Extended Wavelets for Multiple Measures

Antonios Deligiannakis
Dept. of Computer Science
University of Maryland, College Park
adeli@cs.umd.edu

Nick Roussopoulos
Dept. of Computer Science
University of Maryland, College Park
nick@cs.umd.edu

Abstract

While work in recent years has demonstrated that wavelets can be efficiently used to compress large quantities of data and provide fast and fairly accurate answers to queries, little emphasis has been placed on using wavelets in approximating datasets containing multiple measures. Existing decomposition approaches will either operate on each measure individually, or treat all measures as a vector of values and process them simultaneously. We show in this paper that the resulting *individual* or *combined* storage approaches for the wavelet coefficients of different measures that stem from these existing algorithms may lead to suboptimal storage utilization, which results to reduced accuracy to queries. To alleviate this problem, we introduce in this work the notion of an *extended* wavelet coefficient as a flexible storage method for the wavelet coefficients, and propose novel algorithms for selecting which *extended* wavelet coefficients to retain under a given storage constraint. Experimental results with both real and synthetic datasets demonstrate that our approach achieves improved accuracy to queries when compared to existing techniques.

1 Introduction

Decision Support Systems (*DSS*) require processing huge quantities of data to produce exact answers to posed queries. Due to the sheer size of the processed data, queries in *DSS* systems are typically slow. However, many situations arise when an exact answer to a query is not necessary, and the user would be more satisfied by getting a fast and fairly accurate answer to his query, perhaps with some error guarantees, than by waiting for a long time to receive an answer accurate up to the last digit. This is often the case in OLAP applications, where the user first poses general queries while searching for interesting trends on the dataset, and then drills down to areas of interest.

Approximate query processing techniques have been proposed recently as methods for providing fast and fairly accurate answers to complex queries over large quantities of data. The most popular approximate processing techniques include histograms, random sampling and wavelets. In recent years there has been a flurry of research on the application of these techniques to such areas as

selectivity estimation and approximate query processing. The work in [4, 10, 18] demonstrated that wavelets can achieve increased accuracy to queries over histograms and random sampling.

However, one of the shortcomings of wavelets is that they cannot easily extend to datasets containing multiple measures. Such datasets are very common in many database applications. For example, a sales database could contain information on the number of items sold, the revenues and profits for each product, and costs associated with each product, such as the production, shipping and storage costs. Moreover, in order to be able to answer types of queries other than Range-Sum queries, it would be necessary to also store some auxiliary measures. For example, if it is desirable to answer Average queries, then the count for each combination of dimension values also needs to be stored.

Surprisingly, there has not been much work in adapting wavelets to deal with such datasets. Two algorithms, which we will term as *Individual* and *Combined*, have been suggested for multi-measure datasets. The *Individual* algorithm performs an individual wavelet decomposition on each of the measures, and stores the coefficients for each measure separately. On the other hand, the *Combined* algorithm performs a joint wavelet decomposition by treating all the dataset measures as a vector of values, and at the end determines which vectors of coefficient values to retain.

As we will show in this paper, this *individual* or *combined* storage method can lead to suboptimal solutions even in very simple cases. Due to the nature of the wavelet decomposition, there are many cases in which multiple -but not necessarily all- coefficient values with the same coordinates have large values, and are thus beneficial to retain. The *Individual* algorithm will in such cases duplicate the storage of these shared coordinates multiple times, thus not optimizing storage utilization. On the other hand, the *Combined* algorithm stores *all* coefficient values that share the same coefficient coordinates, thus wasting space by also storing some very small coefficient values.

In this work we propose a novel approach on extending wavelets to deal with multiple measures through the use of *extended* wavelet coefficients. Each *extended* wavelet coefficient stores multiple *coefficient values* for different -but not necessarily for all- measures. This flexible storage technique can eliminate the disadvantages of the *Individual* and *Combined* algorithms discussed above. We then provide a dynamic programming algorithm for selecting the optimal *extended* wavelet coefficients to retain such that the weighted sum of squared L^2 error norms for all the measures is minimized. This is a natural extension to the commonly used single-measure optimization problem. Given the high complexity and mainly the memory requirements of our dynamic programming algorithm, we also propose a greedy algorithm that provides near-optimal solutions. The greedy algorithm's decisions on which *extended* wavelet coefficients to retain are based on a defined *per space benefit* metric. While both of these algorithms can be directly applied to the dataset at hand, in some cases it might be desirable to first normalize the values of the dataset's measures before their application. We also demonstrate that both of the presented algorithms can be easily modified to provide some minimal guarantees on the quality of the L^2 error norm for each measure. To the best of our knowledge this is the first work that utilizes a flexible storage method for multi-measure wavelets. The key contributions of our work are as follows:

1. We provide a qualitative and quantitative demonstration of the sub-optimal choices made by the existing algorithms for the wavelet decomposition of multi-measure datasets.
2. We provide a formal definition of an *extended* wavelet coefficient, the *first* wavelet-based compression method that utilizes a flexible storage approach to store the wavelet coefficients.
3. We provide an optimal dynamic programming algorithm that selects which *extended* wavelet coefficients to retain, in order to minimize the weighted sum of the squared L^2 error norms of all measures, under a given storage constraint.
4. We present an alternative greedy algorithm with reduced memory requirements and running time that produces near-optimal results to the above optimization problem. We also prove that the solution of the greedy algorithm is within an approximation ratio bound of 2 from the optimal solution.
5. We demonstrate that both the dynamic programming and the greedy algorithm can be easily modified to provide some minimal guarantees on the L^2 error norm of individual measures.
6. We present extensive experimental results with both synthetic and real datasets to demonstrate how the performance of the our proposed algorithms is influenced by multiple parameters. We first show that our greedy algorithm performs closely in accuracy to the optimal dynamic programming algorithm. We then demonstrate that our technique significantly outperforms random sampling and both of the existing multi-measure wavelet decomposition algorithms in terms of accuracy. To our knowledge, this is the first work that compares the performance of wavelets and random sampling in datasets with multiple measures.

This paper proceeds as follows: Section 2 presents the related work on approximate processing techniques. Section 3 describes the wavelet decomposition, along with the two existing techniques for wavelet decomposition of datasets with multiple measures and presents their advantages and shortcomings. In Section 4 we formally define the notion of an *extended* wavelet coefficient and our optimization problem, and then present an optimal dynamic programming algorithm for it. Section 5 presents a near-optimal greedy algorithm for the optimization problem with provable approximation ratio, while Section 6 presents some interesting modifications to both algorithms to provide some guarantees on the L^2 error norm of each measure and to further improve storage utilization. In Section 7 we validate the performance of our proposed method with an extensive experimental study, and Section 8 summarizes our conclusions.

2 Related Work

The main data reduction mechanisms studied so far include histograms, random sampling and wavelets.

Histograms is the most extensively studied data reduction technique with wide use in query optimizers to estimate the selectivity of queries, and recently in tools for providing fast approximate answers to queries [8, 13]. A classification of the types of histograms proposed in literature is

presented in [14]. The main challenge for histograms is to be able to capture the correlation among different attributes in high-dimensional datasets. Recent work in [16] addresses the problem of constructing accurate high-dimensional histograms with the use of sketching techniques, which were first introduced in [2]. The construction of accurate high-dimensional histograms is also addressed in [5], where statistical interaction models are employed to identify and exploit dependency patterns in the data.

Random Sampling is based on the idea that a small random sample of the data often represents well the entire dataset. The result to an aggregate query is given by appropriately scaling the result obtained by using the random sample. Random sampling possesses several desirable characteristics as a reduction method: the estimator for answering count, sum and average aggregate queries is unbiased, confidence intervals for the answer can be given, construction and update operations are easy and have low overhead, and the method naturally extends to multiple dimensions and measures. In [17] the reservoir sampling algorithm was introduced, which can be used to create and maintain a random sample of a fixed size with very low overhead. This method was further refined in [7] for datasets containing duplicate tuples.

Wavelets are a mathematical tool for the hierarchical decomposition of functions, with extensive use in image and signal processing applications [9, 12, 15]. [15] describes how the two existing wavelet decomposition algorithms for multiple measures, to which we refer to as the *Individual* and *Combined* algorithms, can be used for compression of colored images.

More recently, wavelets have been applied successfully in answering range-sum aggregate queries over data cubes [18, 19], in selectivity estimation [10] and in approximate query processing [4]. In the aforementioned work, wavelets were shown to produce results with increased accuracy to queries when compared to histograms and random sampling. The work in [4, 10, 18] clearly demonstrated that wavelets can be accurate even in high-dimensional datasets, while at the same time exhibiting small construction times. The problem of dynamically maintaining a wavelet-based synopsis was addressed in [11].

A method for providing error guarantees on the relative error of the wavelet decomposition was described in [6]. In this work, the relative error of the approximation is viewed as a more important metric than the L^2 error norm, and the criterion on which wavelet coefficients to retain differs from previous approaches. Extending this work to datasets with multiple measures would be very interesting.

3 Basics

We begin by giving an introduction to wavelets. We then describe the *Individual* and the *Combined* wavelet decomposition algorithms and present some of their advantages and shortcomings.

3.1 Wavelets

Wavelets are a mathematical tool for the hierarchical decomposition of functions in a space-efficient manner. The wavelet decomposition represents a function in terms of a coarse overall shape and details that range from coarse to fine [15]. In this work we will focus on the multi-dimensional Haar [15] wavelets, which are the easiest wavelet type conceptually. To illustrate how Haar wavelets work, we present a simple example of the one-dimensional dataset: [2 8 3 3]. The wavelet decomposition first performs a pair-wise averaging to get a lower-resolution signal of two values: [5 3]. During this down-sampling process, some information is obviously lost. We can reconstruct the original data values by also storing some *detail coefficients*, which in Haar wavelets are the pair-wise differences of the data values divided by 2. In our example, the values of the first *detail coefficient* will be $(2-8)/2=-3$, and similarly the value of the second *detail coefficient* is $(3-3)/2=0$. At this point no information has been lost: the original data values can be calculated precisely given the two average values and the detail coefficients. This pair-wise averaging and differencing process is then repeated recursively at the lower-resolution data array containing the averages of the data elements. The entire procedure is illustrated below:

Resolution Level	Average	Detail Coefficients
0	[4]	[1]
1	[5 3]	[-3 0]
2	[2 8 3 3]	

The detail coefficients produced by the above procedure, along with the overall average value, constitute the wavelet coefficients of the signal. Each coefficient is associated with a *coordinate value*, which helps determine the resolution level of the coefficient, along with its position within this level. For example, the coefficient value 0 of the above table corresponds to position 1 of resolution level 1. One desirable characteristic of the decomposition is that often several of the coefficients are either zero (for example, the coefficient [-3 0]), or have small values, and their omission only introduces small errors in the results.

Not all coefficients have the same importance for reconstructing the original data values. In the above example, the overall average value 4 is used to reconstruct the value of any data point, while the detailed coefficient -3 is used only for the first two data points. To account for this, in the wavelet decomposition the wavelets are normalized by dividing their value by a *normalization factor* which is equal to $\sqrt{2^l}$, where l is the resolution level of the coefficient. Retaining the coefficients with the largest normalized values has been shown to be optimal [15] for minimizing the L^2 error norm of a signal. The L^2 error norm is defined as: $L^2 = \sqrt{\sum_{i=1}^{N} e_i^2}$, where e_i denotes the approximation error of the signal's i-th data value. When the desired metric involves minimizing the relative error, selecting the optimal coefficients to store is more difficult, but recently [6] provided an optimal algorithm for this problem.

Finally, each coefficient in the above example can be represented as a tuple $< C_i, NV_i >$, where C_i is the coordinate of the i-th coefficient, and NV_i is its normalized value. For multi-dimensional

Coordinate	Value	Normalized Value	Stored Tuple
0	4	4	[0 4]
1	1	1	[1 1]
2	-3	$-3/\sqrt{2}$	[2 $-3/\sqrt{2}$]
3	0	0	[3 0]

Table 1: Wavelet Coefficients for Sample Dataset

Resolution	Average	Detail Coefficients
0	$[\begin{bmatrix} 4 \\ 4.5 \end{bmatrix}]$	$[\begin{bmatrix} 1 \\ 0.5 \end{bmatrix}]$
1	$[\begin{bmatrix} 5 \\ 5 \end{bmatrix} \begin{bmatrix} 3 \\ 4 \end{bmatrix}]$	$[\begin{bmatrix} -3 \\ -1 \end{bmatrix} \begin{bmatrix} 0 \\ -1 \end{bmatrix}]$
2	$[\begin{bmatrix} 2 \\ 4 \end{bmatrix} \begin{bmatrix} 8 \\ 6 \end{bmatrix} \begin{bmatrix} 3 \\ 3 \end{bmatrix} \begin{bmatrix} 3 \\ 5 \end{bmatrix}]$	

Table 2: An Example of the Combined Wavelet Decomposition Method

datasets, the corresponding representation of each coefficient is: $< C_{i1}, C_{i2}, ..., C_{iD}, NV_i >$, where C_{ij} is the coordinate of the i-th coefficient across the j-th dimension. Table 1 shows the tuples that constitute the wavelet coefficients of our original signal.

3.2 Existing Approaches for Multiple Measures

Two approaches [15] have been suggested for dealing with datasets containing multiple measures. In the first approach, an individual wavelet decomposition is performed for each measure, and the decision on which coefficients to retain is done independently for each measure. In the second approach, both the original data values and the produced coefficients are treated as $M \times 1$ vectors, where M is the number of the dataset's measures. The pair-wise averaging and differencing procedure described above is performed between vector elements belonging to the same row (thus corresponding to the same measure). The thresholding procedure used in this approach is not much different than the one where only one measure is included in the dataset: the retained vectors are the ones having the largest values of the L^2 norm. We will refer to these two different approaches as the *Individual* and the *Combined* decomposition algorithms, and use the terms *individual* and *combined* coefficient to refer to the storage methods of coefficient values that stem from these two algorithms. A *combined* coefficient thus stores *coefficient values* for all measures of the dataset. An example of the *Combined* decomposition algorithm is shown in Table 2. The dataset contains two measures: the first one is identical to the one of our first example, and the second measure has the values: [4 6 3 5]. Thus, the first row of each vector corresponds to either data or coefficient values of the first measure and, similarly, the second row of each vector corresponds to either data or coefficient values of the second measure.

Case A				
	Coordinate	Values		
Available Coefficients:	0	100	0	0
	1	0	100	0
	Coordinate	Values		
Combined Retains:	0	100	0	0
	Coordinate	Value	Measure	
Individual Retains:	0	100	1	
	1	100	2	
Combined Benefit $= 100^2 = 10000$				
Individual Benefit $= 100^2 + 100^2 = 20000$				
$\frac{\text{Combined Benefit}}{\text{Individual Benefit}} = \frac{10000}{20000} = 50\%$				

Case B				
	Coordinate	Values		
Available Coefficients:	0	100	100	100
	1	0	100	0
	Coordinate	Values		
Combined Retains:	0	100	100	100
	Coordinate	Value	Measure	
Individual Retains:	0	100	3	
	0	100	2	
Combined Benefit $= 100^2 + 100^2 + 100^2 = 30000$				
Individual Benefit $= 100^2 + 100^2 = 20000$				
$\frac{\text{Individual Benefit}}{\text{Combined Benefit}} = \frac{20000}{30000} \approx 66.7\%$				

Table 3: Sub-optimality of the Combined and Individual decomposition methods

The *Combined* decomposition algorithm is expected to achieve better storage utilization than the *Individual* algorithm in datasets where multiple coefficient values corresponding to the same coordinates contain large values.[1] In such cases, the coordinates of the coefficients are stored only once, thus allowing for a more compact representation. The more compact the representation, the larger the number of coefficients values that can be stored, and the better the accuracy of the produced results. On the other hand, in many cases a *combined* coefficient might help reduce significantly the error in only one, or few, measures. In such cases, some of the space that the *combined* coefficient occupies is wasted, without improving the overall quality of the results. Examples of the two cases are depicted in Table 3.

In Table 3, we present only two *combined* coefficients of an one-dimensional dataset with three measures. The actual data that helped construct these two coefficients, or the remaining set of coefficients is not important, since the sole purpose of this example is to show that both of the existing decomposition algorithms can make sub-optimal choices, even when choosing between just two coefficients. For this example we have assumed that each dimension coordinate and each coefficient value require one unit of space. Under this scenario, each input tuple or *combined* coefficient occupies four space units, while each *individual* coefficient occupies two space units. For a storage constraint of four units of space, the *Combined* algorithm can thus select only one tuple to store, while the *Individual* algorithm can store up to two *individual* coefficients. Due to the nature of the Haar transform, the benefit of retaining any single coefficient value is equal to the its squared value. As it can be seen from Table 3, in case A the *Combined* algorithm selects under this storage constraint a solution with only half the benefit of the one picked by the *Individual* algorithm. The roles are reversed in case B, as the *Individual* algorithm selects a solution with only two thirds of the benefit achieved by the *Combined* algorithm. In case B, the ties on the retained

[1]Each time we use the adjective ”large” for a coefficient value, we will be referring to its absolute normalized value.

coefficients for the *Individual* algorithm were broken arbitrarily, as four *individual* coefficients had the same benefit. By increasing in case A the number of measures in the dataset, and in case B the number of dimensions one can easily create examples where the difference in the quality of the optimal from the sub-optimal solution is significantly larger. For example, by expanding our one-dimensional dataset of Table 3 to a dataset with M measures and considering which of $\frac{M+1}{2}$ candidate coefficients to retain under a storage constraint of $M+1$ space units, it can be shown that in case A: $\frac{IndividualBenefit}{CombinedBenefit} = \frac{2}{M+1}$, while in case B: $\frac{CombinedBenefit}{IndividualBenefit} = \frac{M+1}{2 \times M}$.

Another disadvantage of the *Combined* decomposition algorithm is that in cannot be easily adapted for cases when one would like to weight differently the quality of the answers for different measures. In such a case, it would clearly not be advantageous to use the *Combined* algorithm, since it cannot devote different fractions of the available space to the measures, even though the coefficient values within each vector can be weighted differently. Moreover, for datasets with several measures, it would seem unlikely that *all* the coefficient values of each *combined* coefficient would be large. Thus, in such datasets it is expected that the *Combined* algorithm would waste storage space, without significantly improving the accuracy of the produced answers for all measures.

However, there are cases when one would expect that multiple coefficient values of a *combined* coefficient have large values. As we described in Section 3.1, the coefficient values are normalized, based on the resolution level they correspond to. Due to this normalization process, coefficient values at the top resolution levels, will tend to have larger values, and this will happen for all measures.

Another situation when multiple coefficient values might have large values at the same time arises in sparse datasets, as it is the case in many real life high-dimensional datasets. In such datasets, it is often the case that within a region of the data space, only one input tuple is present. As we have mentioned, the wavelet decomposition considers the data values as a signal and tries to approximate it. The presence of large spikes in the signal, such as the ones that can arise by the data values of a sole tuple in a sparse area, will possibly generate large coefficient values for many measures.[2]

4 Problem Definition and Optimal Solution

4.1 Extended Wavelet Coefficients

Given the shortcomings of the existing wavelet decomposition algorithms in dealing with multiple measures, we now introduce the notion of an *extended* wavelet coefficient.

Definition: An extended wavelet coefficient of a D-dimensional dataset with M measures is a triplet $< Bit, C, V >$ *consisting of:*

- *A bitmap Bit of size M, where the i-th bit denotes the existence, or not, of a coefficient value for the i-th measure*

[2]Depending on the size of the region this tuple belongs to and the data values of the tuple.

- *The coordinates C of the coefficient*
- *The stored coefficient values V*

The bitmap of an *extended* wavelet coefficient can be used to determine how many and which coefficient values have actually been stored. An *extended* wavelet coefficient can be used to combine the benefits of the *Combined* and the *Individual* decomposition algorithms, as it provides a flexible storage method that can be used to store from one to M coefficient values corresponding to each combination of wavelet coordinates. This flexible storage method bridges the gap between the two extreme hypotheses that these two algorithms represent, namely that only one or all coefficient values sharing the same coordinate values are important. Since our concern is selecting which coefficient values to store, our algorithms will only influence the final thresholding step of the wavelet decomposition process. The decomposition step of either the *Individual* or the *Combined* algorithm can be used to create the input to our algorithms.

The selection of which *extended* wavelet coefficients to store is based on the optimization problem we are trying to solve. Since in the case of datasets with one measure, the most common objective involves minimizing the L^2 error norm of the approximation, a natural extension, and interesting problem, for datasets with multiple measures is how to minimize the weighted sum of the squared L^2 error norms for all measures. More formally, the optimization problem can be posed as follows:

Problem Definition: Given a set T of candidate combined wavelet coefficients of a D-dimensional dataset with M measures, a storage constraint B, and a set of weights W, select the extended wavelet coefficients to retain in order to minimize the weighted sum $\sum_{i=1}^{M} W_i * (L_i^2)^2$ of the squared L^2 error norms for all the measures.

For the single-measure wavelet decomposition, it can be shown ([15]) that the square of the L^2 error norm is equal to the sum of squared normalized values of the coefficients that have not been stored. Utilizing this result, and using the symbol NV_{ij} to denote the normalized coefficient value for the j-th measure of the i-th candidate combined wavelet coefficient, one can easily transform the above optimization problem to an equivalent, and also more easy to process form:

Problem Definition 2: Given a set T of candidate combined wavelet coefficients of a D-dimensional dataset with M measures, a storage constraint B, and a set of weights W, select the extended wavelet coefficients to retain in order to maximize the weighted sum $\sum_{i=1}^{|T|} W_i * \sum_{j=1}^{M} NV_{ij}^2$ of the squared retained normalized coefficient values for all the measures

Note that any set of *individual* coefficients sharing the same coordinates can be transformed into a *combined* coefficient by setting the coefficient values of the non-stored measures to zero. Thus, the requirement that the input to our algorithm be a set of *combined* wavelet coefficients does not pose any restrictions on the decomposition method being used.

Symbol	Description
D	Number of dataset's dimensions
M	Number of dataset's measures
W	Measure weights
InCoeffs	Set of input combined coefficients
N = |InCoeffs|	Number of input coefficients
Items = $N \times M$	Number of candidate subitems
MS	Storage space for a coefficient value
H	Storage space for the coordinates, along with the bitmap
S = $H + MS$	Storage space for the first coefficient value of an extended wavelet coefficient
B	The storage constraint
Opt[K,SP].ben	The optimal benefit acquired when using at most the first K subitems and at most SP units of space
Force[K,SP].ben	The optimal benefit acquired when using at most the first K subitems and at most SP units of space, but forcing at least one subitem of the last candidate combined coefficient to be included in the solution

Table 4: Notation used in our algorithms and discussion

4.2 The Optimal DynL2 Algorithm

We now provide a dynamic programming algorithm that optimally solves the optimization problem of Section 4.1. The *DynL2* algorithm takes as input a set of *combined* coefficients, a space threshold, and a set of weights which will be used to weight the benefit of the coefficient values for each measure. The coefficient values of each *combined* coefficient are then treated as *subitems* in our problem. An implicit mapping being used maps the j-th coefficient value of the i-th *combined* coefficient to the value $(i-1)*M+j$, where M is the number of dataset's measures. Thus, for each *combined* coefficient, its coefficient values have consecutive mappings. We will use this fact later in the algorithm to simplify our problem. Table 4 summarizes some of the notation that we will use in our discussion.

By including any subitem Q corresponding to measure j and having a normalized value of V_Q to the optimal solution, we get a weighted benefit equal to $W_j \times V_Q^2$. However, the space overhead for storing it will depend on whether this is the first coefficient value being stored from the *extended* wavelet coefficient it belongs to, or not. In the former case, the space needed to store this coefficient value will be equal to the size needed to store the coordinates of the coefficient, along with the bitmap, and the coefficient value. In the latter case, since the bitmap and the coordinates of the coefficient have already been stored, the required space is just equal to the space that the coefficient value takes. This difference in the storage cost of a coefficient value makes the problem harder to

solve.[3]

We now try to formulate a recursion to solve our optimization problem. One requirement of the algorithm is that all subitems occupy an integer number of space units. This can be easily be done if we consider the space unit to be equal to 1 bit and express all space quantities in this unit.[4] For the optimal solution using space at most SP, and considering the first Q subitems, three cases may arise:

1. The optimal solution is the same as using $Q-1$ subitems and the same space SP
2. The optimal solution is achieved by including subitem Q, and Q is the first subitem of its *combined* coefficient included in the optimal solution
3. The optimal solution is achieved by including subitem Q, and Q is not the first subitem of its *combined* coefficient included in the optimal solution

An important thing to note is that in the third case, subitem Q is combined with the optimal solution *SubOpt* using at most the first $Q-1$ subitems, and space at most equal to $SP - MS$ and which includes at least one more subitem of the *combined* coefficient Q belongs to. The last requirement results in a perhaps surprising observation: the *SubOpt* solution is not always an optimal solution for our optimization problem when using only the first $Q-1$ subitems, and space up to $SP - MS$ units. An example is presented in Table 5. This Table includes just two coefficients, corresponding to a three-dimensional dataset with three measures. To keep things simple, let's assume that the storage bound B is equal to the size of one tuple, augmented by a bitmap of three bits, and that each measure has a weight of 1. Under this storage bound, two coefficient values from different coefficients cannot be stored. The unpredictable behavior described above is observed when considering the optimal solution containing at most the first five subitems and space bound $S + MS$. It is easy to see that the optimal solution at this point would be to store subitems 4 and 5, both corresponding to the second *combined* coefficient. It is also easy to observe however, that subitem 4 is not part of *any* optimal solution involving only the first four subitems and using *any* storage space less than B, since subitem 1 can be used at its place and get a solution with a larger benefit.

An encouraging observation is that we only need to store for each cell $[Q, SP]$ just one suboptimal solution, the one which forces at least one subitem of the combined coefficient Q belongs to, to be included in the optimal solution. The *Force* array described in Table 4 will store these suboptimal solutions.

The algorithm is presented in Algorithm 1, and includes the notation described in Table 4. Our dynamic programming algorithm utilizes two arrays, named *Opt* and *Force*, both with sizes $\{1..Items\} \times \{0..B\}$. Each cell in these two arrays contains two fields. The first field is described

[3] Otherwise the solution would have been a simple modification of the dynamic programming algorithm for the knapsack problem.

[4] Some techniques, like encoding the bitmap within some coordinate of the coefficient for small number of measures, can help increase the size of the space unit and decrease the memory requirements, but without reducing the space complexity of the problem.

Candidate Coefficients					
Coordinates			Values		
0	0	1	100	1	2
1	2	0	99	98	97

Considered SubItems	SubItems in Optimal Solution For Space Bound		
	S	S+MS	S+2MS
First 1	1	1	1
First 2	1	1,2	1,2
First 3	1	1,3	1,2,3
First 4	1	1,3	1,2,3
First 5	1	**4,5**	4,5
First 6	1	4,5	4,5,6

Table 5: Unexpected Optimal Solution Arises for Space Bound S+MS

in Table 4. The second field is used to code the choice made to decide the benefit of the cell. We will clarify this point in a while.

The algorithm begins by initializing some entries of both arrays. Line 1 depicts the fact that no coefficient value can be stored in space less than S. The optimal solution for space at least equal to S and while using only the first subitem will obviously only include this subitem (Line 2). The algorithm then iteratively fills the values of the remaining cells (Lines 3-12). For the Opt array, the optimal solution using space at most SP, and considering the first Q subitems, can be generated by one of the three cases described above (Line 7). For the corresponding optimal solution for the Force array (Line 9), the choices are similar. Note that some of the three cases are valid only if the current subitem satisfies some conditions, namely that it does not correspond to a coefficient value of the first measure ($y > 1$). The second field of the cells, which contains the choice made for assigning the benefit of the cell, is assigned the values 2, 3, or 4 correspondingly, depending on which of the three described cases produced the optimal solution. Cells that correspond to cases when no subitem can be stored in the specified space have the value of the second field set to 1 (Line 1).

At the end, the optimal benefit is the benefit achieved when considering all the subitems and using space at most equal to the given space constraint. The optimal solution can be reconstructed, starting from that cell, and moving depending on the second field of the current cell. If at some point we are at cell [i,j], the action that we will perform will depend on the value of the second field of the cell:

1. End of traversal
2. Move to cell [i-1,j] of the same array.
3. Move to cell [i-1, j - S] of the Opt array. This movement will always result in the addition of an *extended* wavelet coefficient (the one that includes the current subitem) to the solution.
4. Move to cell [i-1, j - MS] of the array Force. Add the current subitem to the corresponding *extended* coefficient (create one if needed).

Algorithm 1 DynL2 Dynamic Programming Algorithm

Input: InCoeffs, B, W

1: Set entries of both arrays corresponding to space less than S to have a benefit of 0 and a choice of 1.

2: Initialize first row of both arrays for space at least S. Set benefit to: $W[1] \times Value^2[1,1]$, and choice = 3.

3: **for** i in 2..$Items$ **do**

4: let x = 1 + (i-1) div M {Combined coefficient index}

5: let y = 1 + (i-1) mod M {Measure index}

6: **for** j in $S..B$ **do**

7: $$\text{Opt[i,j].ben} = \max \begin{cases} \text{Opt[i-1,j].ben} & \\ \text{Opt[i-1,j-S].ben+} \\ \text{W[y]} \times Value^2[x,y] & \\ \text{Force[i-1, j-MS].ben+} & \text{y} > 1 \\ \text{W[y]} \times Value^2[x,y] & \end{cases}$$

8: Depending on which of the three choices listed above produced the maximum value, set Opt[i,j].choice to the value 2, 3 or 4, respectively.

9: $$\text{Force[i,j].ben} = \max \begin{cases} \text{Force[i-1,j].ben} & \text{y} > 1 \\ \text{Opt[i-1,j-S].ben+} \\ \text{W[y]} \times Value^2[x,y] & \\ \text{Force[i-1, j-MS].ben+} & \text{y} > 1 \\ \text{W[y]} \times Value^2[x,y] & \end{cases}$$

10: Depending on which of the three choices listed above produced the maximum value, set Force[i,j].choice to the value 2, 3 or 4, respectively.

11: **end for**

12: **end for**

13: Reconstruct optimum solution by doing a reverse traversal starting from the entry $Opt[Items, B]$, and moving based on the choice field of the current entry.

14: Return as maximum benefit $Opt[Items, B].ben$

4.3 Space and Time Complexity

The space needed for the *DynL2* algorithm is essentially the size of *Opt* and *Force* arrays, which is O(NMB). Running the algorithm makes sense only when all the *combined* coefficients need more space to be stored than the space bound. This implies that $B < N \times (D + M)$, which means that the space requirements for the algorithm are bound by $O(N^2M(D + M))$. However, at each step of the algorithm, the part of the *Opt* and *Force* arrays that need to be memory resident is only $O(NM(D + M))$.

Given that the value of each cell can be calculated in O(1) time for both arrays, the time complexity of the algorithm is also O(NMB) and is bound by $O(N^2M(D + M))$. The reverse traversal procedure that is used to identify the coefficient values that are part of the optimal solution takes O(NM) time, since at each step of the procedure we are checking whether a subitem Q belongs in the result, and then proceed to subitem $Q - 1$.

5 Greedy Algorithm

5.1 Greedy Algorithm

We now present a greedy solution to the optimization problem of Section 4.1. Our algorithm, to which we will refer as *GreedyL2*, is based on transforming the optimization problem to match the 0-1 Knapsack Problem, and then selecting which coefficient values to store based on a *per space benefit* metric. The notation used in this section is consistent with the one described in Table 4.

Similar to the dynamic programming algorithm presented in the previous section, *GreedyL2* receives as input a set of candidate *combined* wavelet coefficients, a set of weights, and a storage constraint. Instead of considering the benefit of each coefficient value individually, *GreedyL2* considers at each step the optimal benefit achieved by selecting a set of K $(1 \leq K \leq M)$ coefficient values of the same *combined* coefficient that have not already been stored. It easy to see that the optimal selection will include the non-stored coefficient values that have one of the K largest benefits: $W_j \times V_{ij}^2$, where W_j is the weight corresponding to the coefficient value, and V_{ij} is its normalized value. The storage space for these K values will be equal to $H + K \times MS$, if no value of this *combined* coefficient has been stored before, and $K \times MS$ otherwise. *GreedyL2* maintains a structure with all the optimal sets of size K $(1 \leq K \leq M)$ of all the *combined* coefficients, and selects the set with the largest per space benefit. The coefficient values belonging to the selected set are stored, and the benefits of the optimal sets for the chosen *combined* coefficient have to be recalculated to only consider values that have not already been stored. The algorithm is presented in Algorithm 2.

For each input *combined* coefficient, the first step is to decide the sort order of its coefficient values based on their weighted benefit (Line 4). For each *combined* coefficient we also maintain the number of its coefficient values that have been selected for storage in the Stored variable which at the beginning of the algorithm is initialized to 0 (Line 5). Due to the way our algorithm is formulated, we do not need to remember which of the coefficient's values have been selected for storage, since these will always be the ones with the Stored maximum weighted benefits. We then calculate the optimal benefits of sets containing K coefficient values, $1 \leq K \leq M$ (Line 6). The maximum number of such sets is at most M, but not always equal to M, since we do not need to create any sets that include any coefficient values with zero benefit. The space needed to store each of these K sets is $H + K \times MS$. The per space benefit of each set, along with its occupied space and the identifier of the coefficient it belongs to, are then inserted in an AVL tree. We chose to use such a structure, since each of the insert, delete and finding the maximum value operations has logarithmic cost. However, any other data structure with similar characteristics can be used in its place.

The algorithm then repeatedly (Lines 9-15) picks the set with the maximum per space benefit that can fit in the remaining space. The values corresponding to this set are uniquely identified by the identifier of the corresponding *combined* coefficient *Coeff*, its Stored variable, and the size of the picked set. For the *Coeff* coefficient, the optimal benefits of its sets have to recalculated

Algorithm 2 GreedyL2 Algorithm

Input: InCoeffs, B, W

1: An AVL tree structure is used to maintain the optimal benefits of the candidate sets of coefficient values
2: For each *combined* coefficient, a variable *Stored* maintains the number of its coefficient values that have already been selected to be stored
3: **for** i in 1..N **do**
4: Determine sort order of coefficient values, based on their weighted benefit
5: For the current *combined* coefficient, Stored = 0
6: Calculate the optimal benefit of each set of size K ($1 \leq K \leq M$) and insert it into the AVL tree.
7: **end for**
8: SpaceLeft = B
9: **while** $SpaceLeft > 0$ AND candidate sets exist **do**
10: Select set $PickedSet$ of combined coefficient *Coeff* with maximum per space benefit and that needs space less than SpaceLeft
11: Adjust value of SpaceLeft, based on value of *Coeff*'s Stored variable and size of $PickedSet$
12: *Coeff*.Stored += number of values in $PickedSet$
13: Remove from AVL tree all sets belonging to *Coeff*
14: Calculate new benefits of *Coeff*'s sets of non-stored coefficient values and insert them in the AVL tree
15: **end while**
16: For each *combined* coefficient store the Stored coefficient values with the largest weighted benefit

to include only non-stored coefficient values. At most $M - Stored$ such sets can be created. The previous sets of *Coeff* are removed from the tree and the newly calculated ones are then inserted. Note that the space required for the newly inserted sets does not include the size of the header, since this has already been taken into account. The entire procedure terminates when no set can be stored without violating the storage constraint. In order to create the output *extended* coefficients, we simply have to parse the list of the *combined* coefficients, and for any coefficient C that has a Stored value greater than 0, create an *extended* wavelet coefficient and store in it the Stored coefficient values of C with the largest weighted benefits.

Theorem 1 *The GreedyL2 algorithm has an approximation ratio bound of 2*

Proof: The proof is similar to the corresponding proof for the 0-1 knapsack problem. A significant observation is that whenever we select a set *pickedSet* of coefficient values from a *combined* coefficient *Coeff* for storage, any candidate set *subOpt* of *Coeff* that will later be inserted in the AVL tree for consideration cannot have a larger per space benefit than the one of *pickedSet*. We will prove this by contradiction. Assume that *subOpt* has a larger per space benefit than *pickedSet*. By the way the candidate sets are formed, the following observations hold:

1. The sets *pickedSet* and *subOpt* can not share any coefficient values

2. The largest benefit of a coefficient value of *subOpt* cannot be larger than the smallest benefit

of a coefficient value of *pickedSet*

3. The space overhead of *subOpt* does not include the size of the header, while for *pickedSet* this depends on whether it is the first set of *Coeff* selected for storage.

If we depict the benefits of the coefficient values of *pickedSet* as $v_1, v_2, \ldots, v_k$ and the benefits of the coefficient values of *subOpt* as $v'_1, v'_2, \ldots, v'_p$, then the per space benefits of the two sets are, correspondingly, $\frac{\sum_{i=1}^{k} v_i}{\delta \times H + k \times MS}$ and $\frac{\sum_{i=1}^{p} v'_i}{p \times MS}$, where δ has a value of 1 or 0, depending on whether *pickedSet* is the first set of *Coeff* selected for storage. Since by hypothesis *subOpt* has a larger per space benefit than *pickedSet*:

$$\frac{\sum_{i=1}^{p} v'_i}{p \times MS} > \frac{\sum_{i=1}^{k} v_i}{\delta \times H + k \times MS} \Longrightarrow \sum_{i=1}^{p} v'_i \times (\delta \times H + k \times MS) > \sum_{i=1}^{k} v_i \times p \times MS \qquad (1)$$

At the time *pickedSet* was selected, a candidate set *notPicked* of *Coeff* with $k + p$ subitems existed, having a benefit which is at least equal to the set *union* containing all subitems in *pickedSet* and *subOpt*. Comparing the per space benefit of *notPicked* and *pickedSet*, we have:

$$\begin{aligned} &benefit(notPicked) - benefit(pickedSet) \geq benefit(union) - benefit(pickedSet) = \\ &\frac{\sum_{i=1}^{k} v_i + \sum_{i=1}^{p} v'_i}{\delta \times H + (k+p) \times MS} - \frac{\sum_{i=1}^{k} v_i}{\delta \times H + k \times MS} = \frac{\sum_{i=1}^{p} v'_i \times (\delta \times H + k \times MS) - \sum_{i=1}^{k} v_i \times p \times MS}{(\delta \times H + (k+p) \times MS) \times \delta \times H + k \times MS} > 0 \end{aligned} \qquad (2)$$

The last part of formula (2) follows immediately from the inequality of formula (1). At this point we have reached a contradiction, since *pickedSet* should not have a smaller per space benefit than the set *notPicked*. Therefore, *subOpt* cannot have a larger per space benefit than *pickedSet*.

The above observation also implies that each candidate set inserted in the AVL-tree after the first set selection made by the algorithm cannot have a larger per space benefit than the ones that have already been selected. To prove this, consider any such set *nowInserted* that is inserted in the AVL tree following the selection for storage of another set *nowStored*, of the same candidate combined coefficient *Coeff*, and consider the following two observations:

1. Following the preceeding proof, the set *nowInserted* cannot have a larger per space benefit than any set already picked for storage from the same candidate combined coefficient *Coeff*.

2. Consider any candidate set *otherSet* already selected for storage, which corresponds to a candidate combined coefficient other than *Coeff*. At the moment *otherSet* was selected for storage, the candidate set of *Coeff* with the largest per space benefit that was at that time in the AVL tree could not have a larger per space benefit than *otherSet*, since it would have been selected for storage instead of it, and cannot have a smaller per space benefit than *nowStored*.

Now, consider that *GreedyL2* solution has selected to store the sets $S_1, S_2, ..., S_l$, and that S_{l+1} is the set with the largest per space benefit that cannot be stored due to space constraints.[5] Let

[5]For simplicity, consider that S_{l+1} could fit by itself within the original storage constraint.

$BenStored$ denote the sum of benefits of the l sets included in the solution, $BenFraction$ denote the benefit of set S_{l+1}, and $BenOptimal$ denote the benefit of the optimal solution. If the space needed for S_{l+1} was B_{l+1}, and the remaining storage space at that point of our algorithm was $SpaceLeft$, it can easily be shown [6] that the optimal solution has at most benefit equal to: $BenStored + BenFraction \times \frac{SpaceLeft}{B_{l+1}} \leq BenStored + BenFraction$. At this point, the solution Z = max$\{BenFraction, BenStored\}$ has at least half the benefit of the optimal solution, since:

$$BenOptimal \leq BenFraction + BenStored \leq 2 \times max\{BenFraction, BenStored\}$$

5.2 Space and Time Complexity

Each of the N input *combined* coefficients creates at most M candidate sets. Therefore, the space for the AVL tree is O(NM). For each combined coefficient, maintaining the sort order requires O(M) space. The size of the input *combined* coefficients is O(N(D+M)), making the overall space complexity of the algorithm O(N(D+M)).

Determining the sort order for the values of each *combined* coefficient requires time O(MlogM). Calculating the benefits of the sets produced by each coefficient then takes only O(M) time. The original construction of the AVL-tree can be done in O(NMlog(NM)) time. Each time a set is picked for inclusion in the result, the search requires O(log(NM)) time. Then, we need to make O(M) deletions from the AVL tree, corresponding to all the sets of the chosen *combined* coefficient. Finding all such nodes on the tree requires O(M) time, if they are connected by a cyclic list. Note that all the sets of the same *combined* coefficient are created at the same time, thus making it easy to create such a list. Each of the O(M) insertion and deletion operation then requires O(log(NM)) time. Since at most O(NM) sets can be picked, the total time complexity is O($NM^2 log(NM)$).

6 Related Issues

6.1 Providing Fairness and Error Guarantees

While the optimization problem of Section 4.1 might be a desirable objective in many problems, certain cases may arise when both the greedy and the dynamic programming algorithms presented will significantly favor certain measures at the expense of other measures. This usually happens when two or more measures with significantly different magnitude of coefficient values occur within the same dataset. In such cases, both algorithms will almost exclusively store coefficient values corresponding to the measure with the largest coefficient values. This might not be desirable in certain applications, since it would introduce very large errors for some measures.

In such cases, a plausible solution to this problem would be to normalize the values of all measures such that all measures have the same *energy*. The *energy* of a measure is defined to be the sum of its squared values. Normalization can be achieved by either preprocessing the dataset,

[6] The proof is identical to the optimal proof for the fractional knapsack problem.

or by dividing the weighted benefit of each coefficient value by the energy of the corresponding measure.

Another solution involves adapting our proposed algorithms to provide certain guarantees on the quality of the produced solution. It turns out that both algorithms can be modified such that the resulting benefit for each measure be at least equal to the one produced by the *Individual* algorithm when the storage allocated to each measure is proportional to the measure's weight. This requires first computing the actual solution that the *Individual* algorithm would produce by using the aforementioned space allocation policy among the measures. Then, the solution produced by either the *GreedyL2* or the *DynL2* algorithm can be tested to verify whether for each measure the benefit criterion has been met. If this is not the case for all measures, the algorithm can proceed by splitting the measures into two sets: the ones where the benefit criteria has been satisfied, and those measures where it has not been satisfied. The algorithms can then be called recursively for each of these two sets, with the storage constraint allocated to each set being proportional to the weights of its measures. If at some point an input set contains just one measure, then the algorithm may store the coefficients in the same format as the *Individual* algorithm does. Thus, in the worst case, for any measure we can guarantee that the benefit we achieve for each measure is at least the same as the one produced by the *Individual* algorithm.

6.2 Improving Space Utilization

The space utilization of the *GreedyL2* and *DynL2* algorithms can be further improved at the expense of the query response time. For a dataset with M measures, we can split the produced coefficients into $M+2$ groups of coefficients. One group will be created for each measure and will include all the *extended* wavelet coefficients that have stored a coefficient value only for the corresponding measure. Another group will contain the *extended* coefficients that have stored coefficient values for all M measures, while the final group will include *extended* coefficients that have stored from 2 to $M-1$ coefficient values. From these $M+2$ groups, the bitmap is necessary only for the last group. In the other groups we can simply store the coefficients in the same way that the *Individual* and the *Combined* algorithms would, without the bitmap. The proposed algorithms then only require a slight modification when calculating the size needed to store a coefficient value (for the *DynL2* algorithm) or the size of a candidate set (for the *GreedyL2* algorithm). A query involving X measures would then have to probe $X+2$ groups of coefficients in search for coefficient values that influence the query result. This overhead in response time is in most cases negligible, given the small response times that queries exhibit when using wavelet synopses [4].

7 Experiments

We performed a series of experiments to validate the performance of the *GreedyL2* and the *DynL2* algorithms against the existing approaches. In our experiments we used both synthetic and real datasets. The experiments were performed on a personal computer using an Athlon 1800+ processor

with 512 MB of RAM memory. We compared the performance of the *GreedyL2* and the *DynL2* algorithms to the following four algorithms:

Random Sampling (RS): In all experiments we used the Reservoir algorithm described in [17], since the datasets that we used did not contain duplicate tuples.

Ind: The space allocated to each measure is proportional to its weight. Then the *Individual* algorithm is run for each measure.

IndSorted: Similar to Ind, but no limit is imposed to the size allocated to each measure. The individual coefficients from all measures are sorted according to their weighted benefit, and the ones with the highest benefits are retained.

Combined: The *combined* coefficients are sorted according to their overall weighted benefit, and the ones with the highest benefits are retained.

Histograms were not included in the performance evaluation, due to their inability to extend to datasets with multiple measures. Creating a separate histogram for each measure might be an alternative, but the work in [4, 18] leads us to expect that they would perform worse than the Ind algorithm.

In the experiments presented below we did not incorporate the storage utilization improvements discussed in Section 6.2. As input to our algorithms we used the output of the decomposition step of the *Combined* algorithm, which we found to produce better results than the corresponding output of the *Individual* algorithm.

7.1 Synthetic Datasets

For our synthetic datasets we created a data generator similar to the one used in [18, 4]. The input parameters of the generator along with their default values are described in Table 6. The generator begins by populating *n_regions* rectangular regions of a D-dimensional array, whose size is determined by the number of the dataset's dimensions and the cardinalities $Card_i$ of each dimension. The number of cells within each region is bound by the values V_{min} and V_{max}. The total sum Sum_i of values for each measure is partitioned across the *n_regions* rectangular regions through the use of a Zipf function with parameter Z. Then, within each region each measure's values are distributed by using one of the four distributions described in Table 7, with the parameter's values ranging from z_{min_i} to z_{max_i}. Notice the use of the *Altered-X*[7] distribution, which helps create pairs of measures with similar, but not identical, data distributions. The data generator then also populates a number of cells in the remaining D-dimensional space, outside the dense regions. The fraction of such cells over the total number of populated cells is defined by the *spCount* parameter, and the total sum of the values of these cells by the $spSum_i$ parameter.

We first investigate how close the weighted benefit achieved by the *GreedyL2* algorithm is to the one achieved by the *DynL2* algorithm. We created a synthetic dataset with 4 measures, following

[7] X can be either one of the Center, Middle or Reverse distributions.

Parameter	Description	Default Value
N	Number of dimensions	2
M	Number of measures	6
$Card_i$	Cardinality of dimension i	1024
n_regions	Number of dense regions	10
V_{min}, V_{max}	Minimum and maximum volume of regions	4900, 4900
Z	Skew across regions	0.5
z_{min_i}, z_{max_i}	Minimum and maximum skew within region i	1, 1
Sum_i	Sum of values for measure i	1,000,000
spCount	Fraction of populated cells in sparse areas	0.05
$spSum_i$	Sum of values of populated cells in sparse area i	0.05

Table 6: Data Generator Input Parameters

the Center, Altered-Center, Middle and Reverse distributions, and set the remaining parameters to the default values of Table 6. We modified the storage constraint from 1200 to 6000 bytes and present the results in Table 8. The deviation factor presented in this table is defined as: $1 - \frac{GreedyL2Benefit}{DynL2Benefit}$. The *GreedyL2* algorithm produced solutions with benefit very close to the optimal one. Due to the large amount of memory required by the *DynL2* algorithm, we were unable to execute it for the remaining experiments, and is thus omitted from the presented results.

We now evaluate the impact that several parameters have on the performance of our method. In each experiment, unless specified otherwise, the data generator parameters were set to their default values. For the default number of measures (6), the data distributions were: Center, Altered-Center, Reverse, Altered-Reverse, Middle and Altered-Middle. The query workload always consisted of 100 range queries, with the width of the range on each dimension being equal to 10. The queries targeted the dense areas with greater probability, since most of the data is stored there. The default storage bound was set to 5% of the dataset's size. For each performed query, the weighted absolute error is calculated from the formula:

$$(\sum_{i=1}^{M} W_i)^{-1} \times \sum_{i=1}^{M}(W_i \times |actualresult_i - estimatedresult_i|)$$

The variables $actualresult_i$ and $estimatedresult_i$ denote the exact and the estimated values of the result for measure i, correspondingly. The weighted sum squared and relative errors are defined similarly.

Storage Space: In Figures 1, 2, 3 we present the average weighted sum squared, absolute and relative errors, respectively, for all the algorithms as the storage space is varied from 2 to 10% of the dataset's size. The skew of the data distributions within each region was set to 1.5. Note that the y-axis of Figure 1 is logarithmic, due to the large errors exhibited by Random Sampling.

Distribution	Description
Center	Cells with smaller L1-distance from center have larger values
Reverse	Cells with smaller L1-distance from center have smaller values
Middle	Consider a hyper-rectangle centered at the region's center, and having for each dimension, half the length of the corresponding region length. Cells with smaller L1-distance from this hyper-rectangle have larger values
Altered-X	This measure follows the same distribution as X distribution, but its values are randomly altered by up to 50%

Table 7: Data Generator Value Distributions

	Storage Constraint (Bytes)				
	1200	2400	3600	4800	6000
Deviation Factor	3×10^{-5}	5×10^{-4}	10^{-6}	10^{-4}	10^{-6}

Table 8: Deviation Factor of GreedyL2 Benefit when Compared to the DynL2 Benefit

The *GreedyL2* algorithm produced results with considerably smaller errors than the ones of the other algorithm. In particular, the average weighted sum squared error of *GreedyL2* was in most cases about one third of the closest competitor's error, and as low as 28.5% (6105.25 vs 21399.2 for 8% space). For the average weighted absolute error case, the error of *GreedyL2* was typically about 67-70% of the one produced by the closest competitor (51.85 vs 91.19 for 9% space, a ratio of 64.5%). Finally, for the average weighted relative error, GreedyL2 produced results that were up to 39% less (31.13% vs 50.36% for 2% space) than the ones of the closest competitor. From the remaining methods, the *Combined* algorithm produced the best results.

We will present for the remaining experiments with the synthetic datasets only the results for the average weighted sum squared and absolute errors. We will also omit from the graphs the results for the Random Sampling algorithm and only summarize its errors, since they were consistently much larger than the ones of the other algorithms.

Skew within Regions: We modified the zipfian parameter controlling the skew of the measure's data distributions within each region from 0.5 to 4. Figures 4 and 5 present the obtained results for the weighted average sum squared and absolute errors. As the skew increases, for each distribution the coefficients with large values are limited to an increasingly smaller area. This results on one hand in the reduction of the sum squared error of the results, as the number of coefficient values that greatly influence it becomes smaller. On the other hand, the probability that coefficient values from multiple measures be important simultaneously is decreased. These two factors justify the relative improvement of the performance of IndSorted over the *Combined* algorithm for larger skew values. While the *Combined* algorithm performs closely to the *GreedyL2* algorithm for small skews, the

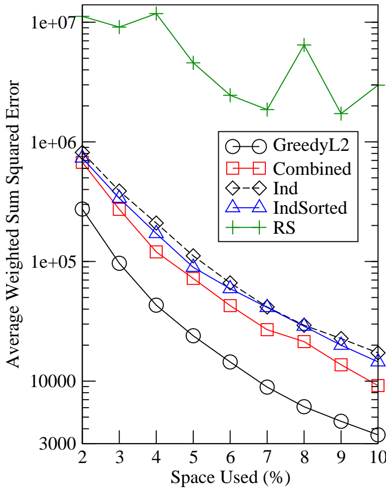

Figure 1: Average Weighted Sum Squared Error

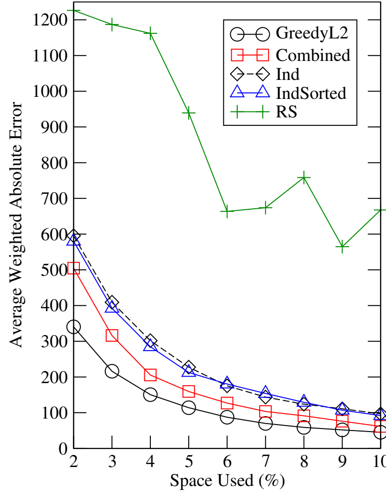

Figure 2: Average Weighted Absolute Error

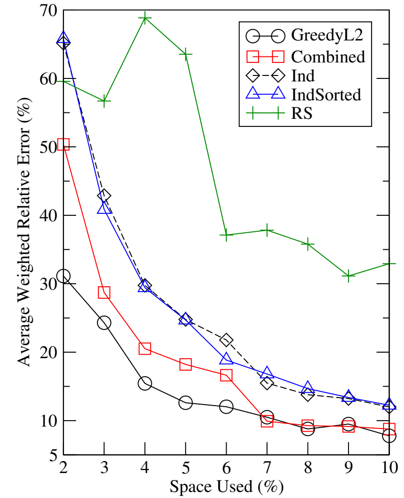

Figure 3: Average Weighted Relative Error

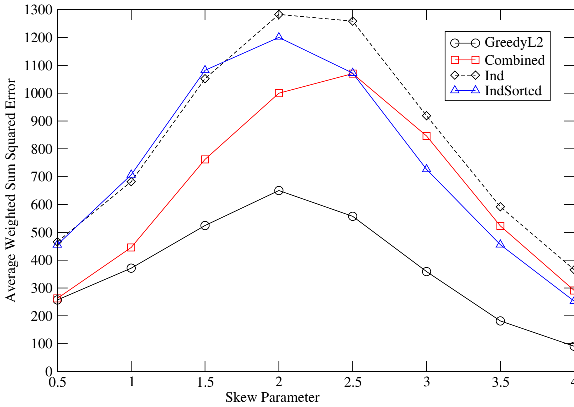

Figure 4: Sensitivity to Skew: Average Weighted Sum Squared Error

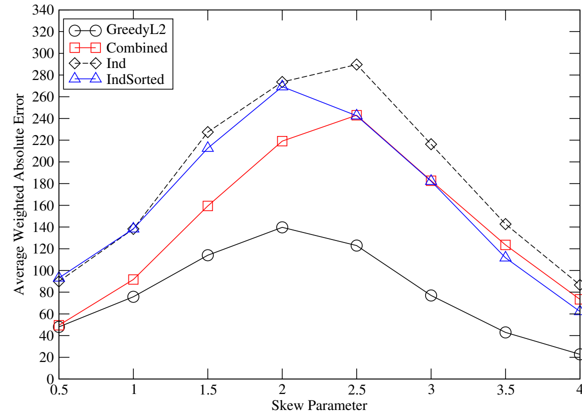

Figure 5: Sensitivity to Skew: Average Weighted Absolute Error

difference becomes very large as the skew increases. For large skews, *GreedyL2* exhibits about one third of the errors observed by the closest competitive algorithm (22.79 vs 62.21 absolute errors for skew parameter = 4). The random sampling technique produced results with 7 to 343 times larger errors than the ones of *GreedyL2*. As the skew parameter increases, the performance of Random Sampling deteriorated significantly, which is consistent with the findings in [4].

Variance in Weights: We varied the weight of the first measure from 0.5 to 4 to identify the impact on the accuracy of the produced result. Figures 6 and 7 present the results. *GreedyL2* exhibits errors that are consistently about two thirds of the closest competitor's errors (97.36 vs 147.15 absolute errors for weight = 3.5, a factor of 66.2%). The improvements are even larger for the case of sum squared error. It is interesting to note that the *GreedyL2* and the IndSorted methods exhibit the greatest improvement in accuracy when the weights are varied significantly, both reducing their errors by about 19%. In this experiment Random Sampling produced errors that were from 6 to 10 times larger than the ones of *GreedyL2*.

It is interesting to see for this experiment how well each measure is approximated by the different algorithms. Figure 8 presents the average error for each measure, for the case when the first measure is assigned a weight value of 4. To calculate the average weighted error for all measures, the error of Measure 1 ($M1$) needs to be multiplied by a factor of 4, and the resulting quantity to be divided by the value 9, which is the sum of the measures' weights. As Figure 8 shows, the Ind algorithm exhibits the smallest error for the measure with the largest weight, about half of the error that *GreedyL2* achieves, while the *Combined* algorithm performs the worst for this measure. However, *GreedyL2* achieves the lowest errors for the remaining five measures thus displaying that even though it can adjust its choices in cases of measures with large weights, it does so without without severely impacting the accuracy of the remaining measures. Another interesting observation is that the second measure, which follows a distribution similar to the heavily weighted Measure 1, benefited significantly in the *GreedyL2* algorithm, a behavior that was not observed in the other algorithms.

Number of Measures: In Figures 9, 10 we present the average weighted sum squared and absolute errors as the number of measures is varied from 2 to 6. The initial two measures are the ones with distributions Center and Middle, and the measures that are later added are: Reverse, Altered-Center, Altered-Reverse, Altered-Middle. As the number of measures increases, the improvement on accuracy of GreedyL2 over the competitive methods increases. Random Sampling produced errors that were from 6 to 13 times larger than the ones of *GreedyL2*.

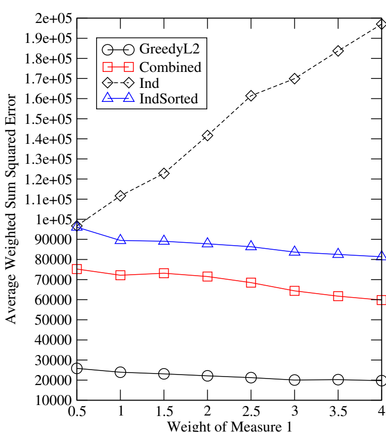

Figure 6: Sensitivity to Weights: Sum Squared Error

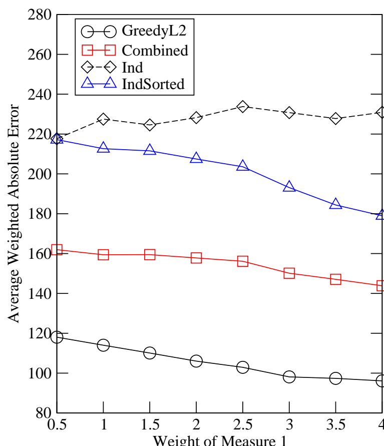

Figure 7: Sensitivity to Weights: Absolute Error

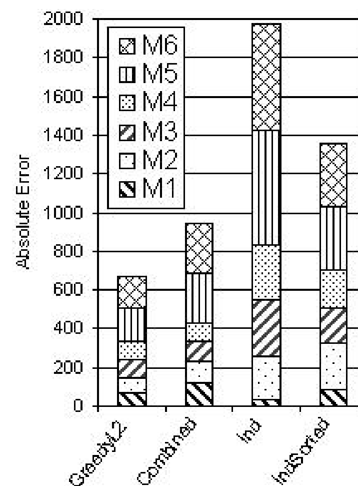

Figure 8: Errors for Different Measures

7.2 Real Dataset

For our real dataset we used weather measurements from the state of Washington [1]. The measures that we used were solar irradiance, wind speed, wind peak, air temperature, dewpoint temperature and relative humidity for the station in the university of Washington, and for the last one year. To simulate enhanced interest to specific measures, we assigned a weight value of 3 to the first

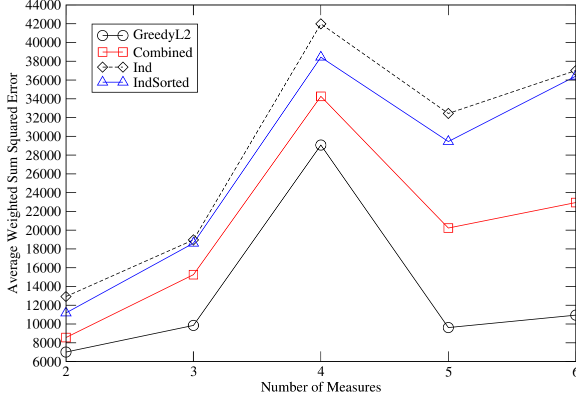

Figure 9: Sensitivity to Number of Measures: Average Weighted Sum Squared Error

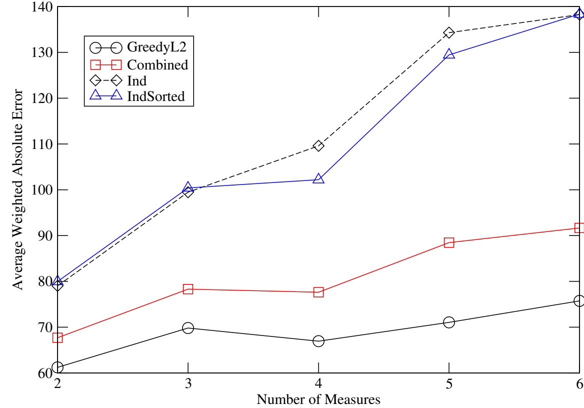

Figure 10: Sensitivity to Number of Measures: Average Weighted Absolute Error

measure, a weight value of 2 to the next two measures, and a weight value of 1 to the remaining measures. The measurements were taken on a per minute basis[8], and the dataset contained two dimensions, the day and the time of the measurement and a total of 521817 tuples. We performed 1000 range queries, where each range for the two dimensions was a random number with maximum value 30 and 180, respectively. Thus, the maximum selectivity of a query was about 1%. From the results of each query, we calculated average values for the stored measures over the queried day and time periods. The average values for each query were calculated by using the number of cells that each query accessed.

All measures were normalized, according to the following process: We first calculated for each measure i its average value $\overline{x}_i$ and its energy (sum if its squared values) $Energy_i$. We then modified each value v of measure i according to the formula (T is the number of input tuples):

$$v = 100 * \frac{(v - \overline{x}_i) \times \sqrt{T}}{\sqrt{Energy_i - T \times \overline{x}_i^2}}$$

With the above proceedure, each signal is modified to have an average value of 0, and an energy equal to $T \times 100^2$. We found out that removing the influence of the average value, which in this dataset carried a large part of the signals' energies, and then setting the energy of each signal to a common value produced better results.

In Figures 11, 12, 13 we present the results for the average weighted sum squared, absolute and relative errors, as the storage bound was varied from 1KB to 10KB. The errors for all wavelet methods decreases with the increase of the space bound. As it can be seen from the three figures, the *GreedyL2* algorithm consistently produced the smallest errors for all error metrics. The average weighted sum squared and absolute errors of Random Sampling were 2 and 1 orders of magnitude larger, respectively, than the corresponding errors of *GreedyL2*, while the average weighted relative errors of Random Sampling were about 3-4 times larger than the errors of *GreedyL2*. Even though the optimization problem of Section 4.1 is directly linked only to the average weighted sum squared

[8] Measurements corresponding to a small number of minutes were missing.

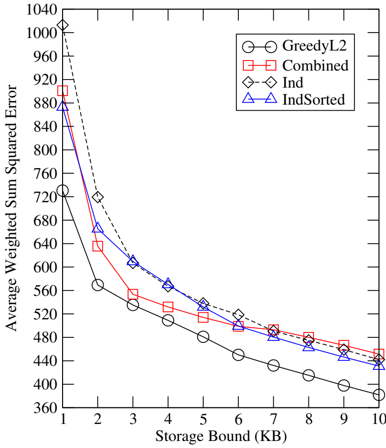

Figure 11: Average Weighted Sum Squared Error

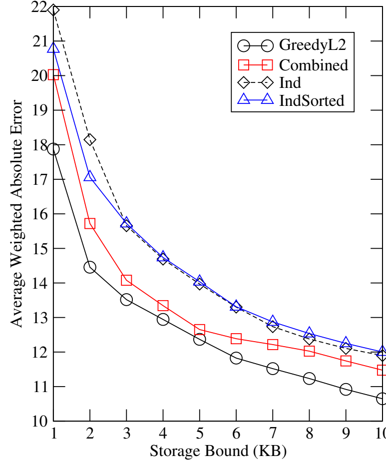

Figure 12: Average Weighted Absolute Error

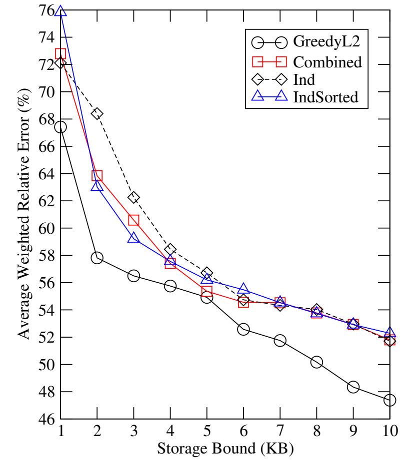

Figure 13: Average Weighted Relative Error

metric, the improved storage utilization of *GreedyL2* resulted in improvements in the accuracy of the average weighted absolute and relative error metrics as well.

Varying the Query Selectivity

In our experiments so far we have used a small query selectivity (1%). We now perform the same experiment as above, but try medium and large query selectivities. In Figures 14, 15, 16 we present the results when the performed queries had a maximum selectivity of 4.4% (maximum query rectangle of 62×372). The corresponding results for a maximum query selectivity of 26% (maximum query rectangle of 183×745) are presented in Figures 17, 18, 19. With the increase of the query selectivity, the errors decrease for all methods. However, *GreedyL2* still outperforms all the other techniques.

7.2.1 Non-Normalized Dataset

In this experiment we do not normalize the measures of the dataset. We now use a different assignment of weights, assigning a weight value of 3 to the two measures involving temperatures, a weight value of 1 to the two measures involving the wind, and a weight value of 2 to the remaining two measures. The query workload had a maximum query selectivity of 1%, as above. In Figures 20, 21 we present the results for the average weighted sum squared and absolute errors, as the storage bound was varied from 1KB to 10KB. The corresponding results for the average weighted relative error are presented in Table 9. Again we observe that the errors for all wavelet methods decreases with the increase of the space bound. The large relative errors for all methods for smaller storage bounds is mainly due to the solar irradiance measure, which was the hardest to approximate. As it can be seen from the two figures and the presented table, the *GreedyL2* algorithm consistently produced the smallest errors for all error metrics, with the exception of three cases for the average weighted relative error. The average weighted sum squared and absolute errors of Random

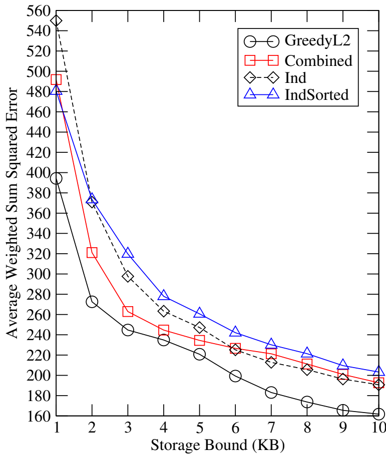

Figure 14: Selectivity 4.4%: Average Sum Squared Error

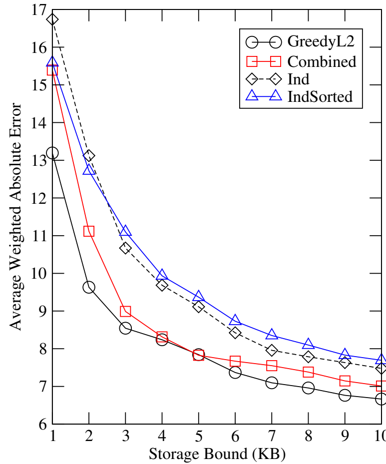

Figure 15: Selectivity 4.4%: Average Absolute Error

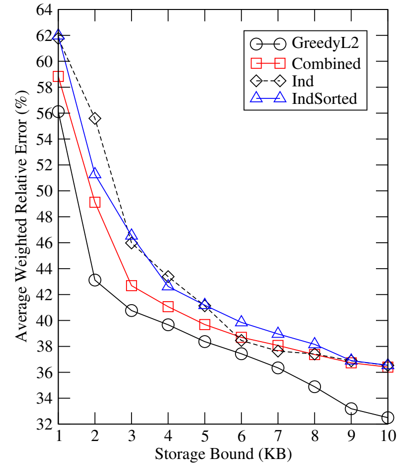

Figure 16: Selectivity 4.4%: Average Relative Error

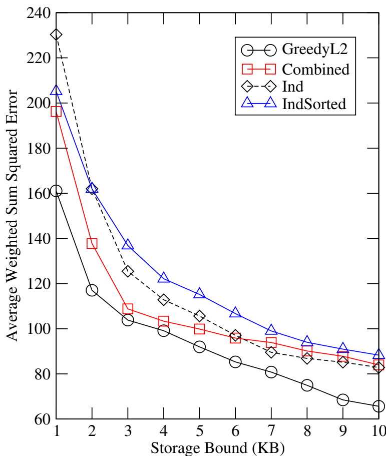

Figure 17: Selectivity 26%: Average Sum Squared Error

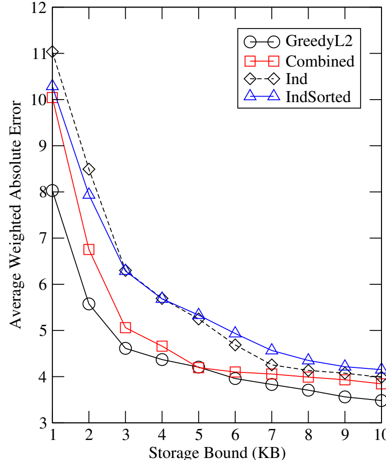

Figure 18: Selectivity 26%: Average Absolute Error

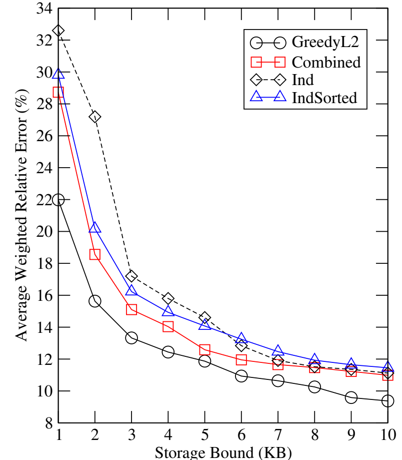

Figure 19: Selectivity 26%: Average Relative Error

Sampling were 2 and 1 orders of magnitude larger, respectively, than the corresponding errors of *GreedyL2*.

8 Conclusions

In this paper we introduced the notion of an *extended* wavelet coefficient as a flexible storage method for maintaining wavelet coefficients for datasets containing multiple measures. This flexible storage method bridges the gap between the two extreme storage hypotheses that the existing algorithms represent, and achieves better storage utilization, which results in improved accuracy to queries. We presented an optimal algorithm for selecting which *extended* wavelet coefficients to retain under a storage constraint such that the weighted sum of the squared L^2 error norms is minimized. We

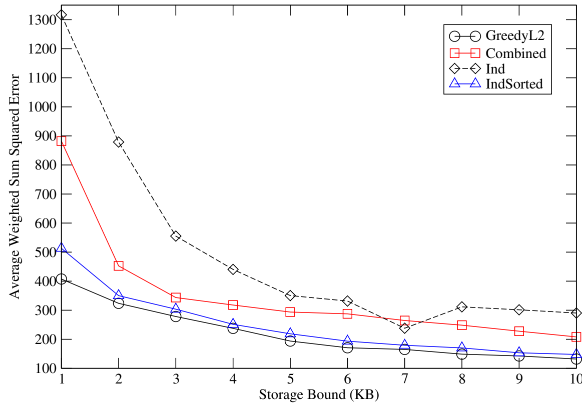

Figure 20: Sum Squared Errors
for Non-Normalized Real Dataset

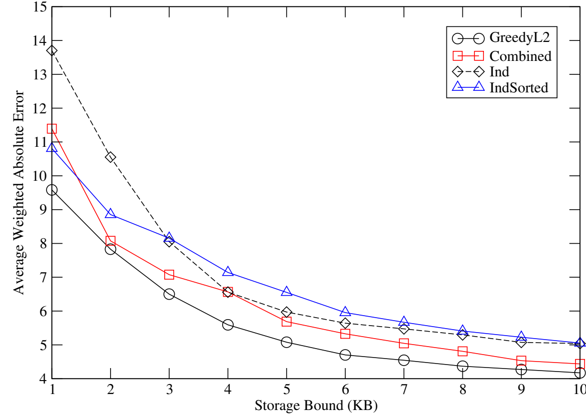

Figure 21: Absolute Errors
for Non-Normalized Real Dataset

Space (KB)	Average Weighted Relative Error (%)				
	GreedyL2	Combined	Ind	IndSorted	RS
1	57.5	161.7	202.9	59.5	154.2
2	52.3	62.4	148.1	51.2	167.4
3	37.2	61.5	59.6	49.3	136.7
4	35.8	59.9	39.9	43.3	124.6
5	32.2	26.9	39.3	40.7	125.4
6	29.6	26.2	39.0	35.5	115.8
7	20.7	25.5	38.5	34.4	108.7
8	18.3	25.1	38.3	28.9	115.9
9	16.6	21.4	32.3	28.0	109.8
10	16.7	21.3	43.5	26.8	112.6

Table 9: Relative Errors for Non-Normalized Real Dataset

then proposed an alternative greedy algorithm with reduced memory requirements and running time, which produces near optimal solutions for the same optimization problem. The results from our extensive experimental study validate the effectiveness of our approach.

References

[1] Pacific Northwest weather data. http://www-k12.atmos.washington.edu/k12/grayskies/nw_weather.html.

[2] N. Alon, Y. Mattias, and M. Szegedy. The Space Complexity of Approximating the Frequency Moments. In *Proc. of the 28th Annual ACM Symp. on the Theory of Computing*, 1996.

[3] T. Barclay, D. Slutz, and J. Gray. Terraserver: A spatial data warehouse, 2000.

[4] K. Chakrabarti, M. Garofalakis, R. Rastogi, and K. Shim. Approximate Query Processing Using Wavelets. In *Proc. of the 26th VLDB Conf.*, 2000.

[5] A. Deshpande, M. Garofalakis, and R. Rastogi. Independence is Good: Dependency-Based Histogram Synopses for Hight-Dimensional Data. In *ACM SIGMOD 2001*.

[6] M. Garofalakis and P. B. Gibbons. Wavelet Synopsis with Error Guarantees. In *ACM SIGMOD 2002*.

[7] P. B. Gibbons. Distinct Sampling for Highly-Accurate answers to Distinct Values Queries and Event Reports. In *Proc. of the 27th VLDB Conf.*, 2001.

[8] Y. E. Ioannidis and V. Poosala. Histogram-Based Approximation of Set-Valued Query Answers. In *Proc. of the 25th VLDB Conf*, 2000.

[9] B. Jawerth and W. Sweldens. Distinct Sampling for Highly-Accurate answers to Distinct Values Queries and Event Reports. In *VLDB 2001*.

[10] Y. Matias, J. S. Vitter, and M. Wang. Wavelet-Based Histograms for Selectivity Estimation. In *ACM SIGMOD 1998*.

[11] Y. Matias, J.S Vitter, and M. Wang. Dynamic Maintenance of Wavelet-Based Histograms. In *Proc. of the 26th VLDB Conf.*, 2000.

[12] A. Natse, R. Rastogi, and K. Shim. WALRUS: A Similarity Retrieval Algorithm for Image Databases. In *ACM SIGMOD 1999*.

[13] V. Poosala and V. Ganti. Fast Approximate Answers to Aggregate Queries on a Data Cube. In *Proc. of the 1999 Intl. Conf. on Scientific and Statistical Database Management*.

[14] V. Poosala and Y. E. Ioannidis. Selectivity Estimation Without the Attribute Value Independence Assumption. In *Proc. of the 23th VLDB Conf.*, 1997.

[15] E. J. Stollnitz, T. D. DeRose, and D. H. Salesin. Wavelets for Computer Graphics - Theory and Applications. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1996.

[16] N. Thaper, S. Guha, P. Indyk, and N. Koudas. Dynamic Multidimensional Histograms. In *ACM SIGMOD 2002*.

[17] J. S. Vitter. Random Sampling with a Reservoir. In *ACM TOMS*, 1985.

[18] J.S Vitter and M. Wang. Approximate Computation of Multidimensional Aggregates of Sparse Data Using Wavelets. In *ACM SIGMOD 1999*.

[19] J.S Vitter, M. Wang, and B. Iyer. Data Cube Approximation and Histograms via Wavelets. In *CIKM 1998*.