Data Cube Approximation and Histograms via Wavelets

(Extended Abstract)

Jeffrey Scott Vitter*
Center for Geometric Computing
and Department of Computer Science
Duke University
Durham, NC 27708–0129 USA
jsv@cs.duke.edu

Min Wang†
Center for Geometric Computing
and Department of Computer Science
Duke University
Durham, NC 27708–0129 USA
minw@cs.duke.edu

Bala Iyer
Database Technology Institute
IBM Santa Teresa Laboratory
P.O. Box 49023
San Jose, CA 95161 USA
balaiyer@vnet.ibm.com

Abstract

There has recently been an explosion of interest in the analysis of data in data warehouses in the field of On-Line Analytical Processing (OLAP). Data warehouses can be extremely large, yet obtaining quick answers to queries is important. In many situations, obtaining the exact answer to an OLAP query is prohibitively expensive in terms of time and/or storage space. It can be advantageous to have fast, approximate answers to queries.

In this paper, we present an I/O-efficient technique based upon a multiresolution wavelet decomposition that yields an approximate and space-efficient representation of the data cube, which is one of the core OLAP operators. We build our compact data cube on the logarithms of the partial sums of the raw data values of a multidimensional array. We get excellent approximations for on-line range-sum queries with limited space usage and computational cost. Multiple data cubes can be handled simultaneously. Each query can generally be answered, depending upon the accuracy supported, in one I/O or a small number of I/Os. Experiments show that our method performs significantly better than other approximation techniques such as histograms and random sampling.

1. Introduction

Computing multiple related group-bys and aggregates is one of the core operations of On-Line Analytical Processing (OLAP) applications. Data warehouses often represent data in the form of several multidimensional databases (MDDB), also known as data cubes [7].

Consider a very simple multidimensional model, in which we have the dimensions *age, income*, and *education_level* and the "measure" *population.* The raw data are stored as a three-dimensional array A. For example, one "cell" of the array A may correspond to (age = 30, $income$ = \$45K, $education\_level$ = 5) with a *population* value of 5000, where education level 5 corresponds to high school graduate. Thus, an MDDB can be viewed as a d-dimensional array A, indexed by the values of the d dimensions (or *functional attributes*), whose cells contain the values of the *measure attribute* for the corresponding combination of the functional attributes. In this paper we shall call this kind of multidimensional array A the *raw data cube* (or just *data cube*) to distinguish it from the *extended data cube* and the *partial sum data cube*, which will be defined later.

Gray et al. [7] propose that the domain of each dimension be augmented with an additional value for each aggregation operation, denoted by *all*, to store aggregated values of the measure attribute among all the cells along that dimension, and this results in the *extended data cube.* In the above example, if we consider the aggregation operation *SUM*, any range-sum query of (*age, income, education_level*), in which each attribute is either a singleton value or *all*, can be answered by accessing a single cell in the extended data cube. For example, the total population of 30-year-olds having an income of \$45K is a query specified by (30, \$45K, *all*), which can be answered by one cell access.

An important class of queries on the data cube are the so-called *range-sum* queries, which are defined by applying the *SUM* operation over a selected contiguous range in the domains of some of the attributes [12]. For instance, in the data cube described above, an example of a range-sum query is to determine the total population for people with age from 25 to 45 and with income from \$50K to \$70K. Range-sum queries are important because several other classes of queries on the data cube are special cases, such as singleton queries and slice queries.

Several efficient algorithms for Relational OLAP (ROLAP) have been developed to compute the extended data cube [10, 1]. Zhao et al. [26] propose an array-based algorithm to compute the extended data cube for Multidimensional OLAP (MOLAP) systems. Given appropriate compression techniques, the MOLAP algorithm can be significantly faster than the ROLAP algorithms.

Previous work on data cube computation has concentrated on how to compute the exact data cube. But in reality, a data warehouse usually has more than one table (in a ROLAP system) or data cube (in a MOLAP system), and each of these tables or cubes contains a very large number of tuples or entries. A user can query on any element in an extended data cube computed from any of those relations/cubes in an OLAP query. Computing all the extended data cubes and storing and retrieving them on disk becomes infeasible when the number of underlying relations/cubes is large since no enough disk storage is available. On the other hand, even

*Supported in part by Army Research Office MURI grant DAAH04–96–1–0013 and by National Science Foundation research grant CCR–9522047.

†Contact author. Supported in part by an IBM Graduate Fellowship and by Army Research Office MURI grant DAAH04–96–1–0013.

with a huge amount of storage space, a range-sum query on a data cube may need to access all the cells covered by that range so that they can be summed. In an interactive exploration of multiple data cubes, which is a dominant OLAP application area, it is imperative to have a system with fast response time.

Ho et al. [12] present an efficient algorithm to speed up range-sum queries on a single data cube. The main idea is to preprocess the (raw) data cube A and precompute all the multidimensional partial sums, which can be represented in what we call the *partial sum data cube* P. Any range-sum query can be answered by accessing and computing $2^{d'}$ entries from the partial sum data cube, where d' is the number of dimensions for which ranges have been specified in the query. For example, in one dimension, the answer to a range-sum query

$$l_1 \le D_1 \le h_1 \tag{1}$$

(for which $d' = 1$) can be answered either as $\sum_{l_1 \le i \le h_1} A[i]$ in terms of the extended data cube or more efficiently as

$$P[h_1] - P[l_1 - 1]$$

in terms of the partial sum data cube. In two dimensions, the range-sum query

$$l_1 \le D_1 \le h_1 \quad \text{AND} \quad l_2 \le D_2 \le h_2$$

(for which $d' = 2$) can be answered either as

$$\sum_{l_1 \le i_1 \le h_1} \sum_{l_2 \le i_2 \le h_2} A[i_1, i_2]$$

in terms of the extended data cube or more efficiently as

$$P[h_1, h_2] - P[l_1 - 1, h_2] - P[h_1, l_2 - 1] + P[l_1 - 1, l_2 - 1]$$

in terms of the partial sum data cube. If query (1), for which $d' = 1$, is given in a two-dimensional setting, the answer can be computed either as $\sum_{l_1 \le i_1 \le h_1} A[i_1, all]$ in terms of the extended data cube or more efficiently as

$$P[h_1, |D_2| - 1] - P[l_1 - 1, |D_2| - 1]$$

in terms of the partial sum data cube.

The problem with this partial sum approach is that the partial sums are typically more dense in terms of storage representation than the original data. The resulting storage required can be proportional to the size of the raw data cube, which is very large. The appropriate $2^{d'}$ partial sum values for a given range-sum query might be stored in different disk blocks in the external memory and accessing them may require up to $2^{d'}$ disk I/Os, which can be very expensive in terms of I/O for high dimensions.

There are a number of scenarios in which a user may prefer an approximate answer in a few seconds over an exact answer that requires tens of minutes or more to compute. An example is a drill-down query sequence in data mining [11]. Another consideration is that the extended data cube may be remote and currently unavailable, so that finding an exact answer is not an option, until the data again become available [6].

In this paper, we present an I/O-efficient technique based upon a multiresolution wavelet decomposition that yields a compact and accurate representation of the data cube. We build a data cube on the *logarithms* of the partial sums of the raw data values. The resulting multidimensional wavelet decomposition, after normalization and thresholding to reduce storage cost, provides fast and accurate answers to on-line range-sum queries.

The idea of using wavelet techniques in database approximation was first proposed by Matias, Vitter, and Wang [16]. Some of the new contributions in this paper are the following:

1. We design I/O efficient algorithm to compute the partial sum cube and multidimensional wavelet decomposition when the raw data cube is huge and can not fit in internal memory. In [16], the data arrays considered are low-dimensional and small in size, and thus the I/O efficiency of the wavelet decomposition operation is not addressed.
2. We propose a new thresholding method based on the logarithm transform that dramatically reduces the relative error and even the absolute error in the approximation of high-dimensional data.
3. By applying the new thresholding method in building wavelet-based histograms as proposed in [16], we can achieve much better accuracy even for the low-dimensional data, especially when reducing relative errors in the approximation becomes important.

Our method is designed to accommodate the approximation of several data cubes. Each query can generally be answered, depending upon the accuracy supported, in one I/O or a small number of I/Os. For example, we get a very good approximation of the partial sum data cube by storing one disk block's worth of wavelet coefficients. In such a case, any set of queries on that data cube can be answered collectively with a total of only one I/O. The accuracy of the resulting approximation, as a function of the storage used, is noticeably better than those of other approximation techniques based upon histograms and random sampling.

In the next section, we discuss the construction of the approximate partial sum data cube and the analysis of I/O performance. The I/O complexity is often $O(N/B)$, which is best possible, and is never more than the I/O bound for sorting, namely, $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$, where M is the size of internal memory and B is the disk block size [25]. The on-line query phase, which requires only a constant number of I/Os in total for an arbitrary number of queries, is explained and analyzed in Section 3. Our experimental results in Section 4 demonstrate a competitive advantage of our wavelet technique over histograms and random sampling in many cases, while while it is comparable to the histogram method for other types of queries.

2. Our Compact Data Cube Construction Algorithm

We adopt the notations in [12] to formulate the problem. Let $D = \{D_1, D_2, \ldots, D_d\}$ denote the set of dimensions, where each dimension corresponds to a functional attribute. We represent the d-dimensional (raw) data cube A by a d-dimensional array of size $|D_1| \times |D_2| \times \cdots \times |D_d|$, where $|D_i|$ is the size of dimension D_i. We assume without loss of generality that each array has starting index 0. For convenience, we call each array element a *cell*. We let $N = \prod_{1 \le i \le d} |D_i|$ denote the total size (number of cells) of data cube A.

The problem of computing a range-sum query in a d-dimensional data cube can be formulated as follows:

$$Sum(l_1 : h_1, \ldots, l_d : h_d) = \sum_{l_1 \le i_1 \le h_1} \cdots \sum_{l_d \le i_d \le h_d} A[i_1, \ldots, i_d].$$

The partial sum data cube P is a d-dimensional array of size $|D_1| \times |D_2| \times \cdots \times |D_d|$ (which is the same as the size of A). Its cells are defined as

$$\begin{aligned} P[x_1, \ldots, x_n] &= Sum(0 : x_1, \ldots, 0 : x_d) \\ &= \sum_{0 \le i_1 \le x_1} \cdots \sum_{0 \le i_d \le x_d} A[i_1, \ldots, i_d]. \end{aligned}$$

At a high level, our approximate data cube construction algorithm works as follows:

1. In a preprocessing step, we form the partial sum data cube P from the (raw) data cube A. (In our method, we further process P by replacing each cell value by its natural logarithm.)
2. We compute the wavelet decomposition of P, obtaining a set of N coefficients, where N is the size of array A.
3. We keep only the C most significant wavelet coefficients, for some C that corresponds to the desired storage usage and accuracy. The choice of which C coefficients to keep depends upon the particular thresholding method we use.

In the on-line phase, a query is answered by using the C wavelet coefficients to reconstruct approximations of the necessary values in P. Details of the on-line phase are given in Section 3.

Steps 1, 2, and 3, respectively, are discussed in the following three subsections. We show in particular that the total I/O complexity of Steps 1 and 2 is often linear in $O(N/B)$, which is optimal, and is never worse than $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$, which is the number of I/Os required for sorting [2, 25, 24]. If we have $(M/B)^c \geq N/B$ for small c, which is typically the case, then the I/O time is $O(N/B)$. If not, it is still possible to get an $O(N/B)$-I/O algorithm by proper use of "chunking." Step 3 uses only $O(N/B)$ I/Os. The thresholding in Step 3 is interesting in that it takes advantage of the logarithm transform of Step 1 to achieve extremely good relative errors in its approximation of the partial sum data cube P.

2.1. Computing the Partial Sum Data Cube P

There are two reasons why we choose to do the wavelet decomposition on the partial sum data cube P instead of on the (raw) data cube A. The first reason is that P is monotone nondecreasing, and a compact data cube built on P seems to give a better approximation than one built directly on A. This interesting observation was made in our previous paper on wavelet-based histograms for selectivity estimation [16] and is confirmed by a series of experiments using different types of data in building compact data cubes. The second reason is that in order to answer a range-sum query using the compact data cube built on P, all we need to do in the on-line phase is to reconstruct the values corresponding to the boundaries of the ranges (instead of reconstructing all the values covered by the query, as in an extended data cube).

Computing the partial sum data cube P from the (raw) data cube A in a naive manner can be expensive in terms of I/O when the size of A is very large and cannot fit in internal memory (which is likely in many OLAP applications). Thus, it is important to design I/O-efficient algorithms.

To compute the partial sum data cube P, we need to do a series of partial sum operations along each of the d dimensions of A. The standard programming language technique of storing a multidimensional array in a certain dimension order (such as row-major order or column-major order in the case of two-dimensional arrays) may not be efficient in terms of I/O when used with virtual memory. For example, in a row-major representation of a two-dimensional array, the partial sums along each row can be computed efficiently, but accessing the array column-wise will cause almost one page fault per element accessed.

To achieve optimal I/O efficiency even when virtual memory is used, we can "chunk" the array, as suggested by [20, 19] and implemented in [23, 26] for other applications. Chunking is a way to divide a d-dimensional array into d-dimensional chunks in which each chunk is stored as one block on disk. Figure 1 shows a chunked three-dimensional array.

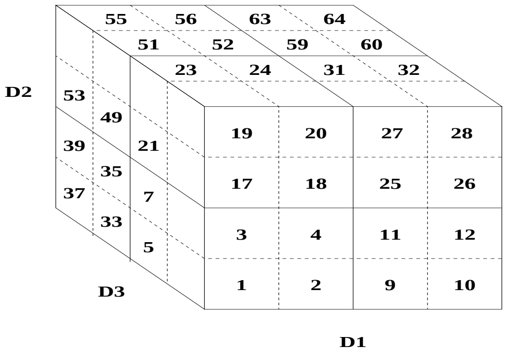


Figure 1: A chunked three-dimensional array with each chunk consisting of $2 \times 2 \times 2$ array cells. The number in each cell indicates its position in the one-dimensional array representation. In this example, cells numbered 1–8 form one chunk, cells 9–16 form the second chunk, and so on. Cells 1–32 form a "hyperplane of chunks" along dimensions D_1 and D_2; cells 33–64 form another hyperplane of chunks.

The size of a d-dimensional chunk is $c_1 \times \cdots \times c_d = B$ where c_i is the chunk size along dimension D_i and B is the size of a disk block. To implement the chunked array, we could use an approach similar to those of [23, 26], in which the disk layout is explicitly managed. For example, the high-level yet efficient TPIE system is used in [23] to do the chunking for matrix multiplication. In this paper, for simplicity, we avoid the need for a separate disk management system like TPIE and use the virtual memory system to our advantage. Instead of forming the chunks and storing them as disk blocks, we form *logical chunks*. We store the multidimensional array as a one-dimensional array, with the order of the array cells specified as in Figure 1. All the cells in the same chunk are put into the same segment of the array. The solid lines are the boundaries between different chunks while the dashed lines separate different cells in the same chunk.

The I/O cost per chunk access is just one I/O, since the virtual memory system will handle the paging. The mapping between the indices of a cell in the logical multidimensional array and the index in the one-dimensional array is very easy to compute as long as the sizes of the dimensions and the size of a chunk along each dimension are known.

Theorem 1 *Consider multidimensional data cube A in chunked form of size $N = \prod_{1 \leq i \leq d} |D_i|$, where $|D_i|$ is the size of dimension D_i and $|D_1| \leq |D_2| \leq \cdots \leq |D_d|$. The total amount of internal memory required to compute the the partial sum data cube P in one linear scan of A is $O(\prod_{1 \leq i \leq d-1} |D_i|)$.*

Proof Sketch: To minimize the memory needed, we use chunks with $c_d = 1$. For $k = 0, 1, \ldots, |D_d| - 1$, we do the following:

1. Read into internal memory the $(d-1)$-dimensional hyperplane for which the value of D_d is k.
2. Compute the partial sums along each of those $d-1$ dimensions.
3. If $k \neq 0$, add the previous hyperplane values to those of the current hyperplane in a cell-by-cell manner along dimension D_d. Then write the resulting hyperplane back to disk. If $k = |D_d| - 1$, also write the current hyperplane back to disk.

□

We define the *dimension order* $(D_{i_1}, \ldots, D_{i_d})$ of the data cube chunks in d dimensions to be the ordering of the chunks that changes most rapidly along the rightmost dimension D_{i_d}, next most rapidly along dimension $D_{i_{d-1}}$, and so on. Different dimension orders correspond to different orders of accessing the data cube chunks. In the example in Figure 1, the chunks are ordered with respect to one another according to the dimension order (D_3, D_2, D_1).[1]

Theorem 2 *In the case of general internal memory size M, we consider a data cube A in chunked form of size $N = \prod_{1\le i\le d} |D_i|$, where $|D_i|$ is the size of dimension D_i. The I/O complexity of computing the partial sum data cube P is $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$, where B is the disk block size.*

Proof Sketch: We partition the d dimensions into g groups so that for the ith group $G_i = \{D_{i_1}, D_{i_2}, \ldots, D_{i_{k_i}}\}$, we have

$$\prod_{1\le j\le k_i} |D_{i_j}| \prod_{\substack{1\le l\le d\\ D_l\notin G_i}} c_l \le M. \tag{2}$$

If (2) cannot be met, that is, if the size of dimension D_{i_1} satisfies $|D_{i_1}|\prod_{l\neq i_1} c_l > M$, we put D_{i_1} into a singleton group containing only itself.

To compute the partial sums, at the ith pass of our algorithm, we read the chunks computed in the previous pass in dimension order $(*, \ldots, *, D_{i_1}, D_{i_2}, \ldots, D_{i_{k_i}})$, where $*$ denotes the dimensions that are not in G_i. If the ith group G_i satisfies (2), we successively read in a k_i-dimensional hyperplane of chunks along dimensions $D_{i_1}, D_{i_2}, \ldots, D_{i_{k_i}}$ (as defined in Figure 1). The size of each such k_i-dimensional hyperplane of chunks is given by the left-hand size of (2), and thus it fits in internal memory. For each hyperplane of chunks, after it is brought into internal memory, we compute the partial sums along each of those dimensions $D_{i_1}, \ldots, D_{i_{k_i}}$ and write the resulting hyperplane of chunks back to disk, chunk by chunk.

For a singleton group G_i, each hyperplane of chunks along dimension D_{i_1} (which because of the one-dimensionality is a "line" of chunks) may not fit in internal memory. However, because of its single dimension, we can read each line of chunks along dimension D_{i_1} and compute its partial sums in one pass using $O(N/B)$ I/Os.

When we are done with the gth pass, we have the partial sum data cube P. By algebraic manipulation we can show that $g = O(\log_{M/B}\frac{N}{B})$. The I/O cost of each of the g passes is $O(\frac{N}{B})$. The desired time bound follows. □

All choices of the chunk sizes c_i yield the upper bound result of Theorem 2. But some choices can do much better than others. Let us consider the example $B = M/2$, $N = \frac{1}{2}M^3$, $d = (\log M) + 1$, $|D_i| = 2$ (for $1 \le i \le d-2$), and $|D_{d-1}| = |D_d| = M$. The smart choice $c_i = 2$ (for $1 \le i \le d-2$) and $c_{d-1} = c_d = 1$ yields $g = 3$, and thus the algorithm runs in $O(N/B)$ I/Os, even though Theorem 1 does not apply. However, the alternate choice $c_i = 1$ (for $1 \le i \le d-1$) and $c_d = M/2 = B$ yields $g = d = (\log M) + 1$, and thus the algorithm runs in $O(\frac{N}{B}\log M)$ I/Os.

2.2. Wavelet Decomposition of the Partial Sum Data Cube P

Wavelets are a mathematical tool for the hierarchical decomposition of functions. Wavelets represent a function in terms of a coarse overall shape, plus details that range from coarse to fine. Regardless of whether the function of interest is an image, a curve, or a surface, wavelets offer an elegant technique for representing the various levels of detail of the function in a space-efficient manner.

The raw data cube A is often sparse in terms of the number of nonzero elements, but the partial sum data cube P tends to be dense. For that reason we confine ourselves in this paper to wavelet decompositions of dense data cubes, especially since the I/O processing is very efficient in terms of the dense representation. (Some possible compression approaches for computing the wavelet decomposition are mentioned in Section 5 and are the subject of continuing work.)

To start the wavelet decomposition procedure, first we need to choose the wavelet basis functions. Haar wavelets are conceptually the simplest wavelet basis functions, and for purposes of exposition in this paper, we focus our discussion on Haar wavelets. They are fastest to compute and easiest to implement.

To illustrate how Haar wavelets work, we start with a simple example. A detailed treatment of wavelets can be found in any standard reference on the subject (e.g., [13, 21]). Suppose we have a one-dimensional "signal":

$$[2,\ 2,\ 7,\ 11].$$

We perform a wavelet transform on it. We first average the signal values, pairwise, to get the new lower-resolution signal with values

$$[2,\ 9].$$

That is, the first two values in the original signal (2 and 2) average to 2, and the second two values 7 and 11 average to 9. Clearly, some information is lost in this averaging process. To recover the original signal from the two averaged values, we need to store some *detail coefficients*, which capture the missing information. Haar wavelets store the pairwise differences of the original values as detail coefficients. In the above example, the two detail coefficients are $2-2=0$ and $11-7=4$. It is easy to see that the original values can be recovered from the averages and differences.

We have succeeded in decomposing the original signal into a lower-resolution version of half the number of entries and a corresponding set of detail coefficients. By repeating this process recursively on the averages, we get the full decomposition:

Resolution	Averages	Detail Coefficients
4	[2, 2, 7, 11]	
2	[2, 9]	[0, 4]
1	$[5\frac{1}{2}]$	[7]

We define the *wavelet transform* (also called *wavelet decomposition*) of the original four-value signal to be the single coefficient representing the overall average of the original signal, followed by the detail coefficients in the order of increasing resolution. Thus, for the one-dimensional Haar basis, the wavelet transform of our original signal is given by

$$[5\tfrac{1}{2},\ 7,\ 0,\ 4]. \tag{3}$$

The individual entries are called the *wavelet coefficients*. The wavelet decomposition is very efficient computationally, requiring only $O(N)$ time and $O(N/B)$ I/Os to compute for a signal of N values.

No information has been gained or lost by this process. The original signal has four values, and so does the transform. Given the transform, we can reconstruct the exact signal by recursively adding and subtracting one half of the detail coefficients from the next-lower resolution.

[1]The definition of dimension order in our paper corresponds to the C programming language array declaration syntax and is different from that of [26].

For compression reasons, the detail coefficients at each level of the recursion are often normalized; the coefficients at the lower resolutions are weighted more heavily than the coefficients at the higher resolutions. One advantage of the normalized wavelet transform is that in many cases a large number of the detail coefficients turn out to be very small in magnitude. Truncating these small coefficients from the representation (i.e., replacing each one by 0) introduces only small errors in the reconstructed signal. We can approximate the original signal effectively by keeping only the most significant coefficients.

The one-dimensional wavelet decomposition and reconstruction procedure can be extended naturally to the multidimensional case. One way to do a multidimensional wavelet decomposition is by a series of one-dimensional decompositions. For example, in the two-dimensional case, we first apply the one-dimensional wavelet transform to each row of the data. Next, we treat these transformed rows as if they were themselves the original data, and we apply the one-dimensional transform to each column.

In terms of the I/O involved, the procedure for doing the multidimensional wavelet decomposition on the partial sum cube P is similar to that of computing A from P, since they both perform certain kinds of operations along each dimension of a d-dimensional array:

Theorem 3 *Consider a multidimensional partial sum data cube P having size $N = \prod_{1\le i\le d} |D_i|$, where $|D_i|$ is the size of dimension D_i and $|D_1| \le |D_2| \le \cdots \le |D_d|$. The total amount of internal memory required to compute the wavelet decomposition of P in one linear scan of P is $O(\prod_{1\le i\le d-1} |D_i|)$.*

Theorem 4 *In the case of general internal memory size M, we consider a multidimensional partial sum data cube P in chunked form of size $N = \prod_{1\le i\le d} |D_i|$, where $|D_i|$ is the size of dimension D_i. The I/O complexity of computing the multidimensional wavelet decomposition of P is $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$, where B is the disk block size.*

The proofs of the above theorems are similar to those of Theorem 1 and Theorem 2, respectively, except that we perform a one-dimensional wavelet decomposition instead of a one-dimensional partial sum operation along each dimension.

2.3. Thresholding and Error Measures

Our motivation in this paper is a compact, yet accurate representation of the partial sum data cube. Given the storage limitation for the compact data cube, we can only "keep" a certain number of the N wavelet coefficients. Let C denote the number of wavelet coefficients that we have room to keep; the remaining wavelet coefficients are implicitly set to 0. Typically we have $C \ll N$. The goal of thresholding is to determine which are the "best" C coefficients to keep, so as to minimize the error of approximation.

We can measure the error of approximation in several ways. Let v_i be the actual answer of a query q_i and let $\widehat{v_i}$ be the approximate answer. We use the following four different error measures for the error e_i of approximating query q_i:

	Notation	Definition
absolute error	e_i^{abs}	$\lvert v_i - \widehat{v_i}\rvert$
relative error	e_i^{rel}	$\dfrac{\lvert v_i - \widehat{v_i}\rvert}{\max\{1, v_i\}}$
modified relative error	$e_i^{\text{m\_rel}}$	$\dfrac{\lvert v_i - \widehat{v_i}\rvert}{\max\{1, \min\{v_i, \widehat{v_i}\}\}}$
combined error	e_i^{comb}	$\min\{\alpha \times e_i^{\text{abs}}, \beta \times e_i^{\text{rel}}\}$

The parameters α and β are positive constants.

Our definition of relative error is slightly different from the traditional one, which is not defined when $v_i = 0$. The modified relative error treats over-approximation and under-approximation in a uniform way. The combined error reflects the importance of having either a good relative error or a good absolute error for each approximation. For example, for very small v_i it may be good enough if the absolute error is small even if the relative error is large, and for large v_i the absolute error may not be as meaningful as the relative error.

Once we choose which of the above measures to represent the errors of the individual queries, we need to choose a norm by which to measure the error of a collection of queries. Let $e = (e_1, e_2, \ldots, e_Q)$ be the vector of errors over a sequence of Q queries. We assume that one of the above four error measures is used for each of the individual query errors e_i. For example, for absolute error, we can write $e_i = e_i^{\text{abs}}$. We define the overall error for the Q queries by one of the following error measures:

	Notation	Definition
1-norm average error	$\lVert e\rVert_1$	$\dfrac{1}{Q} \sum_{1\le i\le Q} e_i$
2-norm average error	$\lVert e\rVert_2$	$\sqrt{\dfrac{1}{Q} \sum_{1\le i\le Q} {e_i}^2}$
infinity-norm error	$\lVert e\rVert_\infty$	$\max_{1\le i\le Q} \{e_i\}$

These error measures are special cases of the p-norm average error, for $p > 0$:

$$\lVert e\rVert_p = \left(\frac{1}{Q} \sum_{1\le i\le Q} {e_i}^p\right)^{1/p}.$$

The first step in thresholding is normalizing the coefficients in a certain way (which corresponds to using a particular basis, such as an orthonormal basis, for example). It is well-known that thresholding by choosing the C largest (in absolute value) wavelet coefficients after normalization is provably optimal in minimizing the 2-norm of the absolute errors, among all possible choices of C nonzero coefficients, assuming that the wavelet basis functions are orthonormal [21]. With proper normalization, which we use, the Haar basis is orthonormal. As a result, normalization and thresholding perform well in practice on other norms of absolute error, such as the p-norms, for $p > 0$. There are no known computationally efficient methods for minimizing these other norms, although some approximation techniques have been studied [5].

In many circumstances, we want to minimize the *relative* error of our approximation. We use the following method to convert a method for achieving good absolute error into a method for achieving good relative error: We take the natural logarithm of each element in P before we do the wavelet decomposition, and we apply the inverse (i.e., exponentiation) after reconstruction in the on-line phase. The intuition for this preprocessing is based on the following fact: If we denote the function we want to approximate by $f(x)$, where x varies over some domain, let us consider its logarithm

$$g(x) = \ln f(x).$$

If we approximate the function g using wavelets with normalization and thresholding, then we can expect that the resulting approximation $\widehat{g}$ has small absolute error. In particular we have

$$\widehat{g}(x) = g(x) + \Delta(x),$$

where $|\Delta(x)|$ is typically small. The approximation $\widehat{f}(x)$ of $f(x)$ can be obtained by

$$\widehat{f}(x) = e^{\widehat{g}(x)} = f(x) \times e^{\Delta(x)}.$$

The relative error $\left|\big(f(x) - \widehat{f}(x)\big)/f(x)\right|$ is thus given by

$$\left|\frac{f(x) - f(x)e^{\Delta(x)}}{f(x)}\right| = \left|1 - e^{\Delta(x)}\right| \approx \left|\Delta(x)\right| + O\big(\Delta(x)^2\big),$$

which is small when $|\Delta(x)|$ is small. The last approximation follows from the Taylor expansion of $e^{\Delta(x)}$ for small $|\Delta(x)|$.

In the wavelet decomposition step, in order to avoid values of $-\infty$ or large negative values, we further modify the partial sum data cube by adding a small constant c to each cell before doing the multidimensional wavelet decomposition. In our experiments, we use $c = 1$. The value c is then subtracted in the on-line phase after the reconstruction is done.

This logarithm transformation has one remarkable property that we discuss further in Section 4: Not only does it dramatically lower the relative error of the approximation in our experiments, it also lowers the absolute error, no matter which norm we use to measure the error. Such a phenomenon does not occur when the logarithm transform is used with histogram methods, such as MaxDiff histogram [18], for example; the MaxDiff relative error shows some improvement (compared with when the logarithm transform is not used), but its absolute error increases substantially.

Using the standard thresholding method, we need to pick the C largest (in absolute value) wavelet coefficients, which can be done in $O(N/B)$ I/Os by using a recursive distribution (or bucketing) method [2]. We can possibly combine the thresholding with the last pass of the wavelet decomposition procedure to further reduce the actual I/O cost.

The C wavelet coefficients together with their C indices (in the one-dimensional order of cells), form the compact data cube. The table storage is thus $2C$ numbers in size. Further compression may be possible by quantization and entropy encoding, but for simplicity we do not consider further improvements in this paper. As a result, our experimental conclusions in Section 4 are conservative.

3. Answering Range Queries in the On-Line Phase

Each range-sum query can be expressed as sums and differences of a certain set of cell values from the multidimensional partial sum data cube P [12]. The set of cells are the ones on the corners of the query hyperplane:

Theorem 5 ([12]) *The answer for the d-dimensional range-sum query*

$$l_1 \leq D_1 \leq h_1 \quad \text{AND} \quad \ldots \quad \text{AND} \quad l_d \leq D_d \leq h_d$$

is

$$v = \sum_{\substack{i_k \in \{l_k - 1, h_k\} \\ \text{for each } 1 \leq k \leq d}} \Big(\prod_{1 \leq k \leq d} s(i_k)\Big) \times P[i_1, i_2, \ldots, i_d], \qquad (4)$$

where

$$s(k) = \begin{cases} 1 & \text{if } i_k = h_k; \\ -1 & \text{if } i_k = l_k - 1. \end{cases}$$

By convention, we define $P[i_1, i_2, \ldots, i_d] = 0$ if $i_j = -1$ for any $1 \leq j \leq d$.

We make use of Theorem 5 to compute our approximation $\widehat{v}$ of the query value v by computing an approximate reconstruction of each needed cell value $P[i_1, i_2, \ldots, i_d]$ in (4). Each reconstruction is based on the inverse wavelet transform of the C wavelet coefficients; the other $N - C$ coefficients are implicitly set to 0.

The time for reconstruction is crucial for the query performance.

Theorem 6 *In a d-dimensional partial sum data cube with dimension sizes $|D_1|, \ldots, |D_d|$, the partial sum value for a given cell can be reconstructed from the C wavelet coefficients using $O(dC)$ space in time $O(\sum_{1 \leq i \leq d} \min\{C, \log |D_i|\})$.*

The proof of this theorem is an extension of the proof for the one-dimensional case [16].

If the C coefficients correspond to one disk block, only one I/O is needed to approximate any or all of the cells of P. The CPU time is $O(C)$ for each cell. By Theorem 5 each range-sum query may require the approximate reconstruction of 2^d cells. However, in typical range-sum queries, where only a few of the dimensions are specified, fewer cells need to be reconstructed and our technique is especially efficient:

Corollary 1 *If only d' of the d dimensions are involved in the query (and the remaining $d - d'$ dimensions are implicitly over their entire domains), we need to reconstruct only $2^{d'}$ cell values in Theorem 5 to answer the range query.*

Proof Sketch: For each of the $d - d'$ dimensions D_j that are completely spanned by the range-sum query, we have $l_j = 0$ and thus $P[i_1, i_2, \ldots, i_d] = 0$ if $i_j = l_j - 1$. Hence, there are only $2^{d'}$ nonzero values of $P[i_1, i_2, \ldots, i_d]$ in the summation in (4). □

4. Experimental Results

In this section we report on some experiments that compare the effectiveness and accuracy of our wavelet-based approximation technique with histogram-based techniques and random sampling.

4.1. Methods Used for Comparison

4.1.1. MaxDiff and Modified MaxDiff Histograms

Histograms approximate the data in one or more attributes of a relation by grouping attribute values into "buckets" and approximating the true attribute values and their frequencies based on summary statistics maintained in each bucket [3]. By replacing the frequencies with the measure attribute values, we can use histograms to approximate a data cube. Since the data cube is a multidimensional array, we concentrate on multidimensional histograms in our discussion.

Muralikrishna and DeWitt [17] use an interesting spatial index partitioning technique for constructing equidepth histograms for multidimensional data. One drawback with this approach is that it considers each dimension only once during the partition. Poosala and Ioannidis [18] propose effective alternatives. Among the new classes of histograms they proposed, the multidimensional MaxDiff(V,A) histograms computed using the MHIST-2 algorithm are most accurate and perform better in practice than previous methods [18]. We compare our wavelet-based compact data cube with this class of histograms, which we shall refer to simply as MaxDiff histograms.

In our experiments, we form MaxDiff histograms to approximate the raw data cube. The range-sum queries are answered in the on-line phase using the approximate data cube represented by the histogram.

As we discussed in Section 2.3, combining a logarithm transformation with the approximation technique can be effective in reducing the relative error of the approximation. Our experiments also test a modified MaxDiff histogram that uses the logarithm transform.

The storage required for each of the b buckets in the histogram is three numbers: one to store the index of the front corner of the bucket (w.r.t. the linear order of the cells), another to store the index of the far corner of the bucket, and a third to store the (average) value associated with the bucket. (Poosala and Ioannidis [18] use $d+1$ numbers to represent each bucket, but we instead use just three numbers per bucket by taking advantage of the one-dimensional order of the cells, as shown in Figure 1.)

4.1.2. Random Sampling

Several random sampling techniques, in which a large set of data is represented by a smaller random sample of the data, have been developed for database optimization [8, 9, 15, 14]. We can approximate the raw data cube by taking a random sample of a certain size from the nonzero cells of the raw data cube. When a range-sum query is presented in the on-line phase, the query is evaluated against the sample, and the approximate answer is extrapolated in the obvious way: If the answer to the query using a sample of size t is s, the approximate answer that will be reported is sT/t, where T is the total number of the nonzero cells in the raw data cube.

To store the samples as a compact data cube, we need to keep the indices of the sampled cells in the linear order, together with the value of the cell. Thus, storing a random sample of size t requires $2t$ numbers.

4.2. Data Description

In [16], a series of experimental results on selectivity estimation in low dimensions is presented that compares the accuracy of the wavelet-based approximation technique with that of histograms and random sampling. The data used in [16] are TPC-D [22] benchmark data and some synthetic data sets generated using Zipf distribution. Those same data sets are used in our experiments for approximating low-dimensional data. We apply our new thresholding method on the wavelet-based histogram technique to show the significant improvement in accuracy.

In many real OLAP applications, the data have high dimension and the correlations among the functional attributes and the measure are intricate and do not match artificial data models. To make our experimental results meaningful, we performed our experiments using real-world data as well as synthetic data. For brevity in this paper, we report our high-dimensional results on only the real-world data.

Our real-world data are obtained from the U.S. Census Bureau databases using their Data Extraction System (DES) [4]. Our data source is the Current Population Survey (CPS) and our extracted file is the March Questionnaire Supplement–Person Data File. This file contains 372 attributes from which we chose 11. Our measure attribute is *income* and the 10 functional attributes are *age, marital status, sex, education_attainment, race, origin, family_type, detailed_household_summary, age group,* and *class_of_worker*. In the original data file, all the attributes are already preprocessed and have a relatively small dimension size; that is, the domain of each dimension D_i is $\{0, 1, \ldots, |D_i|-1\}$, for some small integer $|D_i|$. Although the dimension sizes are generally small, the high dimensionality results in a raw data cube with more than 16 million cells. The density of the raw data cube is about 0.001; there are 15,985 nonzero elements. In this setting we can imagine that several data cubes are approximated, and each data cube must be approximated using very little space.

4.3. Experimental Comparisons of Approximation

We compare the effectiveness of our compact data cube via wavelets with those of MaxDiff histogram and random sampling. Different techniques need to store different types of information. We saw earlier in Sections 2.3 and 4.1 that our wavelet technique needs to store $2C$ numbers to represent C coefficients, MaxDiff needs $3b$ numbers to represent b buckets, and random sampling needs $2t$ numbers to store a sample of size t.

In our experiments, all methods are allowed the same amount of storage. We varied the allowed storage from 400 four-byte numbers to 2000 four-byte numbers. This small storage space corresponds to the practice in OLAP applications of handling several data cubes using a limited amount of storage space. The approximate data cubes can be accessed in the on-line phase in a single I/O.

For example, a space usage of 800 numbers corresponds to keeping $C=400$ wavelet coefficients for our wavelet-based approximation, using $b=267$ buckets for the MaxDiff histogram, and maintaining a random sample of size $t=400$.

We measure the errors of the various approximation techniques in our experiments using five types of query predicates. In each dimension, the range is independently and uniformly generated according to the specified type:

A: $\{D_i \le h_i \mid h_i \in \{0, 1, \ldots, |D_i|-1\}\}$.
B: $\{l_i \le D_i \le h_i \mid l_i, h_i \in \{0, 1, \ldots, |D_i|-1\},\ l_i < h_i\}$.
C: $\{l_i \le D_i \le h_i \mid l_i, h_i \in \{0, 1, \ldots, |D_i|-1\},\ h_i = l_i + \Delta\}$, where Δ is a positive integer constant.
D: $\{D_i = b \mid b \in \{0, 1, \ldots, |D_i|-1\}\}$.
E: $\{D_i = |D_i|-1\}$.

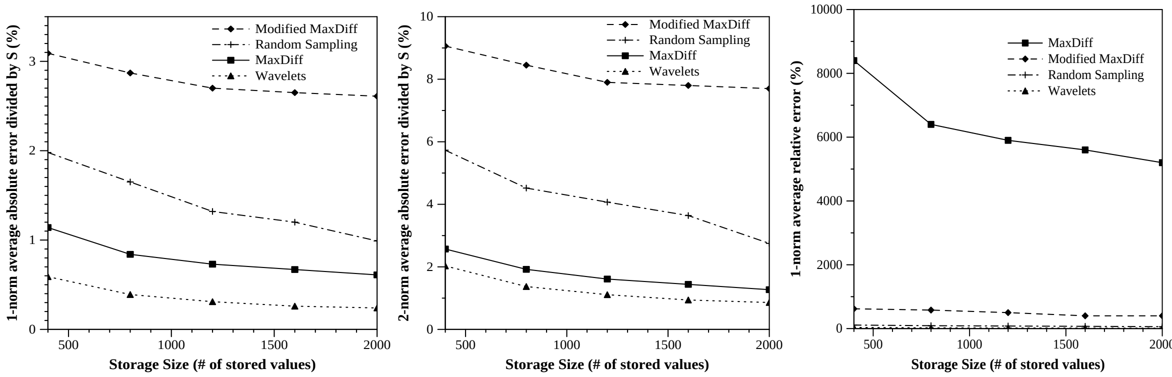
We applied different combinations of these predicates on the set of 10 attributes. Our method is more accurate in approximating most types of on-line queries than the histogram and random sampling techniques, while it is comparable to the histogram method and better than random sampling for other types of queries. We present the results from one experiment of Type A queries. Table 1 gives the detailed comparisons using our wavelet technique, the MaxDiff histograms, and random sampling for a storage size of 800 four-byte numbers. Figure 2 plots the effect of different storage sizes. For random sampling, the errors are the averages over several different runs.

From Table 1 and Figure 2, we can see that our compact wavelet-based data cube is more accurate in approximating on-line queries than the histogram and sampling techniques. As an example, let us consider a range-sum query on income whose exact answer is 10K (dollars). Our compact data cube with storage size 800 may give an answer of 12.2K (with relative error 22%). The average relative error can typically be reduced to about 13% if we increase the storage space from 800 numbers to 2000 numbers.

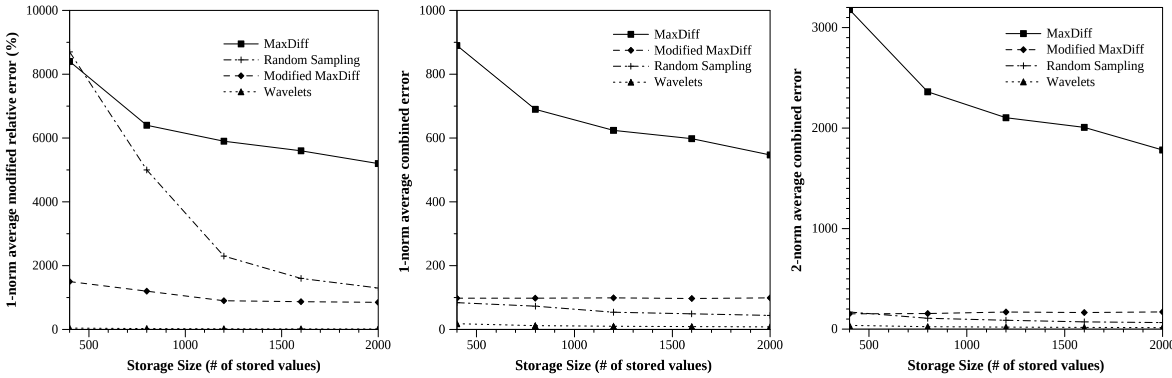
In [16], the wavelet-based histogram method was proposed to approximate low-dimensional data for selectivity estimation. The wavelet-based histograms performed best overall in comparison with MaxDiff histograms and random sampling. In this paper we demonstrate the effectiveness of our new method against that of the previous wavelet-based histograms in [16]. We call our new histograms *new wavelet-based histograms* to distinguish them from the *old wavelet-based histograms* in [16]. The relative effectiveness of the two methods is constant over a wide variety of low-dimensional data sets

Error Norm	*Wavelets*	*MaxDiff*	*Modified MaxDiff*	*Random Sampling*
$\|e^{\mathrm{abs}}\|_1/S$	0.39%	0.84%	2.87%	1.6%
$\|e^{\mathrm{abs}}\|_2/S$	1.37%	1.92%	84.5%	4.5%
$\|e^{\mathrm{rel}}\|_1$	22%	6400%	580%	90%
$\|e^{\mathrm{m\_rel}}\|_1$	27%	6400%	1180%	5000%
$\|e^{\mathrm{comb}}\|_1,\ \alpha=1,\beta=100$	12	690	98	70
$\|e^{\mathrm{comb}}\|_2,\ \alpha=1,\beta=100$	25	2361	155	100
$\|e^{\mathrm{comb}}\|_1,\ \alpha=1,\beta=10$	1.7	232	22	10
$\|e^{\mathrm{comb}}\|_2,\ \alpha=1,\beta=10$	3.1	904	54	10

Table 1: Errors of various methods with storage size 800 on queries of Type A. The absolute errors are normalized by the largest value $S = 907,589$ in the multidimensional partial sum data cube.



(a) 1-norm average absolute error (b) 2-norm average absolute error (c) 1-norm average relative error



(d) 1-norm average modified relative error (e) 1-norm average combined error, $\alpha = 1.0$, $\beta = 100$ (f) 2-norm average combined error, $\alpha = 1.0$, $\beta = 100$

Figure 2: Effect of storage space for various methods on range-sum queries of Type A.

that we used. We present results for Type A queries using the synthetic data described in [16]. Table 2 shows various types of errors for four data sets. In the experiments, all data sets are generated using Zipf distribution with different Zipf parameters. Data sets A and B are one-dimensional data. Their value set size is $n = 500$, the domain size is $N = 4096$, and the relation size is $T = 10^5$. We keep 21 wavelets coefficients in the experiments. Data sets C and D are two-dimensional data. Their value set size is $n = 2500$, the domain size is $N = 65536$, and the relation size is $T = 10^6$. We keep 70 wavelets coefficients in the experiments. The new method dramatically reduces the relative error and combined errors of the approximation, while keeping other error measurements roughly the same.

5. Conclusions

In this paper, we present an I/O-efficient technique based upon a multiresolution wavelet decomposition that yields an approximate and space-efficient representation of the data cube. We build our compact data cube on the logarithms of the partial sums of the raw data values of a multidimensional array. We get excellent approximations for on-line range-sum queries with limited space usage and computational cost. Our new thresholding method of taking the logarithmic transform also provides significant improvement for wavelet-based histograms used in selectivity estimation of low-dimensional data.

One drawback of our current approach is that the con-

Error Norm	Data Set A		Data Set B		Data Set C		Data Set D	
	New Hist.	*Old Hist.*	*New Hist.*	*Old Hist.*	*New Hist.*	*Old Hist.*	*New Hist.*	*Old Hist.*
$\|e^{\text{abs}}\|_1/T$	1.2%	0.8%	2.0%	1.2%	1.0%	1.3%	1.0%	1.5%
$\|e^{\text{abs}}\|_2/T$	3.2%	1.1%	2.5%	1.6%	1.4%	1.7%	1.4%	1.8%
$\|e^{\text{rel}}\|_1$	6.9%	246%	4.6%	8.8%	3.0%	8.3%	5.5%	15%
$\|e^{\text{m-rel}}\|_1$	7.7%	585%	4.9%	9.2%	3.1%	12835%	5.8%	24187%
$\|e^{\text{comb}}\|_1,\ \alpha=1,\beta=100$	6.8	177	4.7	8.1	3.1	8.4	5.5	15.2
$\|e^{\text{comb}}\|_2,\ \alpha=1,\beta=100$	10.5	313	6.5	64	4.1	20.3	10.4	18.6
$\|e^{\text{comb}}\|_1,\ \alpha=1,\beta=10$	0.7	24	0.5	0.9	0.3	0.8	0.8	1.5
$\|e^{\text{comb}}\|_2,\ \alpha=1,\beta=10$	1.1	47	0.7	8.8	0.4	2.0	1.0	1.8

Table 2: Errors of the new wavelet-based histogram and that of the old wavelet-based histogram for query Type A using various low-dimensional data sets.

struction of the wavelet decomposition is performed on the dense data cube, which may be very large. In fact, if the data is sparse and the amount of (nonzero) data is very large, then it may not be feasible to do the wavelet decomposition on the dense data cube in the process of constructing the approximation. To further reduce the I/O cost in constructing the compact data cube, the partial sum data cube can be implicitly represented in a more compressed form, and some sparse techniques may be used to reduce the I/O in computing the wavelet decomposition. We can also do thresholding at each level of the wavelet decomposition so that the sparsity of the data cube is maintained during the course of the construction. We can also extend the advantages of our approach for Type A queries to the other types of queries as well. Results along these important lines of investigation are very promising and will be reported in a coming paper.

Other ongoing work deals with normalization and thresholding methods based upon more sophisticated probability distributions of query patterns. To get further improvements in the space-accuracy tradeoff, we are working on quantizing the wavelet coefficients and entropy encoding of the quantized coefficients. We are developing dynamic efficient algorithms for maintaining the compact data cube, given updates in the underlying raw data cube. We are also investigating aggregations other than *sum*, such as *min* and *max*.

References

[1] A. Agarwal et al. On the computation of multidimensional aggregates. In *Proceedings of the 1996 International Conference on Very Large Databases*, Mumbai, India, 1996.

[2] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9), 1116–1127, 1988.

[3] D. Barbara et al. The New Jersey data reduction report. *Bulletin of the Technical Committee on Data Engineering*, 20(4), 1997.

[4] C. B. Databases. http://www.census.gov/.

[5] D. L. Donoho. Unconditional bases are optimal bases for data compression and statistical estimation. Technical report, Department of Statistics, Stanford University, 1992.

[6] P. B. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, Seattle, Washington, June 1998.

[7] J. Gray et al. Data cube: A relational aggregation operator generalizing group-by, cross-tabs and subtotals. In *Proceedings of the 12th Annual IEEE Conference on Data Engineering (ICDE '96)*, 131–139, 1996.

[8] P. Haas and A. Swami. Sequential sampling procedures for query size estimation. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, 1992.

[9] P. Haas and A. Swami. Sampling-based selectivity for joins using augmented frequent value statistics. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, March 1995.

[10] V. Harinarayan et al. Implementing data cubes efficiently. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, Montreal, May 1996.

[11] J. M. Hellerstein et al. Online aggregation. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, May 1997.

[12] C.-T. Ho et al. Range queries in OLAP data cubes. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, May 1997.

[13] B. Jawerth and W. Sweldens. An overview of wavelet based multiresolution analyses. *SIAM Rev.*, 36(3), 377–412, 1994.

[14] R. Lipton and J. Naughton. Query size estimation by adaptive sampling. *J. of Comput. Sys. Sci.*, 51, 18–25, 1985.

[15] R. Lipton, J. Naughton, and D. Schneider. Practical selectivity estimation through adaptive sampling. In *Proceeding of the 1990 ACM SIGMOD International Conference on Management of Data*, 1–11, 1990.

[16] Y. Matias, J. S. Vitter, and M. Wang. Wavelet-based histograms for selectivity estimation. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, Seattle, Washington, June 1998.

[17] M. Muralikrishna and D. J. DeWitt. Equi-depth histograms for estimating selectivity factors for multi-dimensional queries. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, 28–36, 1988.

[18] V. Poosala and Y. E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *Proceedings of the 1997 International Conference on Very Large Databases*, Athens, Greece, August 1997.

[19] S. Sarawagi and M. Stonebraker. Efficient organization of large multidimensional arrays. In *Proceedings of the 11th Annual IEEE Conference on Data Engineering (ICDE '94)*, Houston, Texas, 1994.

[20] J. E. Savage and J. S. Vitter. Parallelism in space-time tradeoffs. In F. P. Preparata, editor, *Advances in Computing Research, Volume 4*, 117–146. JAI Press, 1987.

[21] E. J. Stollnitz, T. D. Derose, and D. H. Salesin. *Wavelets for Computer Graphics.* Morgan Kaufmann, 1996.

[22] TPC benchmark D (decision support), 1995.

[23] D. E. Vengroff and J. S. Vitter. I/O-efficient scientific computation using TPIE. In *Proceedings of the Goddard Conference on Mass Storage Systems and Technologies*, 553–570, College Park, MD, September 1996.

[24] J. S. Vitter. External memory algorithms. In *Proceedings of the 1998 ACM Symposium on Principles of Database Systems*, June 1998. Invited tutorial.

[25] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2-3), 110–147, 1994. Special double issue on Large-Scale Memories.

[26] Y. Zhao, P. M. Deshpande, and J. F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, May 1997.