed in [21] are in fact optimal. The works in [13, 14] showed that for error metrics other than SSE, keeping the original coefficient values is suboptimal. Hence, they propose approximation algorithms for constructing *unrestricted* wavelet synopses that involve searching for the best value to assign for each coefficient stored.

Wavelets have also found broad use in data stream environments. The dynamic maintenance of Haar synopses was first studied in [22]. The works in [3, 11] use sketching techniques for maintaining conventional wavelet synopses over rapidly changing data streams. The approximation schemes of [13, 14] for unrestricted wavelet synopses are also extensible for the case of time-series data streams. A fast greedy algorithm for maximum-error metrics was introduced in [18] for the problem of constructing wavelet synopses over time-series data streams.

10 Conclusions

In this paper, we proposed a novel compression scheme for constructing wavelet synopses, termed Hierarchically Compressed Wavelet Synopses (HCWS). Our scheme seeks to improve the storage utilization of the wavelet coefficients and, thus, achieve improved accuracy to user queries by reducing the storage overhead of their coordinates. To accomplish this goal, our techniques exploit the hierarchical dependencies among wavelet coefficients that often arise in real data sets due to the existence of large spikes among neighboring data values and, more importantly, incorporate this goal in the synopsis construction process. We initially presented a dynamic programming algorithm, along with a streaming version of this algorithm, for constructing an optimal HCWS that minimizes the sum squared error given a space budget. We demonstrated that while in the worst case the benefit of our DP solution is only equal to the benefit of the conventional thresholding approach, it can often be significantly larger, thus achieving significantly reduced errors in the data reconstruction. We then presented an approximation algorithm with tunable guarantees leveraging a trade-off between synopsis accuracy and running time. Finally, we presented a fast greedy algorithm, along with a streaming version of this algorithm. We demonstrated that both of our greedy heuristics always exhibited near-optimal results in our experimental evaluation, with a running time on par with conventional thresholding algorithms. Extensions for multidimensional data sets, running time improvements for massive data sets and generalization to other error metrics were also introduced. Extensive experimental results demonstrate the effectiveness of HCWS against conventional synopsis techniques. As a concluding remark, future work directions include the design of algorithms for creating HCWS that optimize for an even wider class of error metrics.

References

1. Baraniuk, R., Jones, D.: A signal-dependent time-frequency representation: fast algorithm for optimal kernel design. ISP **42**(1), 134–146 (1994)
2. Chakrabarti, K., Garofalakis, M.N., Rastogi, R., Shim, K.: Approximate query processing using wavelets. In: Proceedings of the International Conference on Very Large Data Bases (VLDB), pp. 111–122 (2000)
3. Cormode, G., Garofalakis, M., Sacharidis, D.: Fast approximate wavelet tracking on streams. In: Proceedings of the International Conference on Extending Database Technology (EDBT) (2006)
4. Deligiannakis, A., Garofalakis, M., Roussopoulos, N.: A fast approximation scheme for probabilistic wavelet synopses. In: Proceedings of the 17th International Conference on Scientific and Statistical Database Management (SSDBM) (2005)
5. Deligiannakis, A., Garofalakis, M., Roussopoulos, N.: Extended wavelets for multiple measures. ACM Transactions on Database Systems **32**(2) (2007)
6. Deligiannakis, A., Roussopoulos, N.: Extended wavelets for multiple measures. In: Proceedings of ACM International Conference on Management of Data (SIGMOD), pp. 229–240 (2003)
7. Deshpande, A., Guestrin, C., Madden, S., Hellerstein, J., Hong, W.: Model-Driven Data Acquisition in Sensor Networks. In: VLDB (2004)
8. Garofalakis, M., Gibbons, P.B.: Wavelet synopses with error guarantees. In: Proceedings of ACM International Conference on Management of Data (SIGMOD), pp. 476–487 (2002)
9. Garofalakis, M., Kumar, A.: Deterministic wavelet thresholding for maximum-error metrics. In: Proceedings of the ACM Symposium on Principles of Database Systems (PODS), pp. 166–176 (2004)
10. Garofalakis, M., Kumar, A.: Wavelet synopses for general error metrics. ACM Transactions on Database Systems **30**(4), 888–928 (2005)
11. Gilbert, A.C., Kotidis, Y., Muthukrishnan, S., Strauss, M.J.: Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In: Proceedings of the International Conference on Very Large Data Bases (VLDB) (2001)
12. Guha, S.: Space efficiency in synopsis construction algorithms. In: Proceedings of the International Conference on Very Large Data Bases (VLDB), pp. 409–420 (2005)
13. Guha, S., Harb, B.: Wavelet synopsis for data streams: minimizing non-euclidean error. In: Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining (KDD), pp. 88–97 (2005)
14. Guha, S., Harb, B.: Approximation algorithms for wavelet transform coding of data streams. In: Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA) (2006)
15. Guha, S., Kim, C., Shim, K.: Xwave: Approximate extended wavelets for streaming data. In: Proceedings of the International Conference on Very Large Data Bases (VLDB), pp. 288–299 (2004)
16. Jahangiri, M., Sacharidis, D., Shahabi, C.: Shift-Split: I/O efficient maintenance of wavelet-transformed multidimensional data. In: Proceedings of ACM International Conference on Management of Data (SIGMOD) (2005)
17. Jawerth, B., Sweldens, W.: An Overview of Wavelet Based Multiresolution Analyses. SIAM Review **36**(3), 377–412 (1994)
18. Karras, P., Mamoulis, N.: One-pass wavelet synopses for maximum-error metrics. In: Proceedings of the International Conference on Very Large Data Bases (VLDB), pp. 421–432 (2005)
19. Mallat, S.: A Wavelet Tour of Signal Processing. Academic Press, 2nd edition (1999)
20. Matias, Y., Urieli, D.: Inner-product based wavelet synopses for range-sum queries. In: Proceedings of the 14th Annual European Symposium on Algorithms (ESA), pp. 504–515 (2006)
21. Matias, Y., Vitter, J.S., Wang, M.: Wavelet-based histograms for selectivity estimation. In: Proceedings of ACM International Conference on Management of Data (SIGMOD), pp. 448–459 (1998)
22. Matias, Y., Vitter, J.S., Wang, M.: Dynamic maintenance of wavelet-based histograms. In: Proceedings of International Conference on Very Large Data Bases (VLDB), pp. 101–110 (2000)
23. Muthukrishnan, S.: Subquadratic algorithms for workload-aware haar wavelet synopses. In: Proceedings of the IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS) (2005)
24. Natse, A., Rastogi, R., Shim, K.: WALRUS: A Similarity Retrieval Algorithm for Image Databases. In: Proceedings of ACM International Conference on Management of Data (SIGMOD) (1999)
25. Poosala, V., Ioannidis, Y.E.: Selectivity Estimation Without the Attribute Value Independence Assumption. In: VLDB (1997)
26. Stollnitz, E.J., Derose, T.D., Salesin, D.H.: Wavelets for computer graphics: theory and applications. Morgan Kaufmann Publishers Inc. (1996)
27. Urieli, D., Matias, Y.: Optimal workload-based weighted wavelet synopses. In: Proceedings of International Conference on Database Theory (ICDT) (2005)
28. Vitter, J.S., Wang, M.: Approximate computation of multidimensional aggregates of sparse data using wavelets. In: Proceedings of ACM International Conference on Management of Data (SIGMOD), pp. 193–204. ACM Press (1999)
29. Vitter, J.S., Wang, M., Iyer, B.R.: Data cube approximation and histograms via wavelets. In: Proceedings of the International Conference on Information and Knowledge Management (CIKM), pp. 96–104 (1998)