The VLDB Journal (2001) 10: 199–223 / Digital Object Identifier (DOI) 10.1007/s007780100049

Approximate query processing using wavelets

Kaushik Chakrabarti[⋆1], **Minos Garofalakis**[2], **Rajeev Rastogi**[2], **Kyuseok Shim**[3]

[1] University of Illinois, 1304 W. Springfield Ave., Urbana, IL 61801, USA; e-mail: kaushikc@cs.uiuc.edu
[2] Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, USA; e-mail: {minos,rastogi}@bell-labs.com
[3] KAIST and AITrc, 373-1 Kusong-Dong, Yusong-Gu, Taejon 305-701, Korea; e-mail: shim@cs.kaist.ac.kr

Edited by A. El Abbadi, G. Schlageter, K.-Y. Whang. Received: 7 August 2000 / Accepted: 1 April 2001
Published online: 7 June 2001 –

Abstract. Approximate query processing has emerged as a cost-effective approach for dealing with the huge data volumes and stringent response-time requirements of today's decision support systems (DSS). Most work in this area, however, has so far been limited in its query processing scope, typically focusing on specific forms of aggregate queries. Furthermore, conventional approaches based on sampling or histograms appear to be inherently limited when it comes to approximating the results of complex queries over high-dimensional DSS data sets. In this paper, we propose the use of multi-dimensional wavelets as an effective tool for general-purpose approximate query processing in modern, high-dimensional applications. Our approach is based on building *wavelet-coefficient synopses* of the data and using these synopses to provide approximate answers to queries. We develop novel query processing algorithms that operate directly on the wavelet-coefficient synopses of relational tables, allowing us to process arbitrarily complex queries *entirely* in the wavelet-coefficient domain. This guarantees extremely fast response times since our approximate query execution engine can do the bulk of its processing over compact sets of wavelet coefficients, essentially postponing the expansion into relational tuples until the end-result of the query. We also propose a novel wavelet decomposition algorithm that can build these synopses in an I/O-efficient manner. Finally, we conduct an extensive experimental study with synthetic as well as real-life data sets to determine the effectiveness of our wavelet-based approach compared to sampling and histograms. Our results demonstrate that our techniques: (1) provide approximate answers of better quality than either sampling or histograms; (2) offer query execution-time speedups of more than two orders of magnitude; and (3) guarantee extremely fast synopsis construction times that scale linearly with the size of the data.

Keywords: Query processing – Data synopses – Approximate query answers – Wavelet decomposition

1 Introduction

Approximate query processing has recently emerged as a viable solution for dealing with the huge amounts of data, the high query complexities, and the increasingly stringent response-time requirements that characterize today's decision support systems (DSS) applications. Typically, DSS users pose very complex queries to the underlying database management system (DBMS) that require complex operations over Gigabytes or Terabytes of disk-resident data and, thus, take a very long time to execute to completion and produce exact answers. Due to the *exploratory nature* of many DSS applications, there are a number of scenarios in which an exact answer may not be required, and a user may prefer a fast, approximate answer. For example, during a drill-down query sequence in ad hoc data mining, initial queries in the sequence frequently have the sole purpose of determining the truly interesting queries and regions of the database [12]. Providing (reasonably accurate) approximate answers to these initial queries gives users the ability to focus their explorations quickly and effectively, without consuming inordinate amounts of valuable system resources. An approximate answer can also provide useful feedback on how well-posed a query is, allowing DSS users to make an informed decision on whether they would like to invest more time and resources to execute their query to completion. Moreover, approximate answers obtained from appropriate *synopses* of the data may be the only available option when the base data is remote and unavailable [2]. Finally, for DSS queries requesting a numerical answer (e.g., total revenues or annual percentage), it is often the case that the full precision of the exact answer is not needed and the first few digits of precision will suffice (e.g., the leading few digits of a total in the millions or the nearest percentile of a percentage) [1].

Prior work. The strong incentive for approximate answers has spurred a flurry of research activity on approximate query processing techniques in recent years [1,7,9,11,12,16,26,33,34]. The majority of the proposed techniques, however, have been somewhat limited in their *query processing scope*, typically focusing on specific forms of *aggregate queries*. Besides the type of queries supported, another crucial aspect of an approximate query processing technique is the employed *data reduction mechanism*; that is, the method used to obtain syn-

⋆ Work done while visiting Bell Laboratories.

opses of the data on which the approximate query execution engine can then operate [3]. The methods explored in this context include *sampling* and, more recently, *histograms* and *wavelets*.

• *Sampling-based techniques* are based on the use of random samples as synopses for large data sets. Sample synopses can be either precomputed and incrementally maintained (e.g., [1, 7]) or they can be obtained progressively at run-time by accessing the base data using appropriate access methods (e.g., [11,12]). Random samples of a data collection typically provide accurate estimates for aggregate quantities (e.g., `count`s or `average`s), as witnessed by the long history of successful applications of random sampling in population surveys [4, 31] and selectivity estimation [20]. An additional benefit of random samples is that they can provide probabilistic guarantees on the quality of the estimated aggregate [10]. Sampling, however, suffers from two inherent limitations that restrict its applicability as an approximate query processing tool. First, a `join` operator applied on two uniform random samples results in a *non-uniform* sample of the join result that typically contains *very few tuples*, even when the join selectivity is fairly high [1]. Thus, `join` operations typically lead to significant degradations in the quality of an approximate aggregate. ("Join synopses" [1] provide a solution, but only for *foreign-key joins that are known beforehand*; that is, they cannot support arbitrary join queries over any schema.) Second, for a *non-aggregate* query, execution over random samples of the data is guaranteed to always produce a small subset of the exact answer which is often *empty* when `join`s are involved [1,16].

• *Histogram-based techniques* have been studied extensively in the context of query selectivity estimation [8,14,15,23,27, 28] and, more recently, as a tool for providing approximate query answers [16,26]. The very recent work of Ioannidis and Poosala [16] is the first to address the issue of obtaining practical approximations to *non-aggregate* query answers, making two important contributions. First, it proposes a novel error metric for quantifying the quality of an approximate set-valued answer (in general, a multiset of tuples). Second, it demonstrates how standard relational operators (like `join` and `select`) can be processed directly over histogram synopses of the data. The experimental results given in [16] prove that certain classes of histograms can provide higher-quality approximate answers compared to random sampling, when considering simple queries over low-dimensional data (one or two dimensions). It is a well-known fact, however, that histogram-based approaches become problematic when dealing with the high-dimensional data sets that are typical of modern DSS applications. The reason is that, as the dimensionality of the data increases, both the *storage overhead* (i.e., number of buckets) and the *construction cost* of histograms that can achieve reasonable error rates increase in an explosive manner [19,33]. The dimensionality problem is further exacerbated by `join` operations that can cause the dimensionality of intermediate query results (and the corresponding histograms) to explode.

• *Wavelet-based techniques* provide a mathematical tool for the hierarchical decomposition of functions, with a long history of successful applications in signal and image processing [18,24,32]. Recent studies have demonstrated the applicability of wavelets to selectivity estimation [21] and the approximation of range-sum queries over OLAP data cubes [33, 34]. The idea is to apply wavelet decomposition to the input data collection (attribute column(s) or OLAP cube) and retain the best few *wavelet coefficients* as a compact synopsis of the input data. The results of Vitter et al. [33,34] have shown that wavelets are effective in handling aggregates over high-dimensional OLAP cubes, while avoiding the high construction costs and storage overheads of histograming techniques. Their wavelet decomposition requires only a logarithmically small number of passes over the data (regardless of the dimensionality) and their experiments prove that a few wavelet coefficients suffice to produce surprisingly accurate results for summation aggregates. Nevertheless, the focus of these earlier studies has always been on a very specific form of queries (i.e., range-sums) over a single OLAP table. Thus, the problem of whether wavelets can provide a solid foundation for general-purpose approximate query processing has hitherto been left unanswered.

Our contributions. In this paper, we significantly extend the scope of earlier work on approximate query answers, establishing the viability and effectiveness of wavelets as a generic approximate query processing tool for modern, high-dimensional DSS applications. More specifically, we propose a novel approach to general-purpose approximate query processing that consists of two basic steps. First, multi-dimensional Haar wavelets are used to efficiently construct compact synopses of general relational tables. Second, using novel query processing algorithms, standard SQL operators (both aggregate and non-aggregate) are evaluated *directly* over the wavelet-coefficient synopses of the data to obtain fast and accurate approximate query answers. The crucial observation here is that, as we demonstrate in this work, our approximate query execution engine can do all of its processing *entirely in the wavelet-coefficient domain*; that is, both the input(s) and the output of our query processing operators are compact collections of wavelet coefficients capturing the underlying relational data. This implies that, for any arbitrarily complex query, we can defer expanding the wavelet-coefficient synopses back into relational tuples till the very end of the query, thus allowing for extremely fast approximate query processing. (In contrast, the histogram-based `join` processing algorithm of Ioannidis and Poosala [16] requires each histogram to be partially expanded to generate the tuple-value distribution for the corresponding approximate relation. As our results demonstrate, this requirement can slow down join processing over histograms significantly, since the partially expanded histogram can give rise to large numbers of tuples, *especially* for high-dimensional data.) The contributions of our work are summarized as follows.

• **New, I/O-efficient wavelet decomposition algorithm for relational tables.** The methodology developed in this paper is based on a different form of the multi-dimensional Haar transform than that employed by Vitter et al. [33,34]. As a consequence, the decomposition algorithms proposed by Vitter and Wang [33] are not applicable. We address this problem by developing a novel, I/O-efficient algorithm for building the wavelet-coefficient synopsis of a relational table. The worst-

case I/O complexity of our algorithm matches that of the best algorithms of Vitter and Wang, requiring only a logarithmically small number of passes over the data. Furthermore, there exist scenarios (e.g., when the table is stored in *chunks* [5,30]) under which our decomposition algorithm can work in a *single pass* over the input table.

• **Novel query processing algebra for wavelet-coefficient data synopses.** We propose a new algebra for approximate query processing that operates *directly over the wavelet-coefficient synopses of relations*, while guaranteeing the correct relational operator semantics. Our algebra operators include the conventional aggregate and non-aggregate SQL operators, like `select`, `project`, `join`, `count`, `sum`, and `average`. Based on the semantics of Haar wavelet coefficients, we develop novel query processing algorithms for these operators that work *entirely* in the wavelet-coefficient domain. This allows for extremely fast response times, since our approximate query execution engine can do the bulk of its processing over compact wavelet-coefficient synopses, essentially postponing the expansion into relational tuples until the end-result of the query. We also propose an efficient algorithm for this final *rendering* step, i.e., for expanding a set of multi-dimensional Haar coefficients into an approximate relation which is returned to the user as the final (approximate) answer of the query.

• **Extensive experiments validating our approach.** We have conducted an extensive experimental study with synthetic as well as real-life data sets to determine the effectiveness of our wavelet-based approach compared to sampling and histograms. Our results demonstrate that: (1) the quality of approximate answers obtained from our wavelet-based query processor is, in general, better than that obtained by either sampling or histograms for a wide range of `select`, `project`, `join`, and aggregate queries; (2) query execution-time speedups of more than two orders of magnitude are made possible by our approximate query processing algorithms; and (3) our wavelet decomposition algorithm is extremely fast and scales linearly with the size of the data.

Roadmap. The remainder of this paper is organized as follows. After reviewing some necessary background material on the Haar wavelet decomposition, Sect. 2 presents our I/O-efficient wavelet decomposition algorithm for multi-attribute relational tables. In Sect. 3, we develop our query algebra and operator processing algorithms for wavelet-coefficient data synopses. Section 3 also proposes an efficient rendering algorithm for multi-dimensional Haar coefficients. In Sect. 4, we discuss the findings of an extensive experimental study of our wavelet-based approximate query processor using both synthetic and real-life data sets. Finally, Sect. 5 concludes the paper.

2 Building synopses of relational tables using multi-dimensional wavelets

2.1 Background: the wavelet decomposition

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 together with detail coefficients that influence the function at various scales [32]. The wavelet decomposition has excellent energy compaction and de-correlation properties, which can be used to effectively generate compact representations that exploit the structure of data. Furthermore, wavelet transforms can generally be computed in linear time, thus allowing for very efficient algorithms.

The work in this paper is based on the multi-dimensional *Haar wavelet* decomposition. Haar wavelets are conceptually simple, very fast to compute, and have been found to perform well in practice for a variety of applications ranging from image editing and querying [24,32] to selectivity estimation and OLAP approximations [21,33]. Recent work has also investigated methods for dynamically maintaining Haar-based data representations [22]. In this section, we discuss Haar wavelets in both one and multiple dimensions.

One-dimensional Haar wavelets. Suppose we are given a one-dimensional data vector A containing the following four values $A = [2, 2, 5, 7]$. 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 $[2, 6]$. In other words, the average of the first two values (that is, 2 and 2) is 2 and that of the next two values (that is, 5 and 7) is 6. Obviously, some information has been lost in this averaging process. To be able to restore the original four values of the data array, we need to 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 0 since $2 - 2 = 0$, while for the second we need to store -1 since $6 - 7 = -1$. Note that it is possible to reconstruct the four values of the original data array from the lower-resolution array containing the two averages and the two detail coefficients. Recursively applying the above pairwise averaging and differencing process on the lower-resolution array containing the averages, we get the following full decomposition.

Resolution	Averages	Detail coefficients
2	[2, 2, 5, 7]	–
1	[2, 6]	[0, -1]
0	[4]	[-2]

We define the *wavelet transform* (also known as the *wavelet decomposition*) of A to be the single coefficient representing the 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 $W_A = [4, -2, 0, -1]$. 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, giving a very effective form of lossy data compression.

Note that, intuitively, wavelet coefficients carry different weights with respect to their importance in rebuilding the original data values. For example, the overall average is obviously more important than any detail coefficient since it affects the reconstruction of all entries in the data array. In order to equalize the importance of all wavelet coefficients, we need to *normalize* the final entries of W_A appropriately. This is achieved by dividing each wavelet coefficient by $\sqrt{2^l}$, where l denotes the *level of resolution* at which the coefficient appears (with $l = 0$ corresponding to the "coarsest" resolution level). Thus, the normalized wavelet transform for our example data array becomes $W_A = [4, -2, 0, -1/\sqrt{2}]$.

Multi-dimensional Haar wavelets. There are two common methods in which Haar wavelets can be extended to transform the data values in a *multi-dimensional* array. Each of these transforms is a generalization of the one-dimensional decomposition process described above. To simplify the exposition to the basic ideas of multi-dimensional wavelets, we assume all dimensions of the input array to be of equal size.

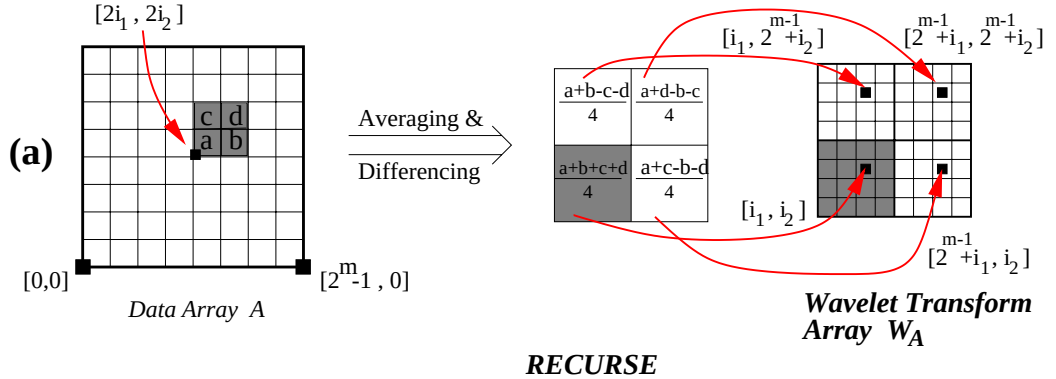
The first method is known as *standard decomposition*. In this method, we first fix an ordering for the data dimensions (say, $1, 2, \ldots, d$) and then proceed to apply the complete one-dimensional wavelet transform for each one-dimensional "row" of array cells along dimension k, for all $k = 1, \ldots, d$. The standard Haar decomposition forms the basis of the recent results of Vitter et al., on OLAP data cube approximations [33, 34].

The work presented in this paper is based on the second method of extending Haar wavelets to multiple dimensions, namely the *nonstandard decomposition*. Abstractly, the nonstandard Haar decomposition alternates between dimensions during successive steps of pairwise averaging and differencing: given an ordering for the data dimensions $(1, 2, \ldots, d)$, we perform *one step of pairwise averaging and differencing* for each one-dimensional row of array cells along dimension k, for each $k = 1, \ldots, d$. (The results of earlier averaging and differencing steps are treated as data values for larger values of k.) This process is then repeated recursively only on the quadrant containing averages across all dimensions. One way of conceptualizing (and implementing [24]) this procedure is to think of a $2 \times 2 \times \cdots \times 2 (= 2^d)$ hyper-box being shifted across the data array, performing pairwise averaging and differencing, distributing the results to the appropriate locations of the wavelet transform array W_A (with the averages for each box going to the "lower-left" quadrant of W_A) and, finally, recursing the computation on the lower-left quadrant of W_A. This procedure is demonstrated pictorially for a (two-dimensional) $2^m \times 2^m$ data array A in Fig. 1a. More specifically, Fig. 1a shows the pairwise averaging and differencing step for one positioning of the 2×2 box with its "root"(i.e., lower-left corner) located at the coordinates $[2i_1, 2i_2]$ of A followed by the distribution of the results in the wavelet transform array. This step is repeated for every possible combination of i_j's, $i_j \in \{0, \ldots, 2^{m-1} - 1\}$. Finally , the process is recursed only on the lower-left quadrant of W_A (containing the averages collected from all boxes). A detailed description of the nonstandard Haar decomposition can be found in any standard reference on the subject (e.g., [18, 32]).

Example 1. Consider the 4×4 array A shown in Fig. 1b.1. Figure 1b.2 shows the result of the first horizontal and vertical pairwise averaging and differencing on the 2×2 hyper-boxes of the original array. During this first level of recursion, the 2×2 sliding hyper-box is placed at the four possible "root" positions on A, namely $[0, 0]$, $[0, 2]$, $[2, 0]$, and $[2, 2]$, and pairwise averaging and differencing is performed on each of them individually. For example, pairwise averaging and differencing on the hyper-box with root position $[2, 0]$ (containing values $A[2, 0] = 2$, $A[3, 0] = 4$, $A[2, 1] = 6$, and $A[3, 1] = 8$) produces the average coefficient $(A[2, 0] + A[3, 0] + A[2, 1] + A[3, 1])/4 = 5$, and detail coefficients $(A[2, 0] + A[2, 1] - A[3, 0] - A[3, 1])/4 = -1$, $(A[2, 0] + A[3, 0] - A[2, 1] - A[3, 1])/4 = -2$, and $(A[2, 0] + A[3, 1] - A[3, 0] - A[2, 1])/4 = 0$ (shown in the same positions ($A[2, 0]$, $A[3, 0]$, $A[2, 1]$, and $A[3, 1]$). The averages of the values for each positioning of the hyper-box are then assigned to the 2×2 lower-left quadrant of the wavelet transform array W_A, while the detail coefficients are distributed in the three remaining 2×2 quadrants of W_A, as shown in Fig. 1b.3. As an example, for the hyper-box with root position $[2, 0]$ (i.e., $i_1 = 1$, $i_2 = 0$, and $m = 2$, using the notation of Fig. 1a), the results $5, -1, -2$, and 0 are placed at positions $[i_1, i_2] = [1, 0]$, $[2^{m-1} + i_1, i_2] = [3, 0]$, $[i_1, 2^{m-1} + i_2] = [1, 2]$, and $[2^{m-1} + i_1, 2^{m-1} + i_2] = [3, 2]$, respectively. The process is then recursed on the lower-left quadrant of W_A (containing the average values $2.5, 7.5, 5$, and 10 from the four hyper-boxes), resulting in the average coefficient 6.25 and detail coefficients -1.25, -2.5, and 0. That ends the recursive decomposition process, producing the final wavelet transform array W_A shown in Fig. 1b.4. □

As noted in the wavelet literature, both methods for extending one-dimensional Haar wavelets to higher dimensionalities have been used in a wide variety of application domains and, to the best of our knowledge, none has been shown to be uniformly superior. Our choice of the nonstandard method was mostly motivated by our earlier experience with nonstandard two-dimensional Haar wavelets in the context of effective image retrieval [24]. An advantage of using the nonstandard transform is that, as we explain later in the paper, it allows for an efficient representation of the sign information for wavelet coefficients. This efficient representation stems directly from the construction process for a nonstandard Haar basis [32]. Using nonstandard Haar wavelets, however, also implies that the standard decomposition algorithms of Vitter and Wang [33] are no longer applicable. We address this problem by proposing a novel I/O-efficient algorithm for constructing the nonstandard wavelet decomposition of a relational table (Sect. 2.2). (We often omit the "nonstandard" qualification in what follows.)

Multi-dimensional Haar coefficients: semantics and representation. Consider a wavelet coefficient W generated during the multi-dimensional Haar decomposition of a d-dimensional data array A. From a mathematical standpoint, this coefficient is essentially a multiplicative factor for an appropriate *Haar basis function* when the data in A is expressed using the d-dimensional Haar basis [32]. The d-dimensional Haar basis function corresponding to W is defined by: (1) a *d-dimensional rectangular support region* in A that captures the region of A's cells that W contributes to during reconstruction; and (2) the *quadrant sign information* that defines



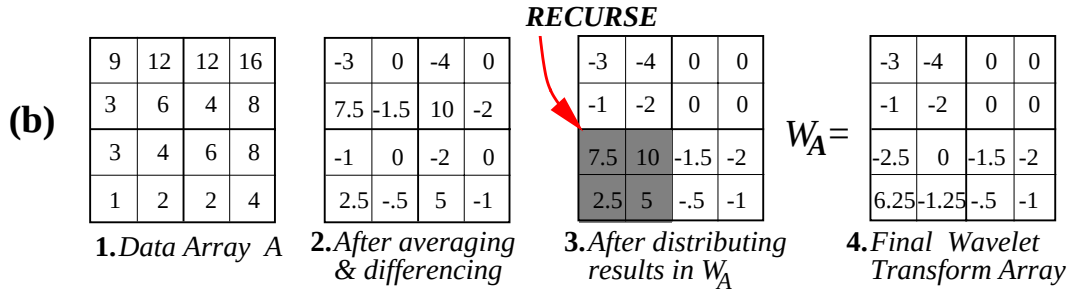


Fig. 1a,b. Non-standard decomposition in two dimensions. **a** Computing pairwise averages and differences and distributing them in the wavelet transform array; **b** example decomposition of a 4×4 array

the sign ($+$ or $-$) of W's contribution (i.e., $+W$ or $-W$) to any cell contained in a given quadrant of its support rectangle. (Note that the wavelet decomposition process guarantees that this sign can only change across quadrants of the support region.) As an example, Fig. 2a depicts the support regions and signs of the sixteen nonstandard, two-dimensional Haar basis functions for coefficients in the corresponding locations of a 4×4 wavelet transform array W_A. 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. Thus, $W_A[0,0]$ is the overall average that contributes positively (i.e.,"$+W_A[0,0]$") to the reconstruction of all values in A, whereas $W_A[3,3]$ is a detail coefficient that contributes (with the signs shown in Fig. 2a) only to values in A's upper right quadrant. 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. Figure 2a also depicts the two *levels of resolution* ($l = 0, 1$) for our example two-dimensional Haar coefficients; as in the one-dimensional case, these levels define the appropriate constants for normalizing coefficient values (see, e.g., [32]).

Example 2. In light of Fig. 2a, let us now revisit Example 1 and consider how the entries of W_A contribute to the reconstruction of values in A. As we have already observed, coefficient $W_A[0,0] = 6.25$ is the overall average that contributes positively (i.e., $+6.25$) to the reconstruction of all sixteen data values in A. On the other hand, the detail coefficient $W_A[0,2] = -1$ affects the reconstruction of only the four data values in the lower-left quadrant of A, contributing -1 to $A[0,0]$ and $A[1,0]$, and $-(-1) = +1$ to $A[0,1]$ and $A[1,1]$. Similarly, the detail coefficient $W_A[2,0] = -.5$ contributes $-.5$ to $A[0,0]$ and $A[0,1]$, and $+.5$ to $A[1,0]$ and $A[1,1]$. For example, based on Fig. 2a, the data value $A[0,1]$ can be reconstructed using the following formula:

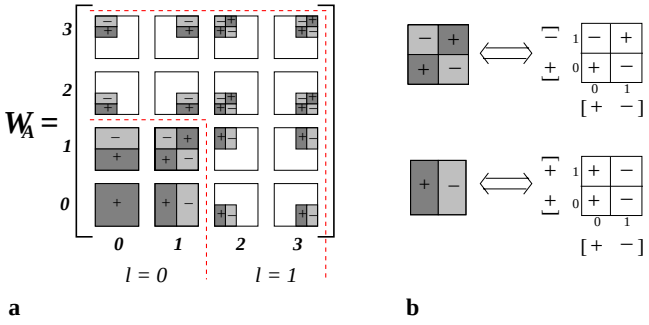


Fig. 2. a Support regions and signs for the sixteen nonstandard two-dimensional Haar basis functions. The coefficient magnitudes are multiplied by $+1$ (-1) where a sign of $+$ (respectively, $-$) appears, and 0 in blank areas; **b** representing quadrant sign information for coefficients using "per-dimension" sign vectors

$$\begin{aligned} A[0,1] &= +W_A[0,0] + W_A[0,1] + W_A[1,0] + W_A[1,1] \\ &\quad -W_A[0,2] + W_A[2,0] - W_A[2,2] \\ &= 2.5 - (-1) + (-.5) = 3. \quad \square \end{aligned}$$

To simplify the discussion in this paper, we abstract away the distinction between a coefficient and its corresponding basis function by representing a Haar wavelet coefficient with the triple $W = \langle R, S, v \rangle$, where:

1. $W.R$ is the *d-dimensional support hyper-rectangle of W* enclosing all the cells in the data array A to which W contributes (i.e., the support of the corresponding basis function). We represent this hyper-rectangle by its low and high boundary values (i.e., starting and ending array cells) along each dimension j, $1 \leq j \leq d$; these are denoted by $W.R.boundary[j].lo$ and $W.R.boundary[j].hi$, respectively. Thus, the coefficient W contributes to each data cell

$A[i_1, \ldots, i_d]$ satisfying the condition $W.R.boundary[j].lo \leq i_j \leq W.R.boundary[j].hi$ for all dimensions j, $1 \leq j \leq d$. For example, for the detail coefficient $W_A[1,2]$ in Fig. 2a, $W.R.boundary[1].lo = 2$, $W.R.boundary[1].hi = 3$, $W.R.boundary[2].lo = 0$, and $W.R.boundary[2].hi = 1$. The space required to store the support hyper-rectangle of a coefficient is $2 \log N$ bits, where N denotes the total number of cells of A.

2. $W.S$ stores the *sign information for all d-dimensional quadrants of W.R.* Storing the quadrant sign information directly would mean a space requirement of $O(2^d)$, i.e., proportional to the number of quadrants of a d-dimensional hyper-rectangle. Instead, we use a more space-efficient representation of the quadrant sign information (using only $2d$ bits) that exploits the regularity of the nonstandard Haar transform. The basic observation here is that a nonstandard d-dimensional Haar basis is formed by scaled and translated products of d one-dimensional Haar basis functions [32]. Thus, our idea is to store a 2-bit *sign vector* for each dimension j, that captures the sign variation of the corresponding one-dimensional basis function. The two elements of the sign vector of coefficient W along dimension j are denoted by $W.S.sign[j].lo$ and $W.S.sign[j].hi$, and contain the sign that corresponds to the lower and upper half of $W.R$'s extent along dimension j, respectively. Given the sign vectors along each dimension and treating a sign of $+$ ($-$) as being equivalent to $+1$ (respectively, -1), the sign for each d-dimensional quadrant can be computed as the product of the d sign-vector entries that map to that quadrant; that is, following exactly the basis construction process. (Note that we will continue to make use of this "+1/-1" interpretation of signs throughout the paper.) Our sign-computation methodology is depicted in Fig. 2b for two example coefficient hyper-rectangles from Fig. 2a. The sign vectors for the topmost coefficient in Fig. 2b are $W.S.sign[1].lo = +1$ and $W.S.sign[1].hi = -1$ along dimension 1 (x-axis), and $W.S.sign[2].lo = +1$ and $W.S.sign[2].hi = -1$ along dimension 1 (y-axis); thus, the signs of the lower left, lower right, upper left, and upper right quadrants of its support region are computed as $W.S.sign[1].lo * W.S.sign[2].lo = +1$, $W.S.sign[1].hi * W.S.sign[2].lo = -1$, $W.S.sign[1].lo * W.S.sign[2].hi = -1$, and $W.S.sign[1].hi * W.S.sign[2].hi = +1$, respectively.
3. $W.v$ is the *(scalar) magnitude of coefficient W*. This is exactly the quantity that W contributes (either positively or negatively, depending on $W.S$) to all data array cells enclosed in $W.R$. For example, the magnitude of $W_A[0,0]$ in Fig. 1b is 6.25, and that of $W_A[1,2]$ is -2.

Thus, our view of a d-dimensional Haar wavelet coefficient is that of a d-dimensional hyper-rectangle with a magnitude and a sign that may change across quadrants. Note that, by the properties of the nonstandard Haar decomposition, given *any pair* of coefficients, their hyper-rectangles are either *completely disjoint* or one is *completely contained* in the other; that is, coefficient hyper-rectangles cannot *partially overlap*. As will be seen later, it is precisely these containment properties coupled with our sign-vector representation of quadrant signs that enable us to efficiently perform `join` operations directly over wavelet-coefficient synopses.

2.2 Building and rendering wavelet-coefficient synopses

Consider a relational table R with d attributes $X_1, X_2, \ldots X_d$. A straightforward way of obtaining a wavelet-based synopsis of R would be to take the traditional two-dimensional array view of a relational table (with attributes on the x-axis and tuples on the y-axis), apply a two-dimensional wavelet decomposition on R, and retain a few large coefficients. It is highly unlikely, however, that this solution will produce a high-quality compression of the underlying data. The reason is that wavelets (like most compression mechanisms) work by exploiting locality (i.e., clusters of constant or similar values), which is almost impossible when grouping together attributes that can have vastly different domains (e.g., consider an `age` attribute adjacent to a `salary` attribute). Similar problems occur in the vertical grouping as well, since even sorting by some attribute(s) cannot eliminate large "spikes" for others. We address these problems by taking a slightly different view of the d-attribute relational table R. We can represent the information in R as a d-dimensional array A_R, whose j^{th} dimension is indexed by the values of attribute X_j and whose cells contain the count of tuples in R having the corresponding combination of attribute values. A_R is essentially the *joint frequency distribution* of all the attributes of R. Figure 3a depicts the tuples of an example relation with two attributes (the "Count" column simply records the number of tuple occurrences); the corresponding joint-frequency array is shown in Fig. 3b. We obtain the wavelet synopsis of R by constructing the nonstandard multi-dimensional wavelet decomposition of A_R (denoted by W_R) and then retaining only some of the coefficients (based on the desired size of the synopsis) using a thresholding scheme. In this section, we propose a novel, I/O-efficient algorithm for constructing W_R. Note that, even though our algorithm computes the decomposition of A_R, it in fact works off the "set-of-tuples" (ROLAP) representation of R. (As noted by Vitter and Wang [33], this is a requirement for computational efficiency since the joint-frequency array A_R is typically very sparse, especially for the high-dimensional data sets that are typical of DSS applications.) We also briefly describe our thresholding scheme for controlling the size of a wavelet-coefficient synopsis. We have also developed a time- and space-efficient algorithm (termed `render`) for *rendering* (i.e., expanding) a synopsis into an approximate "set-of tuples" relation (which is used during query processing as the final step). We begin by summarizing the notational conventions used throughout the paper.

Notation. Let $\mathcal{D} = \{D_1, D_2, \ldots, D_d\}$ denote the set of dimensions of A_R, where dimension D_j corresponds to the *value domain* of attribute X_j. Without loss of generality, we assume that each dimension D_j is indexed by the set of integers $\{0, 1, \cdots, |D_j|-1\}$, where $|D_j|$ denotes the size of dimension D_j. We assume that the attributes $\{X_1, \ldots, X_d\}$ are ordinal in nature, that is, their domains are naturally ordered. This captures all numeric attributes (e.g., age, income) and some categorical attributes (e.g., education). Such domains can always be mapped to the set of integers mentioned above while

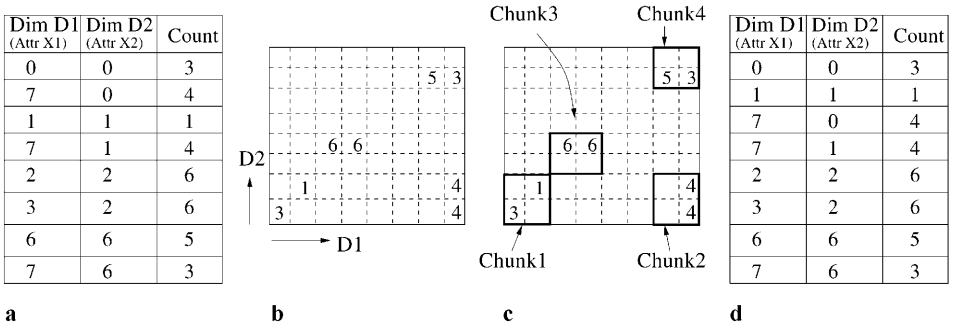


Fig. 3. a An example relation R with two data attributes (X_1 and X_2); **b** the corresponding joint-frequency array A_R; **c** one possible chunking of A_R; all cells inside a chunk are stored contiguously on disk; **d** the corresponding chunked organization of R; all tuples belonging to the same chunk are stored contiguously

preserving the natural domain order and, hence, the locality of the distribution. It is also possible to map unordered domains to integer values; however, such mappings do not always preserve locality. For example, mapping countries to integers using alphabetic ordering can destroy data locality. There may be alternate mappings that are more locality preserving, e.g., assigning neighboring integers to neighboring countries. (Effective mapping techniques for unordered attributes are an open research issue that lies beyond the scope of this paper; a methodology based on concept hierarchies has recently been discussed in [6].) The d-dimensional joint-frequency array A_R comprises $N = \prod_{i=1}^{d} |D_i|$ cells with cell $A_R[i_1, i_2, \ldots, i_d]$ containing the count of tuples in R having $X_j = i_j$ for each attribute $1 \leq j \leq d$. We define N_z to be the number of populated (i.e., non-zero) cells of A_R (typically, $N_z << N$). Table 1 outlines the notation used in this paper with a brief description of its semantics. We provide detailed definitions of some of these parameters in the text. Additional notation will be introduced when necessary.

Most of the notation pertaining to wavelet coefficients W has already been described in Sect. 2.1. The only exception is the *sign-change value vector* $W.S.signchange[j]$ that captures the value along dimension j (between $W.R.boundary[j].lo$ and $W.R.boundary[j].hi$) at which a transition in the value of the sign vector $W.S.sign[j]$ occurs, for each $1 \leq j \leq d$. That is, the sign $W.S.sign[j].lo$ ($W.S.sign[j].hi$) applies to the range $[W.R.boundary[j].lo, \ldots, W.S.signchange[j]-1]$ (respectively, $[W.S.signchange[j], \ldots, W.R.boundary[j].hi]$). As a convention, we set $W.S.signchange[j]$ equal to $W.R.boundary[j].lo$ when there is no "true" sign change along dimension j, i.e., $W.S.sign[j]$ contains $[+,+]$ or $[-,-]$. Note that, for base Haar coefficients with a true sign change along dimension j, $W.S.signchange[j]$ is simply the midpoint between $W.R.boundary[j].lo$ and $W.R.boundary[j].hi$ (Fig. 2). This property, however, no longer holds when arbitrary selections and joins are executed over the wavelet coefficients. As a consequence, we need to store sign-change values explicitly in order to support general query processing operations in an efficient manner.

The COMPUTEWAVELET *decomposition algorithm.* We now present our I/O-efficient algorithm (called COMPUTEWAVELET) for constructing the wavelet decomposition of R. Our algorithm exploits the interaction of nonstandard wavelet decomposition and "chunk-based" organizations of relational tables [30,5]. In chunk-based organizations, the joint-frequency array A_R is split into d-dimensional *chunks* and tuples of R belonging to the same chunk are stored contiguously on disk. Figures 3c,d depict an example chunking of A_R and the corresponding organization of R's tuples. If R is organized in chunks, COMPUTEWAVELET can perform the decomposition in a *single pass* over the tuples of R. Note that such data organizations have already been proposed in earlier work (e.g., the *chunked-file organization* of Deshpande et al. [5] and Orenstein's *z-order* linearization [17,25]), where they have been shown to have significant performance benefits for DSS applications due to their excellent multi-dimensional clustering properties.

We present our I/O-efficient COMPUTEWAVELET algorithm below assuming that R's tuples are organized in d-dimensional chunks. If R is not chunked, then an extra pre-processing step is required to reorganize R on disk. This pre-processing is no more expensive than a sorting step (e.g., in *z-order*) which requires a logarithmic number of passes over R. Thus, while the wavelet decomposition requires just a single pass when R is chunked, in the worst-case (i.e., when R is not "chunked"), the I/O complexity of COMPUTEWAVELET matches that of Vitter and Wang's I/O-efficient algorithm for standard Haar wavelet decomposition [33]. We also assume that each chunk can individually fit in memory. We show that the extra memory required by our wavelet decomposition algorithm (in addition to the memory needed to store the chunk itself) is at most $O(2^d \cdot \log(\max_j\{|D_j|\}))$. Finally, our implementation of COMPUTEWAVELET also employs a dynamic coefficient-thresholding scheme that adjusts the number of wavelet coefficients maintained during the decomposition based on the desired size of the synopsis. We do not discuss the details of our dynamic-thresholding step below to keep the presentation of COMPUTEWAVELET simple.

Our I/O-efficient decomposition algorithm is based on the following observation:

> *The decomposition of a d-dimensional array A_R can be computed by independently computing the decomposition for each of the 2^d d-dimensional subarrays corresponding to A_R's quadrants and then performing pairwise averaging and differencing on the computed 2^d averages of A_R's quadrants.*

Due to the above property, when a chunk is loaded from the disk for the first time, COMPUTEWAVELET can perform the entire computation required for decomposing the chunk right away (hence no chunk is read twice). Lower resolution coefficients are computed by first accumulating, in main memory, averages from the 2^d quadrants (generated from the previous level of resolution) followed by pairwise averaging and differ-

Table 1. Notation

Symbol	Semantics
d	Number of attributes (i.e., dimensionality) of the input relational table
R, A_R	Relational table and corresponding d-dimensional joint-frequency array
X_j , D_j	j^{th} attribute of relation R and corresponding domain of values ($1 \leq j \leq d$)
$\mathcal{D} = \{D_1, \ldots, D_d\}$	Set of all data dimensions of the array A_R
$A_R[i_1, i_2, \cdots, i_d]$	Count of tuples in R with $X_j = i_j$ ($i_j \in \{0, \ldots, \lvert D_j \rvert - 1\}$) , $\forall 1 \leq j \leq d$
$N = \prod_j \lvert D_j \rvert$	Size (i.e., number of cells) of A_R
N_z	Number of *non-zero* cells of A_R ($N_z << N$)
$W_R[i_1, i_2, \cdots, i_d]$	Coefficient located at coordinates $[i_1, i_2, \cdots, i_d]$ of the wavelet transform array W_R
$W.R.boundary[j].\{lo, hi\}$	Support hyper-rectangle boundaries along dimension D_j for coefficient W ($1 \leq j \leq d$)
$W.S.sign[j].\{lo, hi\}$	Sign vector information along dimension D_j for the wavelet coefficient W ($1 \leq j \leq d$)
$W.S.signchange[j]$	Sign-change value along dimension D_j for the wavelet coefficient W ($1 \leq j \leq d$)
$W.v$	Scalar magnitude of the wavelet coefficient W
l	Current level of resolution of the wavelet decomposition

encing, thus requiring no extra I/O. Due to the depth-first nature of the algorithm, the pairwise averaging and differencing is performed *as soon as* all the 2^d averages are accumulated, making the algorithm memory efficient (as, at any point in the computation, there can be no more than one "active" subarray (whose averages are still being accumulated) for each level of resolution).

The outline of our I/O-efficient wavelet decomposition algorithm COMPUTEWAVELET is depicted in Fig. 4. To simplify the presentation, the COMPUTEWAVELET pseudo-code assumes that all dimensions of the data array A_R are of equal size, i.e., $|D_1| = |D_2| = \ldots = |D_d| = 2^m$. We discuss handling of unequal dimension sizes later in this section. Besides the input joint-frequency array (A_R) and the logarithm of the dimension size (m), COMPUTEWAVELET takes two additional arguments: (a) the *root (i.e., "lower-left" endpoint) coordinates* of the d-dimensional subarray for which the wavelet transform is to be computed $(i_1, i_2, \ldots, i_d)$; and (b) the current *level of resolution* for the wavelet coefficients (l). Note that, for a given level of resolution l, the extent (along each dimension) of the d-dimensional array rooted at $(i_1, i_2, \ldots, i_d)$ being processed is exactly 2^{m-l}. The algorithm computes the wavelet coefficients for the elements in the input subarray and returns the overall average (Step 15). The wavelet-coefficient computation is carried out by: (1) performing wavelet decomposition *recursively* on each of the 2^d quadrants of the input subarray (to produce the corresponding wavelet transforms for the next level of resolution, i.e., $l + 1$), and collecting the quadrant averages returned in a $2 \times \cdots \times 2 = 2^d$ temporary hyper-box T (Steps 2–4); (2) performing pairwise averaging and differencing on T to produce the average and detail coefficients for the level-l decomposition of the input subarray (Step 5); and, finally, (3) distributing these level-l wavelet coefficients to the appropriate locations of the wavelet transform array W_R (computing their support hyper-rectangles and dimension sign vectors at the same time) (Steps 6–14). The initial invocation of COMPUTEWAVELET is done with root $(i_1, i_2, \ldots, i_d) = (0, 0, \ldots, 0)$ and level $l = 0$.

Example 3. Figure 5 illustrates the working of the COMPUTEWAVELET algorithm on the example 8×8 joint-frequency

procedure COMPUTEWAVELET(A_R, m, $(i_1, i_2, \ldots, i_d)$, l)
begin
1. **if** $l \geq m$ **return** $A_R[i_1, \ldots, i_d]$
2. **for** t_1 := 0,1 $\cdots$ **for** t_d := 0,1
3. $T[t_1, \ldots, t_d]$:= COMPUTEWAVELET(A_R, m, $(i_1 + t_1 \cdot 2^{m-l-1}, \ldots, i_d + t_d \cdot 2^{m-l-1}), l + 1$)
4. **end** $\cdots$ **end**
5. *perform pairwise averaging and differencing on the* $2 \times \ldots \times 2 = 2^d$ *hyper-box* T
6. **for** t_1 := 0,1 $\cdots$ **for** t_d := 0,1
7. $arrayIndex$:= ($t_1 \cdot 2^l + \frac{i_1}{2^{m-l}}, \ldots, t_d \cdot 2^l + \frac{i_d}{2^{m-l}}$)
8. $W_R[arrayIndex].v := T[t_1, \ldots, t_d]$
9. **for** $j := 1, \ldots, d$
10. $W_R[arrayIndex].R.boundary[j] := [i_j, i_j + 2^{m-l} - 1]$
11. $W_R[arrayIndex].S.sign[j] :=$ $(t_j == 0)\ ?\ [+,+]\ :\ [+,-]$
12. $W_R[arrayIndex].S.signchange[j] :=$ $(t_j == 0)\ ?\ i_j\ :\ i_j + 2^{m-l}$
13. **end**
14. **end** $\cdots$ **end**
15. **return** $T[0, \ldots, 0]$
end

Fig. 4. COMPUTEWAVELET: an I/O-efficient wavelet decomposition algorithm

array A_R corresponding to the relation shown in Fig. 3. The recursive calls of COMPUTEWAVELET for the four resolution levels l form a depth-first invocation tree; the root of the tree (i.e., $l = 0$) corresponds to the initial invocation COMPUTEWAVELET(A_R, 3, $(0,0)$, 0) with the entire array A_R as the input subarray. The root then recursively invokes COMPUTEWAVELET on the subarrays corresponding to the four 4×4 quadrants of A_R "rooted" at cells $(0,0)$, $(0,4)$, $(4,0)$, and $(4,4)$ to compute the wavelet coefficients at the next level of resolution ($l = 1$). COMPUTEWAVELET(A_R, 3, $(0,0)$, 1) in turn invokes COMPUTEWAVELET on the four 2×2 quadrants (rooted at $(0,0)$, $(0,2)$, $(2,0)$, and $(2,2)$) of its input subarray ($l = 2$). Similarly, COMPUTEWAVELET(A_R, 3, $(0,0)$, 2) invokes COMPUTEWAVELET on its four 1×1 "quadrants" (i.e., simple cells) located at (0.0), $(0,1)$, $(1,0)$, and $(1,1)$. Each of

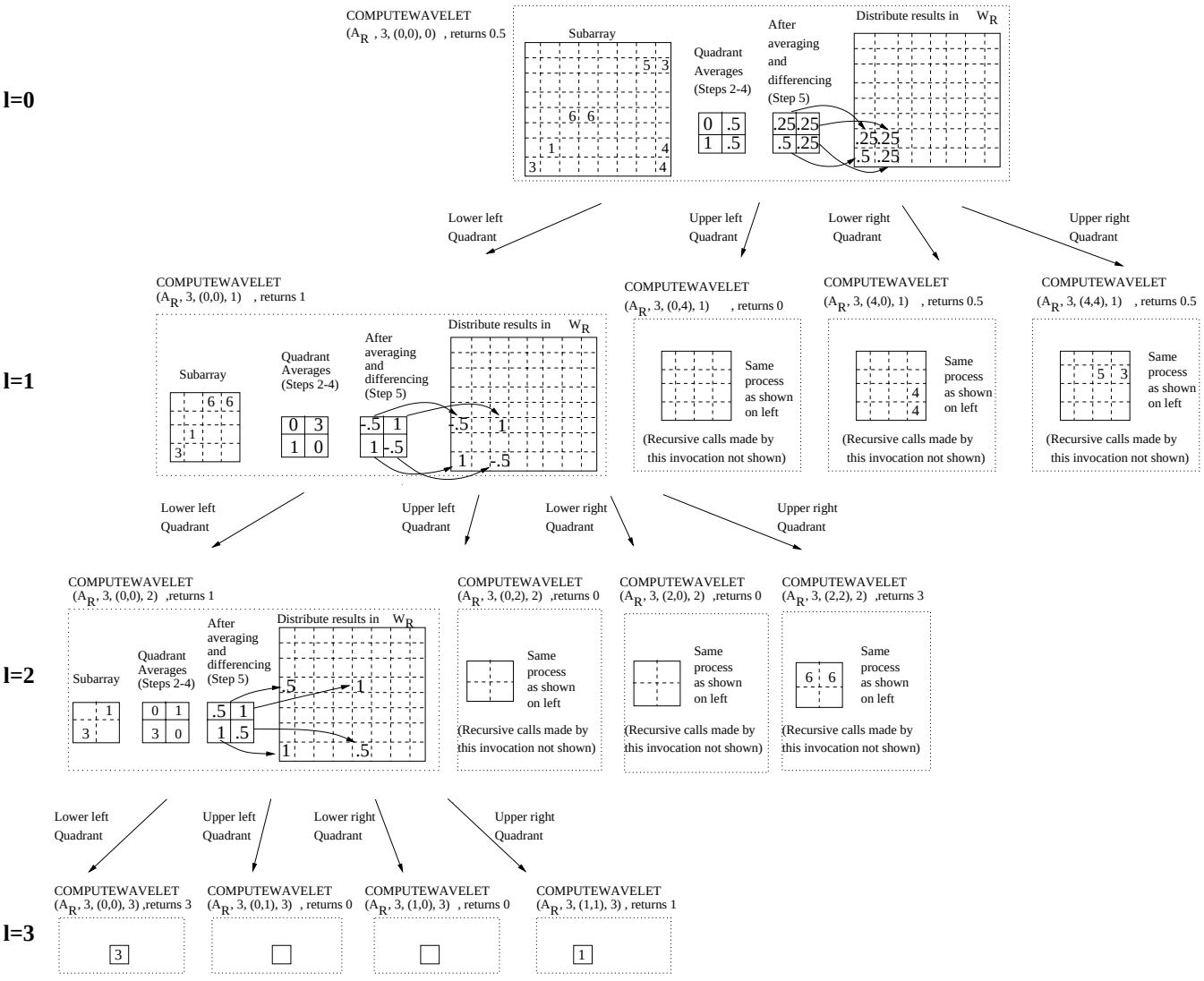


Fig. 5. Execution of the COMPUTEWAVELET algorithm for a 8×8 joint-frequency array. Each invocation of the COMPUTEWAVELET procedure is shown in a dotted box labeled by the procedure call with the right parameters

these four invocations (at resolution $l = 3$) satisfies the "base condition" of the COMPUTEWAVELET recursion (Step 1); thus, each such invocation simply returns the value at its respective 1-cell subarray input (3, 0, 0, and 1, respectively). The caller (COMPUTEWAVELET$(A_R,\ 3,\ (0,0),\ 2)$) collects these four values in the "quadrant-averages array" T, performs pairwise averaging and differencing, distributes the results in the wavelet-transform array W_R, and returns the computed average (i.e., 1) to its caller (the COMPUTEWAVELET $(A_R,\ 3,\ (0,0),\ 1)$ invocation). The other three recursive invocations made by COMPUTEWAVELET$(A_R,\ 3,\ (0,0),\ 1)$ are processed in the same manner. Similarly, COMPUTEWAVELET $(A_R,\ 3,\ (0,0),\ 1)$ simply collects the returned averages (i.e., 1, 0, 0, and 3) in the "quadrant-averages array" T, performs pairwise averaging and differencing, distributes the results in W_R, and returns the computed average (i.e., 1) to its caller (the COMPUTEWAVELET $(A_R,\ 3,\ (0,0),\ 0)$ invocation). The other three recursive invocations made by COMPUTEWAVELET $(A_R,\ 3,\ (0,0),\ 0)$ on the four 4×4 quadrants of A_R are processed in the same manner. As previously, COMPUTEWAVELET$(A_R,\ 3,\ (0,0),\ 0)$ collects the returned averages (i.e., 1, 0, 0.5, and 0.5) in the "quadrant-averages array" T, performs pairwise averaging and differencing, distributes the results in W_R, and returns the overall average of 0.5. □

As we already observed, COMPUTEWAVELET basically exploits chunked array organizations by working in a "depth-first" manner – all the computation required for decomposing an array chunk is done the first time the chunk is loaded into memory. Thus, once a d-dimensional chunk of A_R is loaded into memory, COMPUTEWAVELET computes the (non-standard) wavelet coefficients *at all levels* for that chunk with no additional I/O's. This property essentially guarantees that all computation is completed in a single pass over the chunks of A_R. Thus, assuming a chunked organization of R's tuples, the time complexity of COMPUTEWAVELET is $O(N_z)$[1]; if R

[1] For simplicity, the pseudo-code in Fig. 4 works on the joint-frequency array A_R, which seems to imply a complexity of $O(N)$ for COMPUTEWAVELET. Our implementation, as mentioned before, works on R itself and hence has a time complexity of only $O(N_z)$.

is not chunked, then the complexity of COMPUTEWAVELET is $O(N_z \log N_z)$, due to the extra pre-processing step discussed above.

With respect to the memory requirements of our algorithm, observe that the only extra memory required by COMPUTEWAVELET (in addition to that needed to store the data chunk itself) is the memory for storing the temporary "quadrant-averages hyper-boxes" T. Each such hyper-box consists of exactly 2^d entries and the number of distinct hyper-boxes that can be "active" at any given point in time during the operation of COMPUTEWAVELET is bounded by the depth of the recursion, or equivalently, the number of distinct levels of coefficient resolution. Thus, the extra memory required by COMPUTEWAVELET is at most $O(2^d \cdot m)$ (when $|D_1| = \ldots = |D_d| = 2^m$) or $O(2^d \cdot \log(\max_j\{|D_j|\}))$ (for the general case of unequal dimension extents).

We should note here that both the hyper-rectangle and the sign information for any coefficient generated during the execution of COMPUTEWAVELET over a base relation R can be easily derived from the location of the coefficient in the wavelet transform array W_R, based on the regular recursive structure of the decomposition process. Thus, in order to conserve space, hyper-rectangle boundaries and sign vectors are not explicitly stored in the wavelet-coefficient synopses of base relations. (All that we need are the coefficients' coordinates in W_R.) As we will see later, however, this information does need to be stored explicitly for intermediate collections of wavelet coefficients generated during query processing.

Handling unequal dimension extents. If the sizes of the dimensions of A_R are not equal, then the recursive invocation of COMPUTEWAVELET for quadrant $[t_1, \ldots, t_d]$ (Step 3) takes place only if the inequality $i_j + t_j \cdot 2^{m-l-1} < |D_j|$ is satisfied, for each $j = 1, \ldots, d$. This means that, initially, quadrants along certain "smaller" dimensions are not considered by COMPUTEWAVELET; however, once quadrant sizes become smaller than the dimension size, computation of coefficients in quadrants for such smaller dimensions is initiated. Consequently, the pairwise averaging and differencing computation (Step 5) is performed only along those dimensions that are *"active"* in the current level of the wavelet decomposition. The support hyper-rectangles and dimension sign vectors for such active dimensions are computed as described in Steps 10–12, whereas for an "inactive" dimension j the hyper-rectangle boundaries are set at $boundary[j] := (0, |D_j| - 1)$ (the entire dimension extent) and the sign vector is set at $sign[j] = [+, +]$.

As mentioned in Sect. 2.1, the coefficient values computed by COMPUTEWAVELET need to be properly *normalized* in order to ensure that the Haar basis functions are orthonormal and the coefficients are appropriately weighted according to their importance in reconstructing the original data. This is obviously crucial when thresholding coefficients based on a given (limited) amount of storage space. When all dimensions are of equal extent (i.e., $|D_1| = |D_2| = \ldots = |D_d| = 2^m$), we can normalize coefficient values by simply dividing each coefficient with $\sqrt{2^l}^d$, where l is the level of resolution for the coefficient. As for one-dimensional wavelets, this normalization ensures the orthonormality of the Haar basis [32]. The following lemma shows how to extend the normalization process for nonstandard Haar coefficients to the important case of unequal dimension extents. (The proof follows by a simple verification of the orthonormality property for the constructed coefficients.)

Lemma 4. *Let W be any wavelet coefficient generated by pairwise averaging and differencing during the nonstandard d-dimensional Haar decomposition of $A = |D_1| \times \cdots \times |D_d|$. In addition, let $W.R.length[j] := W.R.boundary[j].hi - W.R.boundary[j].lo + 1$ denote the extent of W along dimension j, for each $1 \le j \le d$. Then, dividing the value $W.v$ of each coefficient W by the factor $\prod_j \sqrt{\frac{|D_j|}{W.R.length[j]}}$ gives an* orthonormal basis. □

Coefficient thresholding. Given a limited amount of storage for maintaining the wavelet-coefficient synopsis of R, we can only retain a certain number C of the coefficients stored in W_R. (The remaining coefficients are implicitly set to 0.) Typically, we have $C << N_z$, which implies that the chosen C wavelet coefficients form a highly compressed approximate representation of the original relational data. The goal of coefficient thresholding is to determine the "best" subset of C coefficients to retain, so that the error in the approximation is minimized.

The thresholding scheme that we have employed for the purposes of this study is to retain the C largest wavelet coefficients in *absolute normalized value*. It is a well-known fact that (for any orthonormal wavelet basis) this thresholding method is in fact *provably optimal* with respect to minimizing the overall mean squared error (i.e., L^2 error norm) in the data compression [32]. Given that our goal in this work is to support effective and accurate *general* query processing over such wavelet-compressed relational tables, we felt that the L^2 error norm would provide a reasonable aggregate metric of the accuracy of the approximation over all the individual tuples of R. Our thresholding approach is also validated by earlier results, where it has been proven that minimizing the L^2 approximation error is in fact optimal (on the average) for estimating the sizes of join query results [15]. Note that it is possible to optimize our COMPUTEWAVELET algorithm for L^2-based coefficient thresholding to ensure that only coefficients that make it into the final synopsis are maintained during the decomposition process; for example, Steps 6–14 can be omitted for coefficients with absolute normalized value less than the C best coefficients found so far. We have chosen not to incorporate such optimizations in our discussion here in order to keep the presentation of COMPUTEWAVELET simple and independent of the specifics of the thresholding scheme. For the remainder of the paper, we use the symbol W_R to denote the set of wavelet coefficients *retained* from the decomposition of relation R (i.e., the *wavelet-coefficient synopsis* of R).

Rendering a wavelet-coefficient synopsis. A crucial requirement for any lossy data-compression scheme is the ability to reconstruct an approximate version of the original data from a given compressed representation. In our context, this requirement translates to *rendering* a given set of wavelet coefficients $W_T = \{W_i = \langle R_i, S_i, v_i \rangle\}$ corresponding to a relational table T, to produce an "approximate version" of T that we denote by `render`(W_T). It is important to note that T can

correspond to either a base relation or the result of an arbitrarily complex SQL query on base relations. As we show in Sect. 3, our approximate query execution engine does the bulk of its processing directly over the wavelet coefficient domain. This means that producing the final approximate query answer in "human-readable" form can always be done by placing a `render()` operator at the root of the query plan or as a post-processing step.

Abstractly, the approximate relation `render`(W_T) can be constructed by summing up the contributions of every coefficient W_i in W_T to the appropriate cells of the (approximate) MOLAP array A_T. Consider a cell in A_T with coordinates $(i_1, \ldots, i_d)$ that is contained in the W_i's support hyper-rectangle $W_i.R$. Then, the contribution of W_i to $A_T[i_1, \ldots, i_d]$ is exactly $W_i.v \cdot \prod_{1 \le j \le d} s_j$, where $s_j = W.S.sign[j].lo$ if $i_j < W.S.signchange[j]$; otherwise, $s_j = W.S.sign[j].hi$. Once the counts for all the cells in the approximate MOLAP array A_T have been computed, the non-zero cells can be used to generate the tuples in the approximate relation `render`(W_T). In Sect. 3.5, we present an efficient algorithm for rendering a set of wavelet coefficients W_T to an approximate MOLAP representation. (The tuple generation step is then trivial.)

3 Processing relational queries in the wavelet-coefficient domain

In this section, we propose a novel query algebra for wavelet-coefficient synopses. The basic operators of our algebra correspond directly to conventional relational algebra and SQL operators, including the (non-aggregate) `select`, `project`, and `join`, as well as aggregate operators like `count`, `sum`, and `average`. There is, however, one crucial difference: our operators are defined *over the wavelet-coefficient domain*; that is, their input(s) and output are *sets of wavelet coefficients* (rather than relational tables). The motivation for defining a query algebra for wavelet coefficients comes directly from the need for efficient approximate query processing. To see this, consider an n-ary relational query Q over $R_1, \ldots, R_n$ and assume that each relation R_i has been reduced to a (truncated) set of wavelet coefficients W_{R_i}. A simplistic way of processing Q would be to render each synopsis W_{R_i} into the corresponding approximate relation (denoted `render`(W_{R_i})) and process the relational operators in Q over the resulting sets of tuples. This strategy, however, is clearly inefficient: the approximate relation `render`(W_{R_i}) may contain just as many tuples as the original R_i itself, which implies that query execution costs may also be just as high as those of the original query. Therefore, such a "render-then-process" strategy essentially defeats one of the main motivations behind approximate query processing.

On the other hand, the synopsis W_{R_i} is a highly-compressed representation of `render`(W_{R_i}) that is typically orders of magnitude smaller than R_i. Executing Q in the compressed wavelet-coefficient domain (essentially, postponing `render`-ing until the final query result) can offer tremendous speedups in query execution cost. We therefore define the operators `op` of our query processing algebra over wavelet-coefficient synopses, while guaranteeing the valid semantics depicted pictorially in the transition diagram of

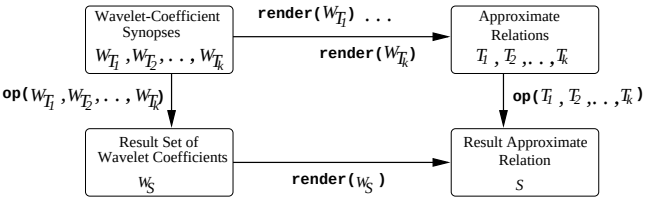


Fig. 6. Valid semantics for processing query operators over the wavelet-coefficient domain

Fig. 6. (These semantics can be translated to the equivalence `render(op`$(T_1, \ldots, T_k)) \equiv$ `op(render`$(T_1, \ldots, T_k)$), for each operator `op`.) Our algebra allows the fast execution of any relational query Q *entirely* over the wavelet-coefficient domain, while guaranteeing that the final (rendered) result is identical to that obtained by executing Q on the approximate input relations.

In the following subsections, we describe our algorithms for processing the SQL operators in the wavelet-coefficient domain. Each operator takes as input one or more set(s) of multi-dimensional wavelet coefficients and appropriately combines and/or updates the components (i.e., hyper-rectangle, sign information, and magnitude) of these coefficients to produce a "valid" set of output coefficients (Fig. 6). Note that, while the wavelet coefficients (generated by COMPUTEWAVELET) for base relational tables have a very regular structure, the same is not necessarily true for the set of coefficients output by an arbitrary `select` or `join` operator. Nevertheless, we loosely continue to refer to the intermediate results of our algebra operators as "wavelet coefficients" since they are characterized by the exact same components as base-relation coefficients (e.g., hyper-rectangle, sign-vectors) and maintain the exact same semantics with respect to the underlying intermediate relation (i.e., the rendering process remains unchanged).

3.1 Selection operator (`select`*)*

Our selection operator has the general form $\texttt{select}_{pred}(W_T)$, where $pred$ represents a generic conjunctive predicate on a subset of the d attributes in T; that is, $pred = (l_{i_1} \le X_{i_1} \le h_{i_1}) \wedge \ldots \wedge (l_{i_k} \le X_{i_k} \le h_{i_k})$, where l_{i_j} and h_{i_j} denote the low and high boundaries of the selected range along each selection dimension D_{i_j}, $j = 1, 2, \cdots, k$, $k \le d$. This is essentially a k-dimensional range selection, where the queried range is specified along k dimensions $\mathcal{D}' = \{D_{i_1}, D_{i_2}, \ldots, D_{i_k}\}$ and left unspecified along the remaining $(d-k)$ dimensions $(\mathcal{D} - \mathcal{D}')$. ($\mathcal{D} = \{D_1, D_2, \ldots, D_d\}$ denotes the set of all dimensions of T.) Thus, for each unspecified dimension D_j, the selection range spans the full index domain along the dimension; that is, $l_j = 0$ and $h_j = |D_j| - 1$, for each $D_j \in (\mathcal{D} - \mathcal{D}')$.

The `select` operator effectively filters out the portions of the wavelet coefficients in the synopsis W_T that do not overlap with the k-dimensional selection range, and thus do not contribute to cells in the selected hyper-rectangle. This process is illustrated pictorially in Fig. 7. More formally, let $W \in W_T$ denote any wavelet coefficient in the input set of our `select` operator. Our approximate query execution engine processes the selection over W as follows. If W's support hyper-rectangle $W.R$ overlaps the k-dimensional selec-

tion hyper-rectangle; that is, if *for every* dimension $D_{i_j} \in \mathcal{D}'$, the following condition is satisfied:

$$l_{i_j} \leq W.R.boundary[i_j].lo \leq h_{i_j} \quad \text{OR}$$
$$W.R.boundary[i_j].lo \leq l_{i_j} \leq W.R.boundary[i_j].hi,$$

then

1. For all dimensions $D_{i_j} \in \mathcal{D}'$ do
   1.1. Set $W.R.boundary[i_j].lo := \max\{ l_{i_j}, W.R.boundary[i_j].lo \}$ and $W.R.boundary[i_j].hi := \min\{h_{i_j}, W.R.boundary[i_j].hi \}$.
   1.2. If $W.R.boundary[i_j].hi < W.S.signchange[i_j]$ then set $W.S.signchange[i_j] := W.R.boundary[i_j].lo$ and $W.S.sign[i_j] := [W.S.sign[i_j].lo, W.S.sign[i_j].lo]$.
   1.3. Else if $W.R.boundary[i_j].lo \geq W.S.signchange[i_j]$ then set $W.S.signchange[i_j] := W.R.boundary[i_j].lo$ and $W.S.sign[i_j] := [W.S.sign[i_j].hi, W.S.sign[i_j].hi]$.
2. Add the (updated) W to the set of output coefficients; that is, set $W_S := W_S \cup \{W\}$, where $S = \texttt{select}_{pred}(T)$.

Our `select` processing algorithm chooses (and appropriately updates) only the coefficients in W_T that overlap with the k-dimensional selection hyper-rectangle. For each such coefficient, our algorithm: (a) updates the hyper-rectangle boundaries according to the specified selection range (Step 1.1); and (b) updates the sign information, if such an update is necessary (Steps 1.2–1.3). Briefly, the sign information along the queried dimension D_{i_j} needs to be updated only if the selection range along D_{i_j} is completely contained in either the low (1.2) or the high (1.3) sign-vector range of the coefficient along D_{i_j}. In both cases, the sign-vector of the coefficient is updated to contain only the single sign present in the selection range and the coefficient's sign-change is set to its leftmost boundary value (since there is no change of sign along D_{i_j} after the selection). The sign-vector and sign-change of the result coefficient remain untouched (i.e., identical to those of the input coefficient) if the selection range spans the original sign-change value.

Example 5. Figure 7a depicts the semantics of a selection operation in the relation domain using an example relation T with two dimensions ($|D_1| = 16, |D_2| = 16$) shown in its joint-frequency array representation A_T. The `select` operator defines a two-dimensional selection hyper-rectangle over T with boundaries $[l_1, h_1] = [4, 13]$ and $[l_2, h_2] = [5, 10]$ along dimensions D_1 and D_2, respectively. The output of the operation consists of only those tuples of T that fall inside the selection hyper-rectangle.

Figure 7b shows the semantics of the same selection operation in the wavelet-coefficient domain. We describe the processing for one of our example coefficients; the others are processed similarly. Consider the wavelet coefficient W_3 having hyper-rectangle ranges $W_3.R.boundary[1] = [9, 15]$ and $W_3.R.boundary[2] = [2, 7]$. The sign information for W_3 is $W_3.S.sign[1] = [+, -]$, $W_3.S.sign[2] = [+, -]$ (Fig. 2b), $W_3.S.signchange[1] = 12$, and $W_3.S.signchange[2] = 4$. Since W_3's hyper-rectangle overlaps with the selection hyper-rectangle, it is processed by the `select` operator as follows. First, in Step 1.1, the hyper-rectangle boundaries of W_3 are updated to $W_3.R.boundary[1] := [9, 13]$ and $W_3.R.boundary[2] := [5, 7]$ (i.e., the region that overlaps with the select ranges along D_1 and D_2). Since $W_3.S.signchange[1] = 12$ which is between 9 and 13 (the new boundaries along D_1), the sign information along D_1 is not updated. Along dimension D_2, however, we have $W_3.S.signchange[2] = 4$ which is less than $W_3.R.boundary[2].lo = 5$, and so Step 1.3 updates the sign information along D_2 to $W_3.S.sign[2] := [-, -]$ and $W_3.S.signchange[2] := 5$ (i.e., the low boundary along D_2). □

3.2 Projection operator (`project`)

Our projection operator has the form $\texttt{project}_{X_{i_1},\ldots,X_{i_k}}(W_T)$, where the k projection attributes $X_{i_1}, \ldots, X_{i_k}$ form a subset of the d attributes of T. Letting $\mathcal{D}' = \{D_{i_1}, \ldots, D_{i_k}\}$ denote the $k \leq d$ projection dimensions, we are interested in *projecting out* the $d - k$ dimensions in $(\mathcal{D} - \mathcal{D}')$. We give a general method for projecting out a single dimension $D_j \in \mathcal{D} - D'$. This method can then be applied repeatedly to project out all the dimensions in $(\mathcal{D} - \mathcal{D}')$, one dimension at a time.

Consider T's corresponding multi-dimensional joint-frequency array A_T. Projecting a dimension D_j out of A_T is equivalent to summing up the counts for all the array cells in each one-dimensional row of A_T along dimension D_j and then assigning this aggregated count to the single cell corresponding to that row in the remaining dimensions $(\mathcal{D} - \{D_j\})$. The above process is illustrated with an example two-dimensional array A_T in Fig. 8a. Consider any d-dimensional wavelet coefficient W in the `project` operator's input set W_T. Remember that W contributes a value of $W.v$ to every cell in its support hyper-rectangle $W.R$. Furthermore, the sign of this contribution for every one-dimensional row along dimension D_j is determined as either $W.S.sign[j].hi$ (if the cell lies above $W.S.signchange[j]$) or $W.S.sign[j].lo$ (otherwise). Thus, we can work directly on the coefficient W to project out dimension D_j by simply adjusting the coefficient's magnitude with an appropriate multiplicative constant $W.v := W.v * p_j$, where p_j is defined as (we omit the "$W.$" prefix for clarity):

$$(R.boundary[j].hi - S.signchange[j] + 1) * S.sign[j].hi + (S.signchange[j] - R.boundary[j].lo) * S.sign[j].lo. \quad (1)$$

A two-dimensional example of projecting out a dimension in the wavelet-coefficient domain is depicted in Fig. 8b. Multiplying $W.v$ with p_j (Eq. (1)) effectively projects out dimension D_j from W by summing up W's contribution on each one-dimensional row along dimension D_j. Of course, besides adjusting $W.v$, we also need to discard dimension D_j from the hyper-rectangle and sign information for W, since it is now a $(d-1)$-dimensional coefficient (on dimensions $\mathcal{D} - \{D_j\}$). Note that if the coefficient's sign-change lies in the middle of its support range along dimension D_j (e.g., see Fig. 2a), then its adjusted magnitude will be 0, which means that it can safely be discarded from the output set of the projection operation.

Repeating the above process for each wavelet coefficient $W \in W_T$ and each dimension $D_j \in \mathcal{D} - D'$ gives the set of output wavelet coefficients W_S, where $S = \texttt{project}_{\mathcal{D}'}(T)$. Equivalently, given a coefficient W, we can simply set $W.v := W.v * \prod_{D_j \in \mathcal{D} - D'} p_j$ (where p_j is as defined in Eq. (1)) and discard dimensions $\mathcal{D} - D'$ from W's representation.

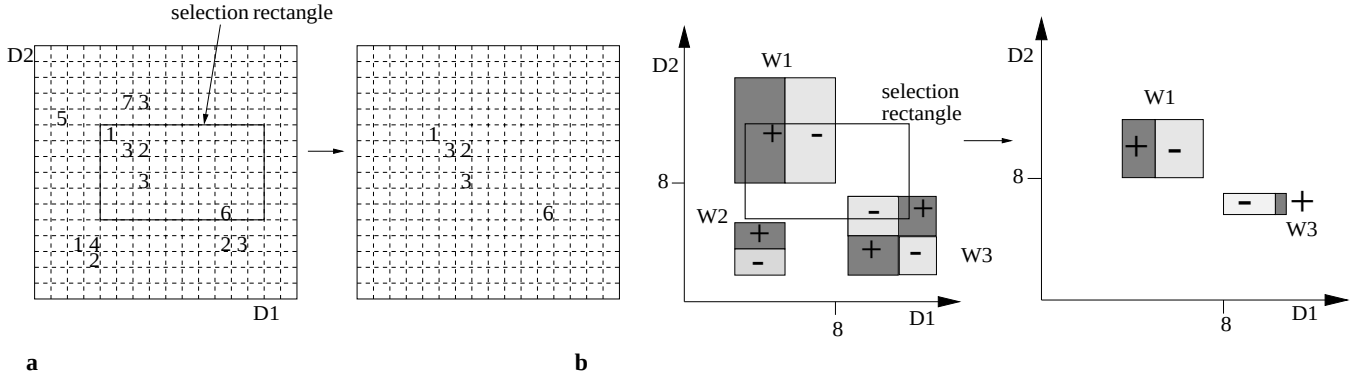


Fig. 7. a Processing a selection operation in the relation domain; **b** processing a selection operation in the wavelet-coefficient domain

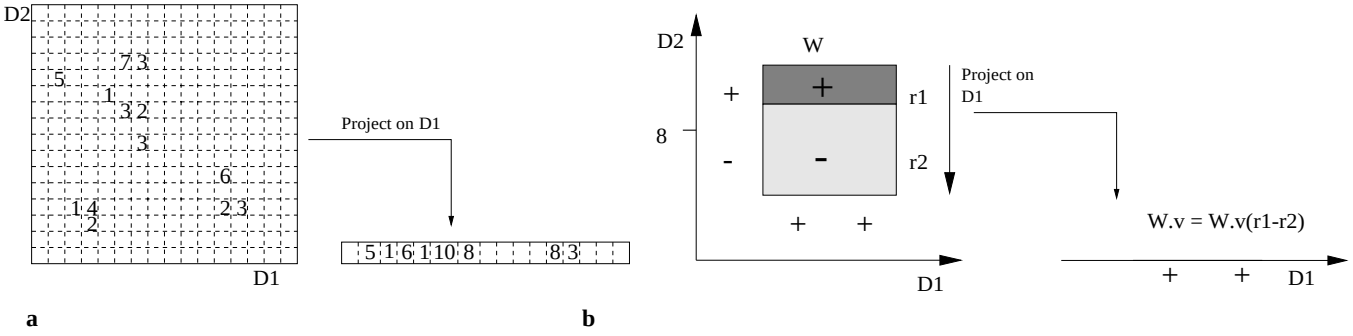


Fig. 8. a Processing a projection operation in the relation domain; **b** processing a projection operation in the wavelet-coefficient domain

Example 6. Figure 8a depicts the semantics of a projection operation in the relation domain, showing the two-dimensional relation T ($|D_1| = 16, |D_2| = 16$) from Example 5 and the result of its projection on dimension D_1.

Figure 8b illustrates the semantics of the same projection operation in the wavelet-coefficient domain. Consider the wavelet coefficient W whose hyper-rectangle and sign information along dimension D_2 are as follows: $W.R.boundary[2] = [4, 11]$, $W.S.sign[2] = [-, +]$, and $W.S.signchange[2] = 10$. In addition, let the magnitude of W be $W.v = 2$. Then, projecting W on dimension D_1 causes $W.v$ to be updated to $W.v := 2 \cdot ((11 - 10 + 1) - (10 - 4)) = -8$. □

3.3 Join operator (`join`)

Our join operator has the general form $\texttt{join}_{pred}(W_{T_1}, W_{T_2})$, where T_1 and T_2 are (approximate) relations of arity d_1 and d_2, respectively, and $pred$ is a conjunctive k-ary equi-join predicate of the form $(X_1^1 = X_1^2) \wedge \ldots \wedge (X_k^1 = X_k^2)$, where X_j^i (D_j^i) ($j = 1, \ldots, d_i$) denotes the j^{th} attribute (respectively, dimension) of T_i ($i = 1, 2$). (Without loss of generality, we assume that the join attributes are the first $k \leq \min\{d_1, d_2\}$ attributes of each joining relation.) Note that the result of the join operation W_S is a set of $(d_1 + d_2 - k)$-dimensional wavelet coefficients; that is, the join operation returns coefficients of (possibly) different arity than any of its inputs.

To see how our join processing algorithm works, consider the multi-dimensional arrays A_{T_1} and A_{T_2} corresponding to the join operator's input arguments. Let $(i_1^1, \ldots, i_{d_1}^1)$ and $(i_1^2, \ldots, i_{d_2}^2)$ denote the coordinates of two cells belonging to A_{T_1} and A_{T_2}, respectively. If the indexes of the two cells match on the join dimensions, i.e., $i_1^1 = i_1^2, \ldots, i_k^1 = i_k^2$, then the cell in the join result array A_S with coordinates $(i_1^1, \ldots, i_{d_1}^1, i_{k+1}^2, \ldots, i_{d_2}^2)$ is populated with the *product* of the count values contained in the two joined cells. Figure 9a illustrates the above process with two example two-dimensional joint-frequency arrays A_{T_1} (with dimensions D_1 and D_2, $|D_1| = |D_2| = 16$) and A_{T_2} (with dimensions D_1 and D_3, $|D_1| = |D_3| = 16$) and join dimension D_1 (shown vertically for both arrays). For example, the cells $(9, 6)$ in A_{T_1} (count value 2) and $(9, 2)$ in A_{T_2} (count value 6) match on the join dimension D_1 (both have a D_1 coordinate of 9); hence, the join output is populated with the cell $(9, 6, 2)$ (count value = $2 * 6 = 12$). Since the cell counts for A_{T_i} are derived by appropriately summing the contributions of the wavelet coefficients in W_{T_i} and, of course, a numeric product can always be distributed over summation, we can process the `join` operator entirely in the wavelet-coefficient domain by considering all pairs of coefficients from W_{T_1} and W_{T_2}. Briefly, for any two coefficients from W_{T_1} and W_{T_2} that overlap in the join dimensions and, therefore, contribute to joining data cells, we define an output coefficient with magnitude equal to the product of the two joining coefficients and a support hyper-rectangle with ranges that are: (a) equal to the overlap of the two coefficients for the k (common) join dimensions; and (b) equal to the original coefficient ranges along any of the $d_1 + d_2 - 2k$ remaining dimensions. The sign information for an output coefficient along any of the k join dimensions is derived by appropriately multiplying the sign-vectors of the joining coefficients along that dimension, taking care to ensure that only signs along the overlapping portion are taken into account. (The sign information along non-join dimensions remains unchanged.) An example

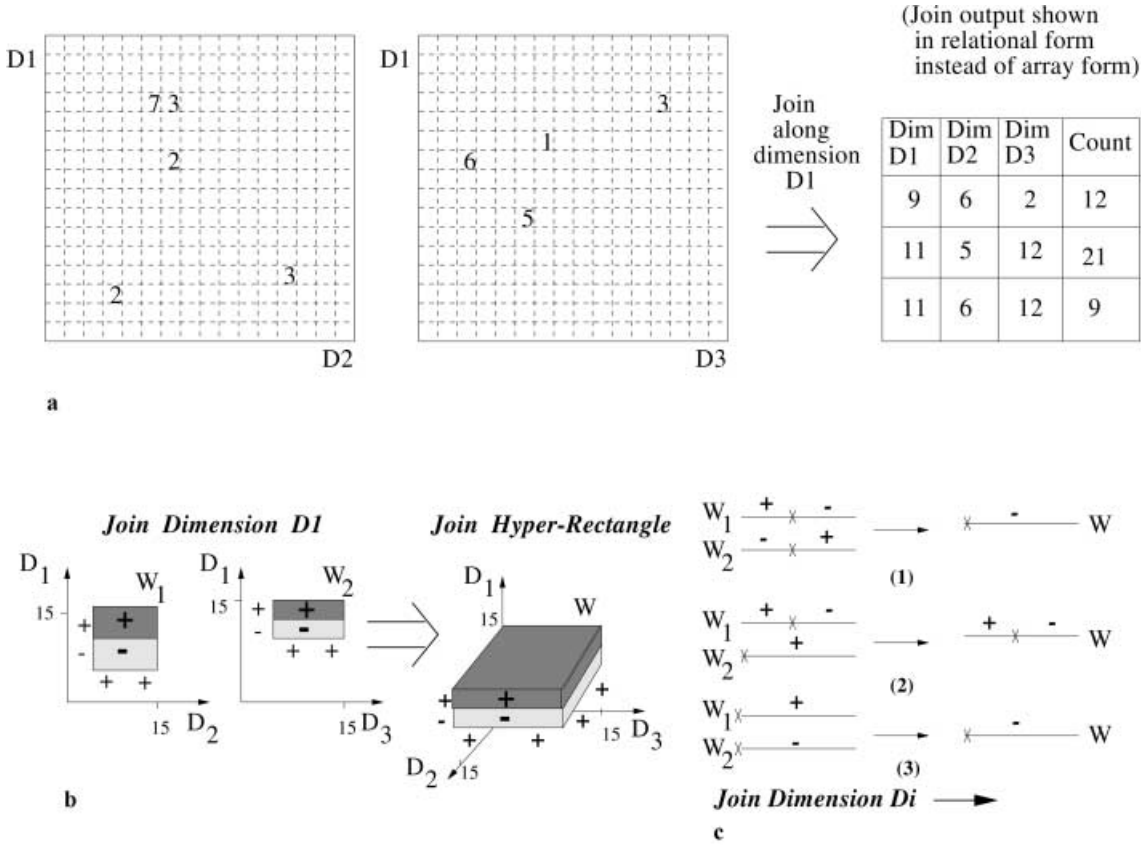


Fig. 9. a Processing a `join` operation in the relation domain; **b** processing a `join` operation in the wavelet-coefficient domain; **c** computing sign information for `join` output coefficients

of this process in two dimensions ($d_1 = d_2 = 2$, $k = 1$) is depicted in Fig. 9b.

More formally, our approximate query execution strategy for joins can be described as follows. (To simplify the notation, we ignore the "1/2" superscripts and denote the join dimensions as $D_1, \ldots, D_k$, and the remaining $d_1 + d_2 - 2k$ dimensions as $D_{k+1}, \ldots, D_{d_1+d_2-k}$.) For each pair of wavelet coefficients $W_1 \in W_{T_1}$ and $W_2 \in W_{T_2}$, if the coefficients' support hyper-rectangles overlap in the k join dimensions; that is, if *for every* dimension D_i, $i = 1 \ldots, k$, the following condition is satisfied:

$$(W_1.R.boundary.lo[i] \leq W_2.R.boundary.lo[i] \quad \text{AND} \quad W_2.R.boundary.lo[i] \leq W_1.R.boundary.hi[i])$$
$$\text{OR}$$
$$(W_2.R.boundary.lo[i] \leq W_1.R.boundary.lo[i] \quad \text{AND} \quad W_1.R.boundary.lo[i] \leq W_2.R.boundary.hi[i]),$$

then the corresponding output coefficient $W \in W_S$ is defined in the following steps.

1. For all join dimensions D_i, $i = 1, \ldots, k$ do
   1.1. Set $W.R.boundary[i].lo := \max\{ W_1.R.boundary[i].lo, W_2.R.boundary[i].lo\}$ and $W.R.boundary[i].hi := \min\{ W_1.R.boundary[i].hi, W_2.R.boundary[i].hi\}$.
   1.2. For $j = 1, 2$ /* s_j is a temporary sign-vector variable */
      1.2.1. If $W.R.boundary[i].hi < W_j.S.signchange[i]$ then set $s_j := [W_j.S.sign[i].lo, W_j.S.sign[i].lo]$.
      1.2.2. Else if $W.R.boundary[i].lo \geq W_j.S.signchange[i]$ then set $s_j := [W_j.S.sign[i].hi, W_j.S.sign[i].hi]$.
      1.2.3. Else set $s_j := W_j.S.sign[i]$.
   1.3. Set $W.S.sign[i] := [s_1.lo * s_2.lo,\ s_1.hi * s_2.hi]$.
   1.4. If $W.S.sign[i].lo == W.S.sign[i].hi$ then set $W.S.signchange[i] := W.R.boundary[i].lo$.
   1.5 Else set $W.S.signchange[i] := \max_{j=1,2}\{ W_j.S.signchange[i] : W_j.S.signchange[i] \in [W.R.boundary[i].lo, W.R.boundary[i].hi] \}$.
2. For each (non-join) dimension D_i, $i = k + 1, \ldots, d_1$ do: Set $W.R.boundary[i] := W_1.R.boundary[i]$, $W.S.sign[i] := W_1.S.sign[i]$, and $W.S.signchange[i] := W_1.S.signchange[i]$.
3. For each (non-join) dimension D_i, $i = d_1 + 1, \ldots, d_1 + d_2 - k$ do: Set $W.R.boundary[i] := W_2.R.boundary[i - d_1 + k]$, $W.S.sign[i] := W_2.S.sign[i - d_1 + k]$, and $W.S.signchange[i] := W_2.S.signchange[i - d_1 + k]$.
4. Set $W.v := W_1.v * W_2.v$ and $W_S := W_S \cup \{W\}$, where $S = \texttt{join}_{pred}(T_1, T_2)$.

Note that the bulk of our join processing algorithm concentrates on the correct settings for the output coefficient W along the k join dimensions (Step 1), since the problem becomes trivial for the $d_1 + d_2 - 2k$ remaining dimensions (Steps 2–3). Given a pair of joining input coefficients and a join dimension D_i, our algorithm starts out by setting the hyper-rectangle range of the output coefficient W along D_i equal to the overlap of the two input coefficients along D_i (Step 1.1). We then proceed to compute W's sign information along join dimension D_i (Steps 1.2–1.3) , which is slightly more involved.

(Remember that T_1 and T_2 are (possibly) the results of earlier `select` and/or `join` operators, which means that their rectangle boundaries and signs along D_i can be arbitrary.) The basic idea is to determine, for each of the two input coefficients W_1 and W_2, where the boundaries of the join range lie with respect to the coefficient's sign-change value along dimension D_i. Given an input coefficient W_j ($j = 1, 2$), if the join range along D_i is completely contained in either the low (1.2.1) or the high (1.2.2) sign-vector range of W_j along D_i, then a temporary sign-vector s_j is appropriately set (with the same sign in both entries). Otherwise, i.e., if the join range spans W_j's sign-change (1.2.3), then s_j is simply set to W_j's sign-vector along D_i. Thus, s_j captures the sign of coefficient W_j in the joining range, and multiplying s_1 and s_2 (element-wise) yields the sign-vector for the output coefficient W along dimension D_i (Step 1.3). If the resulting sign vector for W does not contain a true sign change (i.e., the low and high components of $W.S.sign[i]$ are the same), then W's sign-change value along dimension D_i is set equal to the low boundary of $W.R$ along D_i, according to our convention (Step 1.4). Otherwise, the sign-change value for the output coefficient W along D_i is set equal to the maximum of the input coefficients' sign-change values that are contained in the join range (i.e., $W.R$'s boundaries) along D_i (Step 1.5).

In Fig. 9c, we illustrate three common scenarios for the computation of W's sign information along the join dimension D_i. The left-hand side of the figure shows three possibilities for the sign information of the input coefficients W_1 and W_2 along the join range of dimension D_i (with crosses denoting sign changes). The right-hand side depicts the resulting sign information for the output coefficient W along the same range. The important thing to observe with respect to our sign-information computation in Steps 1.3–1.5 is that the join range along any join dimension D_i can contain *at most one* true sign change. By this, we mean that if the sign for input coefficient W_j actually changes in the join range along D_i, then this sign-change value is unique; that is, the two input coefficients cannot have true sign changes at distinct points of the join range. This follows from the *complete containment* property of the base coefficient ranges along dimension D_i (Sect. 2.1). (Note that our algorithm for `select` retains the value of a true sign change for a base coefficient if it is contained in the selection range, and sets it equal to the value of the left boundary otherwise.) This range containment along D_i ensures that if W_1 and W_2 both contain a true sign change in the join range (i.e., their overlap) along D_i, then that will occur *at exactly the same value* for both (as illustrated in Fig. 9(c.1)). Thus, in Step 1.3, W_1's and W_2's sign vectors in the join range can be multiplied to derive W's sign-vector. If, on the other hand, one of W_1 and W_2 has a true sign change in the join range (as shown in Fig. 9(c.2)), then the $\max$ operation of Step 1.5 will always set the sign change of W along D_i correctly to the true sign-change value (since the other sign change will either be at the left boundary or outside the join range). Finally, if neither W_1 nor W_2 have a true sign change in the join range, then the high and low components of W's sign vector will be identical and Step 1.4 will set W's sign-change value correctly.

Example 7. Figure 9a depicts the semantics of a join operation in the relation domain as described above. Figure 9b illustrates the semantics of the same operation in the wavelet-coefficient domain. Consider the wavelet coefficients W_1 and W_2. Let the boundaries and sign information of W_1 and W_2 along the join dimension D_1 be as follows: $W_1.R.boundary[1] = [4, 15]$, $W_2.R.boundary[1] = [8, 15]$, $W_1.S.sign[1] = [-, +]$, $W_2.S.sign[1] = [-, +]$, $W_1.S.signchange[1] = 8$, and $W_2.S.signchange[1] = 12$. In the following, we illustrate the computation of the hyper-rectangle and sign information for join dimension D_1 for the coefficient W that is output by our algorithm when W_1 and W_2 are "joined". Note that for the non-join dimensions D_2 and D_3, this information for W is identical to that of W_1 and W_2 (respectively), so we focus solely on the join dimension D_1.

First, in Step 1.1, $W.R.boundary[1]$ is set to $[8, 15]$, i.e., the overlap range between W_1 and W_2 along D_1. In Step 1.2.2, since $W.R.boundary[1].lo = 8$ is greater than or equal to $W_1.S.signchange[1] = 8$, we set $s_1 = [+, +]$. In Step 1.2.3, since $W_2.S.signchange[1] = 12$ lies in between $W.R$'s boundaries, we set $s_2 = [-, +]$. Thus, in Step 1.3, $W.S.sign[1]$ is set to the product of s_1 and s_2 which is $[-, +]$. Finally, in Step 1.5, $W.S.signchange[1]$ is set to the maximum of the sign change values for W_1 and W_2 along dimension D_1, or $W.S.signchange[1] := \max\{8, 12\} = 12$. □

3.4 Aggregate operators

In this section, we show how conventional aggregation operators, like `count`, `sum`, and `average`, are realized by our approximate query execution engine in the wavelet-coefficient domain[2]. As before, the input to each aggregate operator is a set of wavelet coefficients W_T. If the aggregation is not qualified with a `GROUP-BY` clause, then the output of the operator is a simple scalar value for the aggregate. In the more general case, where a `GROUP-BY` clause over dimensions $\mathcal{D}' = \{D_1, \ldots, D_k\}$ has been specified, the output of the aggregate operator consists of a k-dimensional array spanning the dimensions in $\mathcal{D}'$, whose entries contain the computed aggregate value for each cell.

Note that, unlike our earlier query operators, we define our aggregate operators to provide output that is essentially a rendered data array, rather than a set of (un-rendered) wavelet coefficients. This is because there is no clean, general method to map the computed aggregate values (e.g., attribute sums or averages) onto the semantics and structure of wavelet coefficients. We believe, however, that exiting the coefficient domain after aggregation has no negative implications for the effectiveness of our query execution algorithms. The reason is that, for most DSS queries containing aggregation, the aggregate operator is the final operator at the root of the query execution plan, which means that its result would have to be rendered anyway.

While the earlier work of Vitter and Wang [33] has addressed the computation of aggregates over a wavelet-compressed relational table, their approach is significantly

[2] Like most conventional data reduction and approximate querying techniques (e.g., sampling and histograms), wavelets are inherently limited to "trivial answers" when it comes to `min` or `max` aggregate functions (see, for example, [16]). In our case, this would amount to selecting the *non-zero* cell in the reconstructed array with minimum/maximum coordinate along the specified query range. We do not consider `min` or `max` aggregates further in this paper.

different from ours. Vitter and Wang focus on a very specific form of aggregate queries, namely *range-sum queries*, where the range(s) are specified over one or more *functional attribute* and the summation is done over a prespecified *measure attribute*. Their wavelet decomposition and aggregation algorithm are both geared towards this specific type of queries that essentially treats the relation's attributes in an "asymmetric" manner (by distinguishing the single measure attribute). Our approach, on the other hand, has a much broader query processing scope. As a result, all attributes are treated in a completely symmetric fashion, thus enabling us to perform a broad range of aggregate (and non-aggregate) operations over *any* attribute(s).

Count operator (`count`*).* Our count operator has the general form $\texttt{count}_{\mathcal{D}'}(W_T)$, where the k `GROUP-BY` dimensions $\mathcal{D}' = \{D_{i_1}, \ldots, D_{i_k}\}$ form a (possibly empty) subset of the d attributes of T. Counting is the most straightforward aggregate operation to implement in our framework, since each cell in our approximate multi-dimensional array already stores the count information for that cell. Thus, processing $\texttt{count}_{\mathcal{D}'}(W_T)$ is done by simply projecting each input coefficient onto the `GROUP-BY` dimensions $\mathcal{D}'$ and rendering the result into a multi-dimensional array of counts, as follows.

1. Let $W_S := \texttt{project}_{\mathcal{D}'}(W_T)$ (see Sect. 3.2).
2. Let $A_S := \texttt{render}(W_S)$ and output the cells in the $|\mathcal{D}'|$-dimensional array A_S with non-zero counts.

Sum operator (`sum`*).* The general form of our summation operator is $\texttt{sum}_{\mathcal{D}'}(W_T, D_j)$, where $\mathcal{D}' = \{D_{i_1}, \ldots, D_{i_k}\}$ denotes the set of `GROUP-BY` dimensions and $D_j \notin \mathcal{D}'$ corresponds to the attribute of T whose values are summed. The `sum` operator is implemented in three steps. First, we project the input coefficients W_T on dimensions $\mathcal{D}' \cup \{D_j\}$. Second, for each coefficient W output by the first step and for each row of cells along the summation attribute D_j, we compute the sum of the product of the coefficient's magnitude $W.v$ and the index of the cell along D_j [3]. This sum (essentially, an *integral* along D_j) is then assigned to the coefficient's magnitude $W.v$ and the summing dimension D_j is discarded. Thus, at the end of this step, $W.v$ stores the contribution of W to the summation value for every $|\mathcal{D}'|$-dimensional cell in $W.R$. Third, the resulting set of wavelet coefficients is rendered to produce the output multi-dimensional array on dimensions $\mathcal{D}'$. More formally, our $\texttt{sum}_{\mathcal{D}'}(W_T, D_j)$ query processing algorithm comprises the following steps.

1. Let $W_S := \texttt{project}_{\mathcal{D}' \cup \{D_j\}}(W_T)$ (Sect. 3.2).
2. For each wavelet coefficient $W \in W_S$ do
   2.1. Set $W.v$ according to the following equation:

$$W.v := W.v * \left(W.S.sign[j].lo * \sum_{k=W.R.boundary[j].lo}^{W.S.signchange[j]-1} k + W.S.sign[j].hi * \sum_{k=W.S.signchange[j]}^{W.R.boundary[j].hi} k \right).$$

   Note that, the summations of the index values along D_j in the above formula can be expressed in closed form using straightforward algebraic methods.
   2.2. Discard dimension D_j from the hyper-rectangle and sign information for W.
3. Let $A_S := \texttt{render}(W_S)$ and output the cells in the $|\mathcal{D}'|$-dimensional array A_S with non-zero values for the summation.

Average operator (`average`*).* The averaging operator $\texttt{average}_{\mathcal{D}'}(W_T, D_j)$ (where $\mathcal{D}'$ is the set of `GROUP-BY` dimensions and $D_j \notin \mathcal{D}'$ corresponds to the averaged attribute of T) is implemented by combining the computation of $\texttt{sum}_{\mathcal{D}'}(W_T, D_j)$ and $\texttt{count}_{\mathcal{D}'}(W_T)$. The idea is to compute the attribute sums and tuple counts for every cell over the data dimensions in the `GROUP-BY` attributes $\mathcal{D}'$, as described earlier in this section. We then render the resulting coefficients and output the average value (i.e., the ratio of sum over count) for every cell with non-zero sum and count.

3.5 Rendering a set of wavelet coefficients

Since our approximate query execution engine does the bulk of its processing in the wavelet coefficient domain, an essential final step for every user query is to *render* an output set W_S of d-dimensional wavelet coefficients (over, say, $\mathcal{D} = \{D_1, \ldots, D_d\}$) to produce the approximate query answer in a "human-readable" form. (Note that rendering is required as a final step even for the aggregate processing algorithms described in the previous section.) The main challenge in the rendering step is how to *efficiently* expand the input set of d-dimensional wavelet coefficients W_S into the corresponding (approximate) d-dimensional array of counts A_S.

A naive approach to rendering W_S would be to simply consider each cell in the multi-dimensional array A_S and sum the contributions of every coefficient $W \in W_S$ to that cell in order to obtain the corresponding tuple count. However, the number of cells in A_S is potentially huge, which implies that such a naive rendering algorithm could be extremely inefficient and computationally expensive (typically, of order $O(N \cdot |W_S|)$, where $N = \prod_{i=1}^{d} |D_i|$ is the number of array cells). Instead of following this naive and expensive strategy, we propose a more efficient algorithm (termed `render`) for rendering an input set of multi-dimensional wavelet coefficients. (Note that `render` can be seen either as a (final) query processing operator or as a post-processing step for the query.) Our algorithm exploits the fact that the number of coefficients in W_S is typically *much smaller* than the number of array cells N. This implies that we can expect A_S to consist of large, contiguous multi-dimensional regions, where all the cells in each region contain exactly the same count. (In fact, because of the sparsity of the data, many of these regions will have counts of 0.) Furthermore, the total number of such "uniform-count" regions in A_S is typically considerably smaller that N. Thus, the basic idea of our efficient rendering algorithm is to partition the multi-dimensional array A_S, one dimension at a time, into such uniform-count data regions and output the (single) count value corresponding to each such region (the same for all enclosed cells).

Our `render` algorithm (depicted in Fig. 10) recursively partitions the d-dimensional data array A_S, one dimension

[3] To simplify the exposition, we assume that the (integer) cell index values along dimension D_j are identical to the domain values for the corresponding attribute X_j of T. If that is not the case, then a reverse mapping from the D_j index values to the corresponding values of X_j is needed to sum the attribute values along the boundaries of a coefficient.

```
procedure render(COEFF, i)
begin
1.  if (i > d) {
2.      count := 0
3.      for each coefficient W in COEFF
4.          sign := ∏_{D_j ∈ 𝒟} sign_j
        /* sign_j := W.S.sign[j].lo  if  W.R.boundary[j].lo <
           W.S.signchange[j]; else, sign_j := W.S.sign[j].hi  */
5.          count := count + sign * W.v
6.      end
7.      output (W.R.boundary, count)
            /* W is any coefficient in COEFF  */
8.      return
9.  }
10. Q := ∅          /* elements e in priority queue Q are sorted
                       in increasing order of e.key  */
11. for each coefficient W in COEFF
12.     insert element e into Q where e.key :=
        W.R.boundary[i].lo − 1 and e.val := W
13.     insert element e into Q where e.key :=
        W.R.boundary[i].hi and e.val := W
14.     if (W.R.boundary[i].lo < W.S.signchange[i] ≤
                                 W.R.boundary[i].hi)
15.         insert element e into Q where e.key :=
            W.S.signchange[i] − 1 and e.val := W
16. end
17. prev := −∞, TEMP1 := ∅
18. while (Q is not empty) do
19.     TEMP2 := ∅, topkey := e.key for element e at head of Q
20.     dequeue all elements e with e.key = topkey at the head
        of Q and insert e.val into TEMP1
21.     for each coefficient W in TEMP1
22.         delete W from TEMP1 if W.R.boundary[i].hi < prev + 1
23.         if ( W.R.boundary[i] overlaps with the interval
                        [prev + 1, topkey] along dimension D_i ) {
24.             W' := W, W'.R.boundary[i].lo := prev + 1
                W'.R.boundary[i].hi := topkey
25.             insert W' into TEMP2
26.         }
27.     end
28.     render(TEMP2, i + 1)
29.     prev := topkey
30. end  /* while */
end
```

Fig. 10. `render`: an efficient algorithm for rendering multi-dimensional wavelet coefficients

at a time and in the dimension order $D_1, \ldots, D_d$. Algorithm `render` takes two input arguments: (a) the index (i) of the next dimension D_i along which the array A_S is to be partitioned; and (b) the set of wavelet coefficients (COEFF) in the currently processed partition of A_S (generated by the earlier partitionings along dimensions $D_1, \ldots, D_{i-1}$). The initial invocation of `render` is done with $i = 1$ and COEFF $= W_S$.

When partitioning A_S into uniform-count ranges along dimension D_i, the only points that should be considered are those where the cell counts along D_i could potentially change. These are precisely the points where a new coefficient W starts contributing ($W.R.boundary[i].lo$), stops contributing ($W.R.boundary[i].hi$), or the sign of its contribution changes ($W.S.signchange[i]$). Algorithm `render` identifies these points along dimension D_i for each coefficient in COEFF

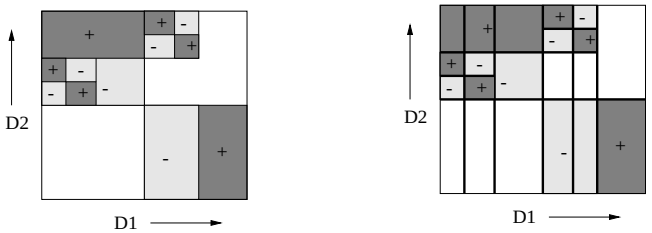


Fig. 11. Partitioning a two-dimensional array by procedure `render`

and stores them in sorted order in a priority queue Q (Steps 11–16). Note that, for any pair of consecutive partitioning points along D_i, the contribution of *each* coefficient in COEFF (and, therefore, their sum) is guaranteed to be *constant* for any row of cells along D_i between the two points. Thus, abstractly, our partitioning generates one-dimensional uniform-count ranges along D_i. Once the partitioning points along dimension D_i have been determined, they are used to partition the hyper-rectangles of the wavelet coefficients in COEFF along D_i (Steps 18–30). Algorithm `render` is then recursively invoked with the set of (partial) coefficients in each partition of D_i to further partition the coefficients along the remaining dimensions $D_{i+1}, \ldots, D_d$. Once the array has been partitioned along all dimensions in $\mathcal{D}$ (i.e., `render` is invoked with parameter $i > d$), a coefficient W in the input set of coefficients COEFF is guaranteed to have a constant contribution to every cell in the corresponding d-dimensional partition. This essentially means that we have discovered a d-dimensional uniform-count partition in A_S, and we can output the partition boundaries and the corresponding tuple count (Steps 2–8).

Figure 11b depicts the partitioning of a two-dimensional data array generated by `render` for the input set consisting of the four wavelet coefficients shown in Fig. 11a. The time complexity of our `render` algorithm can be shown to be $O(|W_S| \cdot P)$, where P is the number of uniform-count partitions in A_S. As we have already observed, P is typically much smaller than the number of array cells N. In addition, note that `render` requires only $O(|W_S| \cdot d)$ of memory, since it only needs to keep track of the coefficients in the partition currently being processed for each dimension.

4 Experimental study

In this section, we present the results of an extensive empirical study that we have conducted using the novel query processing tools developed in this paper. The objective of this study is twofold: (1) to establish the effectiveness of our wavelet-based approach to approximate query processing; and (2) to demonstrate the benefits of our methodology compared to earlier approaches based on sampling and histograms. Our experiments consider a wide range of queries executed on both synthetic and real-life data sets. The major findings of our study can be summarized as follows.

- **Improved answer quality.** The quality/accuracy of the approximate answers obtained from our wavelet-based query processor is, in general, better than that obtained by either sampling or histograms for a wide range of data sets and `select`, `project`, `join`, and aggregate queries.

• **Low synopsis construction costs.** Our I/O-efficient wavelet decomposition algorithm is extremely fast and scales linearly with the size of the data (i.e., the number of cells in the MOLAP array). In contrast, histogram construction costs increase explosively with the dimensionality of the data.

• **Fast query execution.** Query execution-time speedups of more than two orders of magnitude are made possible by our approximate query processing algorithms. Furthermore, our query execution times are competitive with those obtained by the histogram-based methods of Ioannidis and Poosala [16], and sometimes significantly faster (e.g., for `join`s).

Thus, our experimental results validate the thesis of this paper that wavelets are a viable, effective tool for general-purpose approximate query processing in DSS environments. All experiments reported in this section were performed on a Sun Ultra-2/200 machine with 512 MB of main memory, running Solaris 2.5.

4.1 Experimental testbed and methodology

Techniques. We consider three approximate query answering techniques in our study.

• *Sampling.* A random sample of the non-zero cells in the multi-dimensional array representation for each base relation is selected , and the counts for the cells are appropriately scaled. Thus, if the total count of all cells in the array is t and the sum of the counts of cells in the sample is s, then the count of every cell in the sample is multiplied by $\frac{t}{s}$. These scaled counts give the tuple counts for the corresponding approximate relation.

• *Histograms.* Each base relation is approximated by a multi-dimensional MaxDiff(V,A) histogram. Our choice of this histogram class is motivated by the recent work of Ioannidis and Poosala [16], where it is shown that MaxDiff(V,A) histograms result in higher-quality approximate query answers compared to other histogram classes (e.g., EquiDepth or EquiWidth). We process `select`s, `join`s, and aggregate operators on histograms as described in [16]. For instance, while `select`s are applied directly to the histogram for a relation, a `join` between two relations is done by first partially expanding their histograms to generate the tuple-value distribution of the each relation. An indexed nested-loop `join` is then performed on the resulting tuples.

• *Wavelets.* Wavelet-coefficient synopses are constructed on the base relations (using algorithm COMPUTEWAVELET) and query processing is performed entirely in the wavelet-coefficient domain, as described in Sect. 3. In our `join` implementation, overlapping pairs of coefficients are determined using a simple nested-loop join. Furthermore, during the rendering step for non-aggregate queries, cells with negative counts are not included in the final answer to the query.

Since we assume d dimensions in the multi-dimensional array for a d-attribute relation, c random samples require $c * (d + 1)$ units of space; d units are needed to store the index of the cell and 1 unit is required to store the cell count. Storing c wavelet coefficients also requires the same amount of space, since we need d units to specify the position of the coefficient in the wavelet transform array and 1 unit to specify the value for the coefficient. (Note that the hyper-rectangle and sign information for a base coefficient can easily be derived from its location in the wavelet transform array.) On the other hand, each histogram bucket requires $3 * d + 1$ units of space; $2 * d$ units to specify the low and high boundaries for the bucket along each of the d dimensions, d units to specify the number of distinct values along each dimension, and 1 unit to specify the average frequency for the bucket [27]. Thus, for a given amount of space corresponding to c samples/wavelet coefficients, we store $b \approx \frac{c}{3}$ histogram buckets to ensure a fair comparison between the methods.

Queries. The workload used to evaluate the various approximation techniques consists of four main query types: (1) `SELECT` *queries*: ranges are specified for (a subset of) the attributes in a relation and all tuples that satisfy the conjunctive range predicate are returned as part of the query result: (2) `SELECT-SUM` *queries*: the total `sum` of a particular attribute's values is computed for all tuples that satisfy a conjunctive range predicate over (a subset of) the attributes; (3) `SELECT-JOIN` *queries*: after performing selections on two input relations, an equi-join on a single join dimension is performed and the resulting tuples are output; and (4) `SELECT-JOIN-SUM` *queries*: the total `sum` of an attribute's values is computed over all the tuples resulting from a `SELECT-JOIN`.

For each of the above query types, we have conducted experiments with multiple different choices for: (a) `select` ranges; and (b) `select`, `join`, and `sum` attributes. The results presented in the next section are indicative of the overall observed behavior of the schemes. Furthermore, the queries presented in this paper are fairly representative of typical queries over our data sets.

Answer-quality metrics. In our experiments with aggregate queries (e.g., `SELECT-SUM` queries), we use the *absolute relative error* in the aggregate value as a measure of the accuracy of the approximate query answer. That is, if *actual_aggr* is the result of executing the aggregation query on the actual base relations, while *approx_aggr* is the result of running it on the corresponding synopses, then the accuracy of the approximate answer is given by $\frac{|actual\_aggr - approx\_aggr|}{actual\_aggr}$.

Deciding on an error metric for non-aggregate queries is slightly more involved. The problem here is that non-aggregate queries do not return a single value, but rather a set of tuples (with associated counts). Capturing the "distance" between such an answer and the actual query result requires that we take into account how these two (multi)sets of tuples differ in both: (a) the tuple frequencies; and (b) the actual values in the tuples [16]. (Thus, simplistic solutions like "symmetric difference" are insufficient.) When deciding on an error metric for non-aggregate results, we considered both the *Match And Compare* (MAC) error of Ioannidis and Poosala [16] and the network-flow-based *Earth Mover's Distance* (EMD) error of Rubner et al. [29]. We eventually chose a variant of the EMD error metric, since it offers a number of advantages over MAC error (e.g., computational efficiency, natural handling of non-integral counts) and, furthermore, we found that MAC error can show unstable behavior under certain circumstances [13].

We briefly describe the MAC and EMD error metrics below and explain why we chose the EMD metric.

The EMD and MAC set-error metrics. One of the main observations of Ioannidis and Poosala [16] was that a correct error metric for capturing the distance between two set-valued query answers (i.e., multisets of tuples) should take into account how these two (multi)sets of tuples differ in both: (a) the tuple frequencies; and (b) the actual values in the tuples. A naive option is to simply define the distance between two sets of elements S_1 and S_2 as $|(S_1 - S_2) \cup (S_2 - S_1)|$. However, as discussed in [16], this measure does not take into account the frequencies of occurrences of elements or their values. For example, by the above measure, the two sets $\{5\}$ and $\{5, 5, 5\}$ would be considered to be at a distance of 0 from each other, while the set $\{5\}$ would be at the same distance from both $\{5.1\}$ and $\{100\}$.

In [16], the authors define the notion of *Match And Compare* (MAC) distance to measure the error between two multisets S_1 and S_2. Let $dist(e_1, e_2)$ denote the distance between elements $e_1 \in S_1$ and $e_2 \in S_2$ (in this paper, we use the Euclidean distance between elements). The MAC error involves matching pairs of elements from S_1 and S_2 such that each element appears in at least one matching pair, and the sum of the distances between the matching pairs is minimum. The sum of the matching pair distances, each weighted by the maximum number of matches an element in the pair is involved in, yields the MAC error. Though the MAC error has a number of nice properties and takes both frequency and value of elements in the sets into account, in some cases, it may be unstable [13]. In addition, the MAC error, as defined in [16], could become computationally expensive, since multiple copies of a cell need to be treated separately, thus making set sizes potentially large.

Due to the stability and computational problems of the MAC error, in our experiments, we use the *Earth Mover's Distance* EMD error instead, which we have found to solve the above-mentioned problems. The EMD error metric was proposed by Rubner et al. [29] for computing the dissimilarity between two distributions of points and was applied to computing distances between images in a database. The main idea is to formulate the distance between two (multi)sets as a bipartite network flow problem, where the objective function incorporates the distance in the values of matched elements and the flow captures the distribution of element counts. More formally, the EMD error involves solving the bipartite network flow problem which can be formalized as the following linear programming problem. Let S_1 and S_2 be two sets of elements and let c_i denote the count of element e_i. Without loss of generality, let the sum of the counts of elements in S_1 be greater than or equal to the sum of counts of elements in S_2. Consider an assignment of non-negative flows $f(e_i, e_j)$ such that the following sum is minimized:

$$\sum_{e_i \in S_1} \sum_{e_j \in S_2} f(e_i, e_j) * dist(e_i, e_j) \tag{2}$$

subject to the following constraints:

$$\sum_{e_i \in S_1} f(e_i, e_j) = c_j \tag{3}$$

$$\sum_{e_j \in S_2} f(e_i, e_j) \leq c_i \tag{4}$$

The EMD error, that we employ in this paper[4] is as follows:

$$EMD(S_1, S_2) = \sum_{e_i \in S_1} \sum_{e_j \in S_2} f(e_i, e_j) * dist(e_i, e_j) * \left(\frac{\sum_{e_i \in S_1} c_i}{\sum_{e_j \in S_2} c_j} \right)$$

Thus, intuitively, the flows $f(e_i, e_j)$ distribute the counts of elements in S_1 across elements in S_2 in a manner that the sum of the distances over the flows is minimum. Note that since S_2 has a smaller count than S_1, we require that the inflow into each element e_j of S_2 is equal to c_j (Constraint 3). In addition, the outflow out of each element e_i in S_1 cannot exceed c_i (Constraint 4). In addition, observe that since the count of S_1 could be much larger than that of S_2, we scale the sum in Eq. 2 by the ratio of the sum of counts of S_1 and S_2. This ensures that counts for elements in S_1 that are not covered as part of the flows get accounted for in the EMD error computation.

Thus, the EMD naturally extends the notion of distance between single elements to distance between sets of elements. In addition, the EMD has the nice property that if the counts of S_1 and S_2 are equal, then the EMD is a true metric. There are efficient algorithms available to compute the flows $f(e_i, e_j)$ such that constraints (2), (3) and (4) are satisfied. Another added benefit of the EMD error is that it is naturally applicable to the cases when elements in the sets have non-integral counts. Since in a number of cases, the number of tuples computed by the approximation techniques can be fractions, this is an advantage. Hence we chose EMD as the error metric for non-aggregate queries.

4.2 Experimental results – synthetic data sets

The synthetic data sets we use in our experiments are similar to those employed in the study of Vitter and Wang [33]. More specifically, our synthetic data generator works by populating randomly-selected rectangular regions of cells in the multi-dimensional array. The input parameters to the generator along with their description and default values are as illustrated in Table 2. The generator assigns non-zero counts to cells in r rectangular regions each of whose volume is randomly chosen between v_{min} and v_{max} (the volume of a region is the number of cells contained in it). The regions themselves are uniformly distributed in the multi-dimensional array. The sum of the counts for all the cells in the array is specified by the parameter t. Portion $t \cdot (1 - n_c)$ of the count is partitioned across the r regions using a Zipfian distribution with value z. Within each region, each cell is assigned a count using a Zipfian distribution with value between z_{min} and z_{max}, and based on the L_1 distance of the cell from the center of the region. Thus, the closer a cell is to the center of its region, the larger is its count value. Finally, we introduce noise into the data set by randomly choosing cells such that these noise cells constitute

[4] Rubner et al. [29] define the EMD error as the ratio $\frac{\sum_{e_i \in S_1} \sum_{e_j \in S_2} f(e_i, e_j) * dist(e_i, e_j)}{\sum_{e_j \in S_2} c_j}$.

Table 2. Input parameters to synthetic data generator

Parameter	Description	Default value
d	Number of dimensions	2
s	Size of each dimension (equal for all dimensions)	1,024
r	Number of regions	10
v_{min}, v_{max}	Minimum and maximum volume of each region	2,500, 2,500
z	Skew across regions	0.5
z_{min}, z_{max}	Minimum and maximum skew within each region	1.0, 1.0
n_v, n_c	Noise volume and noise count	0.05, 0.05
t	Total count	1,000,000
c	Number of coefficients/ samples retained	1,250
b	Number of histogram buckets	420
sel	Selectivity in terms of volume	4%

Table 3. Wavelet transform computation times

	No. of Cells in multi-dimensional array			
	250×10^3	1×10^6	4×10^6	16×10^6
Exec. time (sec	6.3	26.3	109.9	445.4

a fraction n_v of the total number of non-zero cells. The noise count $t \cdot n_c$ is then uniformly distributed across these noise cells.

Note that with the default parameter settings described in Table 2, there are a total of a million cells of which about 25,000 have non-zero counts. Thus, the density of the multi-dimensional array is approximately 2.5%. Further, in the default case, the approximate representations of the relations occupy only 5% of the space occupied by the original relation – this is because we retain 1,250 samples/coefficients out of 25,000 non-zero cells which translates to a compression ratio of 20. The same is true for histograms. Finally, we set the default selectivity of range queries on the multi-dimensional array to be 4% – the SELECT query range along each dimension was set to (512,720).

Time to compute the wavelet transform. In order to demonstrate the efficiency of our algorithm for computing the wavelet transform of a multi-dimensional array, in Table 3, we present the running times of COMPUTEWAVELET as the number of cells in the multi-dimensional array is increased from 250,000 to 16 million. The density of the multi-dimensional array is kept constant at 2.5% by appropriately scaling the number of cells with non-zero counts in the array. From the table, it follows that the computation time of our COMPUTEWAVELET algorithm scales linearly with the total number of cells in the array. We should note that the times depicted in Table 3 are actually dominated by CPU-computation costs – COMPUTEWAVELET required a single pass over the data in all cases.

SELECT queries. In our first set of experiments, we carry out a sensitivity analysis of the EMD error for SELECT queries to parameters like storage space, skew in cell counts within a region, cell density, and query selectivity. In each experiment, we vary the parameter of interest while the remaining parameters are fixed at their default values. Our results indicate that for a broad range of parameter settings, wavelets outperform both sampling and histograms – in some cases, by more than an order of magnitude.

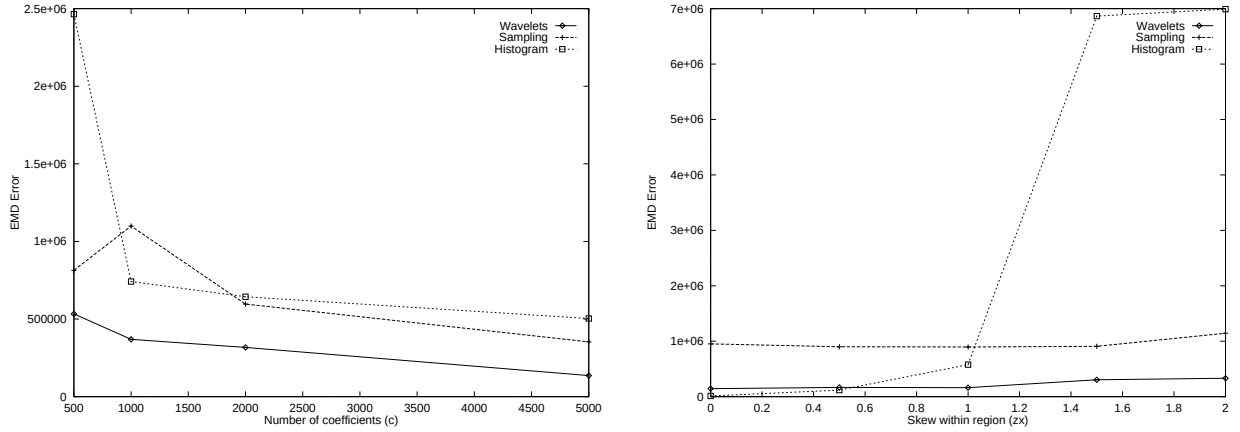
• *Storage space*. Figure 12a depicts the behavior of the EMD error for the three approximation methods as the space (i.e., number of retained coefficients) allocated to each is increased from 2% to 20% of the relation. For a given value of the number of wavelet coefficients c along the x-axis, histograms are allocated space for $\approx \frac{c}{3}$ buckets. As expected, the EMD error for all the cases reduces as the amount of space is increased. Note that for 500 coefficients, the EMD error for histograms is almost five times worse that the corresponding error for wavelets. This is because the few histogram buckets are unable to accurately capture the skew within each region (in our default parameter settings, the Zipfian parameter for the skew within a region is 1).

• *Skew within regions*. In Fig. 12b, we plot the EMD error as the Zipfian parameter z_{max} that controls the maximum skew within each region is increased from 0 to 2.0. Histograms perform the best for values of z_{max} between 0 and 0.5 when the cell counts within each region are more or less uniformly distributed. However, once the maximum skew increases beyond 0.5, the histogram buckets can no longer capture the data distribution in each region accurately. As a consequence, we observe a spike in the EMD error for region skew corresponding to a value of $z_{max} = 1.5$. Incidentally, a similar behavior for MaxDiff histograms has been reported earlier in [16].

• *Cell density*. In Fig. 13a, we plot the graphs for EMD error as v_{max}, the maximum volume of regions is varied between 1,000 (1% density) and 5,000 (5% density) (v_{min} is fixed at 1,000). As the number of non-zero cells in the multi-dimensional array increases, the number of coefficients, samples and histogram buckets needed to approximate the underlying data also increases. As a consequence, in general, the EMD error is more when regions have larger volumes. Note the sudden jump in the EMD error for histograms when the volume becomes 5,000. This is because the histogram buckets overestimate the total of the cell counts in the query region by almost 50%. In contrast, the error in the sum of the cell counts within the query range with wavelets is less than 0.1%.

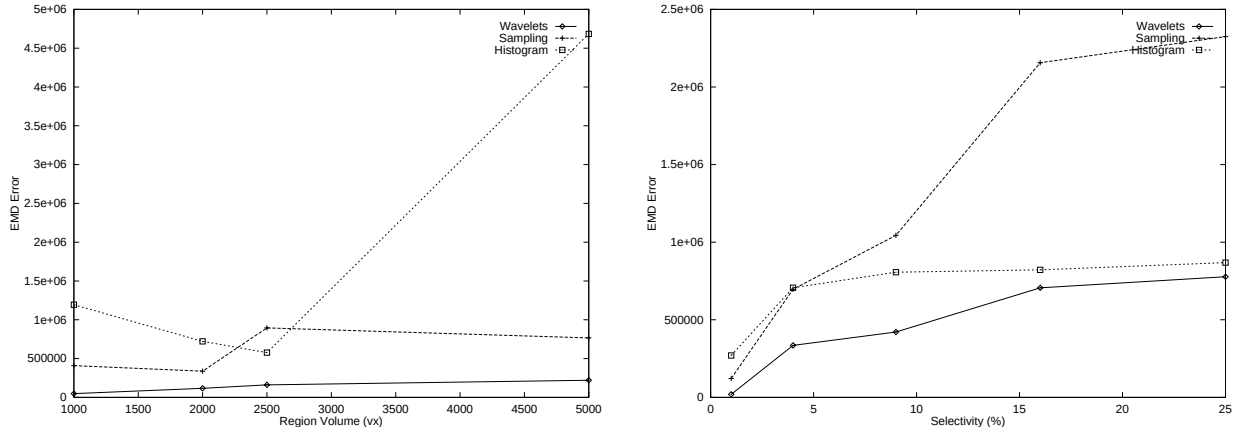
• *Selectivity of query*. Figure 13b illustrates the EMD errors for the techniques as the selectivity of range queries is increased from 2% to 25%. Since the number of tuples in both the accurate as well as the approximate answer increase, the EMD error increases as the selectivity of the query is increased (recall that the EMD error is the sum of the pairwise distances between elements in the two sets of answers weighted by the flows between them).

SELECT-SUM queries. Figure 14a depicts the performance of the various techniques for SELECT-SUM queries as the allocated space is increased from 2% to 20% of the relation. Both wavelets and histograms exhibit excellent performance compared to random sampling; the relative errors are extremely low for both techniques – 0.2% and 0.6%, respectively. These



a **b**

Fig. 12. SELECT queries: sensitivity to **a** allocated space; **b** skew within regions



a **b**

Fig. 13. SELECT queries: sensitivity to **a** cell density; **b** query selectivity

results are quite different from the EMD error curves for the three schemes (see Fig. 12a). We can thus conclude that although histograms are excellent at approximating aggregate frequencies, they are not as good as wavelets at capturing the distribution of values accurately. In [33], wavelets were shown to be superior to sampling for aggregation queries – however, the work in [33] did not consider histograms.

SELECT-JOIN and SELECT-JOIN-SUM queries. For join queries, in Fig. 14b, we do not show the errors for sampling since in almost all cases, the final result contained zero tuples. In addition, we only plot the relative error results for SELECT- JOIN-SUM queries, since the EMD error graphs for SELECT- JOIN queries were similar.

When the number of coefficients retained is 500, the relative error with wavelets is more than four times better than the error for histograms – this is because the few histogram buckets are not as accurate as wavelets in approximating the underlying data distribution. For histograms, the relative error decreases for 1,000 and 2,000 coefficients, but shows an abrupt increase when the number of coefficients is 5,000. This is because at 5,000 coefficients, when we visualized the histogram buckets, we found that a large bucket appeared in the query region (that was previously absent), in order to capture the underlying noise in the data set. Cells in this bucket contributed to the dramatic increase in the join result size, and subsequently, the relative error.

We must point out that although the performance of histograms is erratic for the query region in Fig. 14b, we have found histogram errors to be more stable on other query regions. Even for such regions, however, the errors observed for histograms were, in most cases, more than an order of magnitude worse than those for wavelets. Note that the relative error for wavelets is extremely low (less than 1%) even when the coefficients take up space that is about 4% of the relation.

Query execution times. In order to compare the query processing times for the various approaches, we measured the time (in seconds) for executing a SELECT-JOIN-SUM query using each approach. We do not consider the time for random sampling since the join results with samples did not generate any tuples, except for very large sample sizes. The running time of the join query on the original base relations (using an indexed nested-loop join) to produce an exact answer was 3.6 s. In practice, we expect that this time will be much higher since in our case, the entire relations fit in main memory. As is evident from Fig. 15a, our wavelet-based technique is more

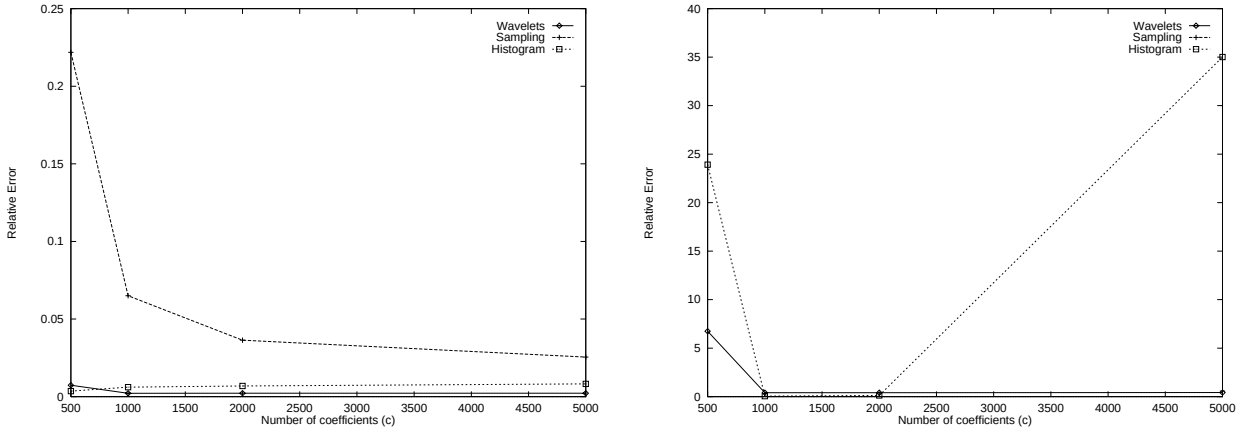


Fig. 14. Effect of allocated space on **a** `SELECT-SUM`; **b** `SELECT-JOIN-SUM` queries

Technique	Number of Coefficients			
	500	1000	2000	5000
Wavelets	0.01	0.02	0.04	0.08
Histograms	9.8	1.48	0.43	1.26

a

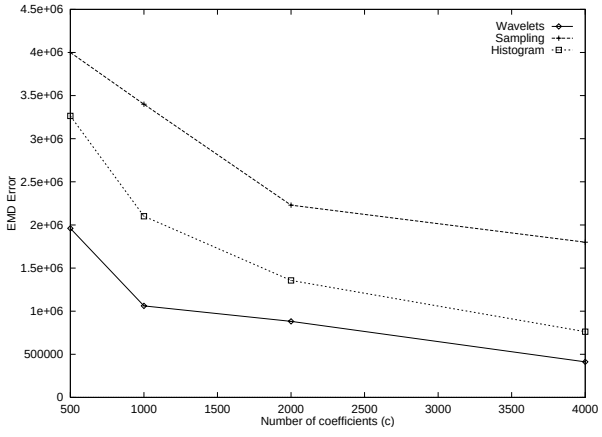


Fig. 15. a `SELECT-JOIN-SUM` query execution times; **b** `SELECT` query errors on real-life data

than two orders of magnitude faster compared to running the queries on the entire base relations.

In addition, note that the performance of histograms is much worse than that of wavelets. The explanation lies in the fact that the `join` processing algorithm of Ioannidis and Poosala [16] requires joining histograms to be partially expanded to generate the tuple-value distribution for the corresponding approximate relations. The problem with this approach is that the intermediate relations can become fairly large and may even contain more tuples than the original relations. For example, with 500 coefficients, the expanded histogram contains almost five times as many tuples as the base relations. The sizes of the approximate relations decrease as the number of buckets increase, and thus execution times for histograms drop for larger numbers of buckets. In contrast, in our wavelet approach, join processing is carried out exclusively in the compressed domain, that is, joins are performed directly on the wavelet coefficients without ever materializing intermediate relations. The tuples in the final query answer are generated at the very end as part of the rendering step and this is the primary reason for the superior performance of the wavelet approach.

4.3 Experimental results – real-life data sets

We obtained our real-life data set from the US Census Bureau (`www.census.gov`). We employed the Current Population Survey (CPS) data source and within it the Person Data Files of the March Questionnaire Supplement. We used the 1992 data file for the select and select sum queries, and the 1992 and 1994 data files for the join and join sum queries. For both files, we projected the data on the following four attributes whose domain values were previously coded: *age* (with value domain 0 to 17), *educational attainment* (with value domain 0 to 46), *income* (with value domain 0 to 41), and *hours per week* (with value domain 0 to 13). Along with each tuple in the projection, we stored a count which is the number of times it appears in the file. We rounded the maximum domain values off to the nearest power of 2 resulting in domain sizes of 32, 64, 64, and 16 for the four dimensions, and a total of 2 million cells in the array. The 1992 and the 1994 collections had 16,271 and 16,024 cells with non-zero counts, respectively, resulting in a density of ≈ 0.001. (The data-file sizes for our CPS data projections were approximately 318 kB (1992 collection) and 313 kB (1994 collection).) Even though the density of the resulting joint-frequency arrays is very low, we did observe large dense regions within the arrays when we visualized the

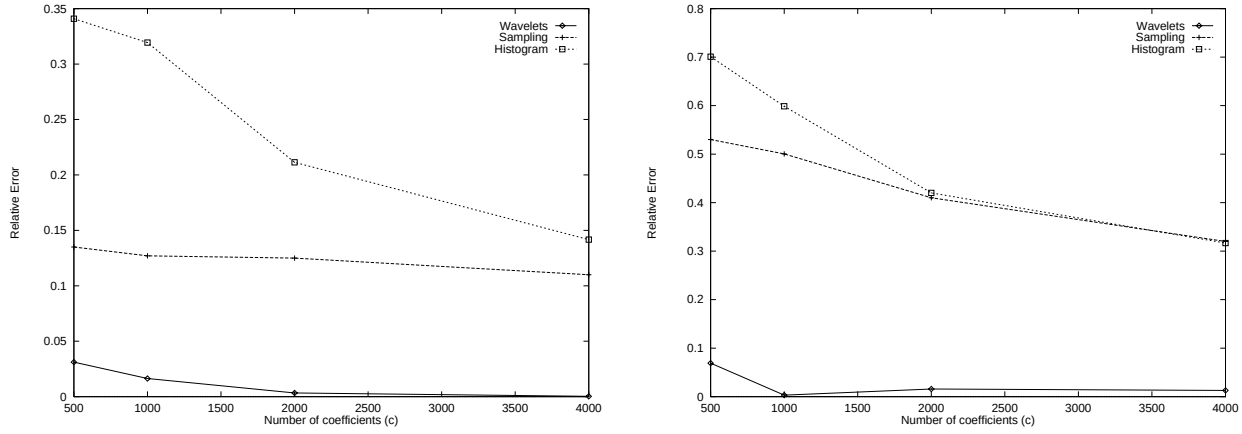


Fig. 16. a `SELECT-SUM` and **b** `SELECT-JOIN-SUM` queries on real-life data

data – these dense regions spanned the entire domains of the *age* and *income* dimensions.

For all the queries, we used the following select range: $5 \leq age < 10$ and $10 \leq income < 15$ that we found to be representative of several select ranges that we considered (the remaining two dimensions were left unspecified). The selectivity of the query was 1,056/16,271= 6%. For `sum` queries, the `sum` operation was performed on the *age* dimension. For `join` queries, the `join` was performed on the *age* dimension between the 1992 and 1994 data files.

SELECT queries. In Figs. 15b and 16a, we plot the EMD error and relative error for `SELECT` and `SELECT-SUM` queries, respectively, as the space allocated for the approximations is increased from 3% to 25% of the relation. From the graphs, it follows that wavelets result in the least value for the EMD error, while sampling has the highest EMD error. For `SELECT-SUM` queries, wavelets exhibit more than an order of magnitude improvement in relative error compared to both histograms and sampling (the relative error for wavelets is between 0.5% and 3%). Thus, the results for the select queries indicate that wavelets are effective at accurately capturing both the value as well as the frequency distribution of the underlying real-life data set.

Note that unlike the EMD error and the synthetic data cases, the relative error for sampling is better than for histograms. We conjecture that one of the reasons for this is the higher dimensionality of the real-life data sets, where histograms are less effective.

JOIN queries. We only plot the results of the `SELECT-JOIN-SUM` queries in Fig. 16b, since the EMD error graphs for `SELECT-JOIN` queries were similar. Over the entire range of coefficients, wavelets outperform sampling and histograms, in most cases by more than an order of magnitude. With the real-life data set, even after the `join`, the relative aggregate error using wavelets is very low and ranges between 1% to 6%. The relative error of all the techniques improve as the amount of allocated space is increased. Note that compared to the synthetic data sets, where the result of a `join` over samples contained zero tuples in most cases, for the real-life data sets, sampling performs quite well. This is because the size of the domain of the *age* attribute on which the join is performed is only 18, which is quite small. Consequently, the result of the `join` query over the samples is no longer empty.

4.4 Summary

In summary, our experimental results have demonstrated that our wavelet-based approach consistently outperforms earlier approaches based on random sampling and histograms. Sampling suffers mainly for non-aggregate queries since it always produces small subsets of the exact query answer. As we expected, this problem with random sampling is particularly acute when `join` operations are involved, as the result of joining sample synopses is often the *empty set* (especially for sparse, multi-dimensional data). On the other hand, histograms give poor approximate-querying performance for non-uniform, high-dimensional data sets, as such data distributions cannot be accurately captured with a small number of disjoint rectangular buckets containing uniformly distributed points. Our results prove that out wavelet-based approach does not suffer from such problems. More specifically, even though wavelets have their weaknesses (e.g., they can behave poorly for very "spiky" distributions), we have found that they are very effective in capturing *localities* in the input data distribution, that is, regions of neighboring data cells with similar frequencies. Further, the hierarchical nature of the wavelet decomposition allows wavelet coefficients to capture such localities at different levels of resolution in a very concise and accurate manner. Based on our experience, most data sets in real-life DSS applications do exhibit such localities; thus, we firmly believe that the wavelet-based approach proposed in this paper is an effective approximate query processing solution for DSS applications.

5 Conclusions

Approximate query processing is slowly emerging as an essential tool for numerous data-intensive applications requiring interactive response times. Most work in this area, however, has so far been limited in its scope and conventional

approaches based on sampling or histograms appear to be inherently limited when it comes to complex approximate queries over high-dimensional data sets. In this paper, we have proposed the use of multi-dimensional wavelets as an effective tool for general-purpose approximate query processing in modern, high-dimensional applications. Our approach is based on building wavelet-coefficient synopses of the data and using these synopses to provide approximate answers to queries. We have developed novel query processing algorithms that operate directly on the wavelet-coefficient synopses of relational data, allowing us to process arbitrarily complex queries *entirely* in the wavelet-coefficient domain. This guarantees extremely fast response times since our approximate query execution engine can do the bulk of its processing over compact sets of wavelet coefficients, essentially postponing the expansion into relational tuples until the end-result of the query. We have also proposed a novel I/O-efficient wavelet decomposition algorithm for building the synopses of relational data. Finally, we have conducted an extensive experimental study with synthetic as well as real-life data sets to determine the effectiveness of our wavelet-based approach compared to sampling and histograms. Our results demonstrate that our wavelet-based query processor: (a) provides approximate answers of better quality than either sampling or histograms; (b) offers query execution-time speedups of more than two orders of magnitude; and (c) guarantee fast synopsis construction times that scale linearly to the size of the relation.

Acknowledgements. We would like to thank Vishy Poosala for providing us with his MaxDiff histogram computation code. This work was partially supported by the Korea Science and Engineering Foundation (KOSEF) through the Advanced Information Technology Research Center (AITrc).

References

1. Acharya S., Gibbons P.B., Poosala V., Ramaswamy S. Join Synopses for Approximate Query Answering. In: Proc. 1999 ACM SIGMOD International Conference on Management of Data, pp. 275–286, Philadelphia, Pa., May 1999
2. Amsaleg L., Bonnet P., Franklin M.J., Tomasic A., Urhan T. Improving Responsiveness for Wide-Area Data Access. IEEE Data Engineering Bulletin, 20(3):3–11, 1997. (Special Issue on Improving Query Responsiveness)
3. Barbara D., DuMouchel W., Faloutsos C., Haas P.J., Hellerstein J.M., Ioannidis Y., Jagadish H.V., Johnson T., Ng R., Poosala V., Ross K.A., Sevcik K.C. The New Jersey Data Reduction Report. IEEE Data Engineering Bulletin, 20(4):3–45, 1997. (Special Issue on Data Reduction Techniques)
4. Cochran W.G. Sampling Techniques. Wiley, New York, 1977. (3rd edn)
5. Deshpande P.M., Ramasamy K., Shukla A., Naughton J.F. Caching Multidimensional Queries Using Chunks. In: Proc. 1998 ACM SIGMOD International Conference on Management of Data, pp. 259–270, Seattle, Wash., June 1998
6. Ester M., Kohlhammer J., Kriegel H.-P. The DC-Tree: a Fully Dynamic Index Structure for Data Warehouses. In: Proc. Sixteenth International Conference on Data Engineering, pp. 379–388, San Diego, Calif., February 2000
7. Gibbons P.B., Matias Y. New Sampling-Based Summary Statistics for Improving Approximate Query Answers. In: Proc. 1998 ACM SIGMOD International Conference on Management of Data, pp. 331–342, Seattle, Wash., June 1998
8. Gibbons P.B., Matias Y., Poosala V. Fast Incremental Maintenance of Approximate Histograms. In: Proc. 23rd International Conference on Very Large Data Bases, pp. 466–475, Athens, Greece, August 1997
9. Gibbons P.B., Matias Y., Poosala V. Aqua Project White Paper. Unpublished Manuscript (Bell Laboratories), December 1997
10. Haas P.J. Large-Sample and Deterministic Confidence Intervals for Online Aggregation. In: Proc. Ninth International Conference on Scientific and Statistical Database Management, Olympia, Wash., August 1997
11. Haas P.J., Hellerstein J.M. Ripple Joins for Online Aggregation. In: Proc. 1999 ACM SIGMOD International Conference on Management of Data, pp. 287–298, Philadelphia, Pa., May 1999
12. Hellerstein J.M., Haas P.J., Wang H.J. Online Aggregation. In: Proc. 1997 ACM SIGMOD International Conference on Management of Data, Tucson, Ariz., May 1997
13. Ioannidis Y.E. Personal Communication, August 1999
14. Ioannidis Y.E., Christodoulakis S. On the Propagation of Errors in the Size of Join Results. In: Proc. 1991 ACM SIGMOD International Conference on Management of Data, pp. 268–277, Denver, Colo., May 1991
15. Ioannidis Y.E., Poosala V. Balancing Histogram Optimality and Practicality for Query Result Size Estimation. In: Proc. 1995 ACM SIGMOD International Conference on Management of Data, pp. 233–244, May 1995
16. Ioannidis Y.E., Poosala V. Histogram-Based Approximation of Set-Valued Query Answers. In: Proc. 25th International Conference on Very Large Data Bases, Edinburgh, Scotland, September 1999
17. Jagadish H.V. Linear Clustering of Objects with Multiple Attributes. In: Proc. 1990 ACM SIGMOD International Conference on Management of Data, pp. 332–342, Atlantic City, N.J., May 1990
18. Jawerth B., Sweldens W. An Overview of Wavelet Based Multiresolution Analyses. SIAM Review, 36(3):377–412, 1994
19. Lee J.-H., Kim D.-H., Chung C.-W. Multi-dimensional Selectivity Estimation Using Compressed Histogram Information. In: Proc. 1999 ACM SIGMOD International Conference on Management of Data, pp. 205–214, Philadelphia, Pa., May 1999
20. Lipton R.J., Naughton J.F., Schneider D.A. Practical Selectivity Estimation through Adaptive Sampling. In: Proc. 1990 ACM SIGMOD International Conference on Management of Data, pp. 1–12, Atlantic City, N.J., May 1990
21. Matias Y., Vitter J.S., Wang M. Wavelet-Based Histograms for Selectivity Estimation. In: Proc. 1998 ACM SIGMOD International Conference on Management of Data, pp. 448–459, Seattle, Wash., June 1998
22. Matias Y., Vitter J.S., Wang M. Dynamic Maintenance of Wavelet-Based Histograms. In: Proc. 26th International Conference on Very Large Data Bases, Cairo, Egypt, September 2000
23. Muralikrishna M., DeWitt D.J. Equi-Depth Histograms for Estimating Selectivity Factors for Multi-Dimensional Queries. In: Proc. 1988 ACM SIGMOD International Conference on Management of Data, pp. 28–36, Chicago, Ill., June 1988
24. Natsev A., Rastogi R., Shim K. WALRUS: a Similarity Retrieval Algorithm for Image Databases. In: Proc. 1999 ACM SIGMOD International Conference on Management of Data, Philadelphia, Pa., May 1999
25. Orenstein J.A. Spatial Query Processing in an Object-Oriented Database System. In: Proc. 1986 ACM SIGMOD International Conference on Management of Data, pp. 326–336, Washington, D.C., June 1986

26. Poosala V., Ganti V. Fast Approximate Answers to Aggregate Queries on a Data Cube. In: Proc. Eleventh International Conference on Scientific and Statistical Database Management, Cleveland, Ohio, July 1999
27. Poosala V., Ioannidis Y.E. Selectivity Estimation Without the Attribute Value Independence Assumption. In: Proc. 23rd International Conference on Very Large Data Bases, pp. 486–495, Athens, Greece, August 1997
28. Poosala V., Ioannidis Y.E., Haas P.J., Shekita E.J. Improved Histograms for Selectivity Estimation of Range Predicates. In: Proc. 1996 ACM SIGMOD International Conference on Management of Data, pp. 294–305, Montreal, Quebec, June 1996
29. Rubner Y., Tomasi C., Guibas L. A Metric for Distributions with Applications to Image Databases. In: Proc. 1998 IEEE International Conference on Computer Vision, Bombay, India, 1998
30. Sarawagi S., Stonebraker M. Efficient Organization of Large Multidimensional Arrays. In: Proc. Tenth International Conference on Data Engineering, pp. 328–336, Houston, Tex., February 1994
31. Särndal C.-E., Swensson B., Wretman J. Model Assisted Survey Sampling. Springer, Berlin Heidelberg New York, (Springer Series in Statistics), 1992
32. Stollnitz E.J., DeRose T.D., Salesin D.H. Wavelets for Computer Graphics – Theory and Applications. Morgan Kaufmann, San Francisco, Calif., 1996
33. Vitter J.S., Wang M. Approximate Computation of Multidimensional Aggregates of Sparse Data Using Wavelets. In: Proc. 1999 ACM SIGMOD International Conference on Management of Data, Philadelphia, Pa., May 1999
34. Vitter J.S., Wang M., Iyer B. Data Cube Approximation and Histograms via Wavelets. In: Proc. Seventh International Conference on Information and Knowledge Management, pp. 96–104, Bethesda, Md., November 1998