Approximate Computation of Multidimensional Aggregates of Sparse Data Using Wavelets

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

Abstract

Computing multidimensional aggregates in high dimensions is a performance bottleneck for many OLAP applications. Obtaining the exact answer to an aggregation query can be prohibitively expensive in terms of time and/or storage space in a data warehouse environment. It is advantageous to have fast, approximate answers to OLAP aggregation queries.

In this paper, we present a novel method that provides approximate answers to high-dimensional OLAP aggregation queries in massive sparse data sets in a time-efficient and space-efficient manner. We construct a *compact data cube*, which is an approximate and space-efficient representation of the underlying multidimensional array, based upon a multiresolution wavelet decomposition. In the on-line phase, each aggregation query can generally be answered using the compact data cube in one I/O or a small number of I/Os, depending upon the desired accuracy.

We present two I/O-efficient algorithms to construct the compact data cube for the important case of *sparse high-dimensional arrays*, which often arise in practice. The traditional histogram methods are infeasible for the massive high-dimensional data sets in OLAP applications. Previously developed wavelet techniques are efficient only for dense data. Our on-line query processing algorithm is very fast and capable of refining answers as the user demands more accuracy. Experiments on real data show that our method provides significantly more accurate results for typical OLAP aggregation queries than other efficient approximation techniques such as random sampling.

1. Introduction

Computing multiple related group-bys and aggregates is one of the core operations in On-Line Analytical Processing (OLAP) applications. A particular characteristic of the data sets—and the primary concern of this paper—is that they are *massive* and *sparse*.

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 underlying data by a d-dimensional array S of size $|D_1| \times |D_2| \times \cdots \times |D_d|$, where $|D_i|$ is the size of dimension D_i. Without loss of generality, we assume that each dimension D_i has an index domain $\{0, 1, \ldots, |D_i| - 1\}$. For convenience, we call each array element a *cell*. A cell contains $S(i_1, i_2, \ldots, i_d)$, the value of the *measure attribute* for the corresponding combination $(i_1, i_2, \ldots, i_d)$ of the functional attributes. We let $N = \prod_{1 \leq i < d} |D_i|$ denote the total size (i.e., number of cells) of array S, and we define N_z to be the number of populated (nonzero) entries in S. We also refer to N_z as the *size of the raw data*. The *density* of array S is defined as

$$density(S) = \frac{N_z}{N}. \tag{1}$$

The sparse (ROLAP) representation of S is

$$\left\{ (i_1, i_2, \ldots, i_d, S(i_1, i_2, \ldots, i_d)) \ \big| \ S(i_1, i_2, \ldots, i_d) \neq 0 \right\} \tag{2}$$

and is used extensively in practice for sparse data.

An important class of aggregation queries are the so-called (general) *range-sum* queries, which are defined by applying the *Sum* operation over a selected contiguous range in the domains of some of the attributes [HAMS97]. A range-sum query can generally be formulated as follows:

$$Sum(l_1 : h_1, \ldots, l_d : h_d) = \sum_{l_1 \leq i_1 \leq h_1} \cdots \sum_{l_d \leq i_d \leq h_d} S(i_1, \ldots, i_d).$$

An interesting subset of the general range-sum queries are d'-*dimensional range-sum queries* in which $d' \ll d$. Ranges are explicitly specified for only d' dimensions, and the ranges for the other $d - d'$ dimensions are implicitly set to be the entire domain $all_i = \{0, \ldots, |D_i| - 1\}$. The d' dimensions with explicit ranges can be any subset of the d dimensions and can vary from query to query. For simplicity in notation, for any given query, let us write the d' dimensions first so that the d' explicit ranges are on dimensions $D_1, D_2, \ldots, D_{d'}$ and the last $d - d'$ dimensions have implicitly defined ranges over the entire domain (i.e., $l_j = 0$ and $h_j = |D_j| - 1$, for $d' + 1 \leq j \leq d$). Using the above notation, these queries have the form $Sum(l_1 : h_1, \ldots, l_{d'} : h_{d'}, all_{d'+1}, \ldots, all_d)$. For brevity, we simply write $Sum(l_1 : h_1, \ldots, l_{d'} : h_{d'})$.

The popular *data cube* operator [GBLP96] can be viewed as computing the special case of all range-sums with singleton ranges, $Sum(l_1:h_1, \ldots, l_{d'}:h_{d'})$, in which $0 \le l_i = h_i < |D_i|$, for $1 \le i \le d'$. In traditional approaches of answering range-sum queries using data cube, all the subcubes of the data cube need to be precomputed and stored. When a query is given, a search is conducted in the data cube and relevant information is fetched. The search results may or may not need to be further processed, depending upon the type of the given query, and a final *exact* answer is output.

In this paper we take a different approach. As usual, we preprocess the original array S. But instead of computing and storing all the subcubes, we compute and store *one* much smaller *compact data cube*, which usually fits in one or a small number of disk blocks, depending upon the desired accuracy for the queries. In the on-line phase, for any given query, the compact data cube is consulted to give an approximate answer.

Our approach is preferable to the traditional approaches in two important respects. First of all, the traditional approaches require a huge amount of storage space for both the precomputation and the storage of the precomputed data cube. As we all know, the size of the precomputed data cube is much larger than that of the underlying raw data, especially when S is high-dimensional (e.g., more than six dimensions) [PC98]. In some applications there may be 100 dimensions! Even in moderately sized scenarios, there are usually many tables (in a ROLAP system) or multidimensional arrays (in a MOLAP system), and most of them are already very large in size by themselves. Since a query may be issued against any table (array), we have to compute and store one data cube for each of them. This fact would easily make the task infeasible even for moderately sized applications. Our approach in this paper does not have the storage space problem. The size of each compact data cube is very small.

Secondly, even when a huge amount of storage space is available and all data cubes can be stored comfortably, it may take too long to answer a range-sum query, since all cells covered by the range need to be accessed. The problem persists even if the partial-sum technique is used (see Section 3). However, our new approach does not have this problem at all. A query can be answered by retrieving the compact data cube, which in typical cases takes just one or a small number of I/Os.

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 [HHW97]. Another consideration is that sometimes the base data are remote and unavailable, so that an exact answer is not an option until the data again become available [FJS97].

In developing our wavelet-based techniques to approximately answer OLAP range-sum queries, we resolve the following four important issues in this paper:
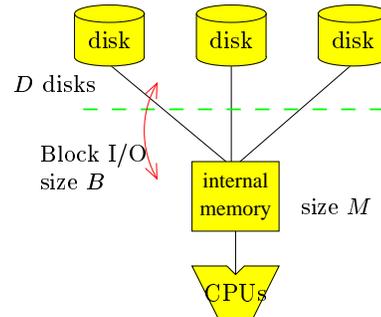
1. *I/O-efficiency* of the compact data cube construction, especially when the underlying multidimensional array is very sparse. Our earlier wavelet approach [VWI98] requires a dense storage representation during the construction of the compact data cube, which is infeasible for very large sparse data sets. Histogram techniques [PI97, PIHS96] usually require excessive I/O costs when the data size is large and the dimensionality is high. Our new wavelet approach is fast and space-efficient even for massive sparse data.

2. *Response time* in answering an on-line query. Generally one or a small number of I/Os suffice, and the CPU time is small.

3. *Accuracy* in answering typical OLAP queries. The performance of the algorithm is generally superior to that of random sampling.

4. *Progressive refinement* of the approximate answers in case more accuracy is desired.

In the next two sections we describe our model of I/O performance and summarize previous work on the problem. We describe our new wavelet approach in Section 4. The details of the construction process are given in Section 5. We show how to process on-line queries in Section 6. We present our experimental results in Section 7 and draw conclusions in Section 8.

2. I/O Model

We use the conventional *parallel disk model*, popularized in [VS94, Vit99]:

The parameters (in units of items) are

$$
\begin{aligned}
M &= \text{size of internal memory;} \\
B &= \text{size of disk block;} \\
I &= \text{number of independent disks.}
\end{aligned}
$$

Data are transferred in large units of *blocks* of size B so as to amortize the latency of moving the read-write head and waiting for the disk to spin into position. For brevity in this paper, we restrict our attention to the case $I = 1$ of only one disk, but our results can be extended to the case $I > 1$ of parallel disks; the I/O results are improved by a factor of I.

3. Previous Work

There are two classes of methods for processing OLAP queries: exact methods and approximate methods. Most previous work has concentrated on how to compute the exact data cube [AAD$^+$96, GBLP96, HRU96, ZDN97]. Ho et al. [HAMS97] present an efficient algorithm to speed up range-sum queries on a single data cube. The main idea is to preprocess the array S and precompute all the multidimensional partial sums, which can be represented in what we call the *partial-sum data cube* P. Any d'-dimensional range-sum query can be answered by accessing and computing $2^{d'}$ entries from P.

The biggest problem with this partial-sum approach is that the partial-sum data cube is typically very dense even when the original array is sparse. For sparse data, the partial-sum approach becomes very expensive since it takes huge

amount of space to store the partial-sum data cube P and the sparse representation does not help us at all. To answer a d'-dimensional range query, $2^{d'}$ entries need to be accessed. The corresponding $2^{d'}$ partial-sum values for a given range-sum query might be stored in different disk blocks and accessing them may require up to $2^{d'}$ disk I/Os, which is prohibitive.

Approximation methods are becoming attractive in OLAP applications [HHW97, GM98, VWI98]. They have been used in DBMSs for a long time. For example, selectivity estimation in query optimization is always an approximation process [SAC+79, PIHS96, MVW98]. In choosing proper approximation techniques, there are two major concerns: the efficiency in applying the techniques and the accuracy of the methods.

Histograms and sampling are used in a variety of important applications where quick approximations of a (possibly multidimensional) array of values are needed, such as query optimization [SAC+79], parallel join load balancing [PI96], and approximate query processors [BDF+97]. Matias et al. [MVW98] first explored the use of wavelet-based techniques to construct analogs of histograms in databases. Their experiments show that wavelet-based approximation methods can offer substantial improvements in accuracy over random sampling and other histogram-based approaches. Traditional histograms are too inefficient to construct when the underlying data are high-dimensional and cannot fit in internal memory. (It is interesting that our method in this paper constructs compact data cubes that can be viewed as some sort of general histograms of the underlying raw data.)

Random sampling is a simple and natural way to answer aggregation queries approximately. An advantage of sampling is that the construction procedure is very efficient to run. (In Section 7, we compare our new method with random sampling.)

Vitter et al. [VWI98] give the first algorithm for approximating the OLAP data cube, based upon the wavelet approach developed in [MVW98]. Using the partial-sum data cube [HAMS97], the algorithm computes an approximate version, called a *compact partial-sum data cube*, using wavelet techniques. The algorithm performs a series of linear scans, in which each scan is done over a carefully selected group of dimensions. A nice provable property is that the number of scans is $O(\log_{M/B} \frac{N}{B})$, and thus the total wavelet decomposition takes $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os, where N is the size (number of cells) of the partial-sum data cube. The algorithm's I/O performance is optimal for dense data cubes, and the resulting compact data cube can generally fit in only one or two disk blocks.

The critical problem with the above approach is that in typical OLAP applications, the data are massive and yet at the same time very sparse, that is, the number N_z of nonzero cells in the array is much smaller than N, perhaps by a factor of several million. Since the partial-sum data cube is typically dense regardless of the sparsity of the original data cube, the method of Vitter et al. [VWI98] may have to process a data set that is several million times larger than the original data set and there simply may not be enough disk space or time to process the dense partial-sum data cube.

4. Our Method: A High-Level Outline

In this section we summarize our basic method. We elaborate on the details in the following sections. Our method has three main components:

1. *Decomposition.* We compute the wavelet decomposition of the multidimensional array S, obtaining a set of C' wavelet coefficients, where C' is roughly equal to the number N_z of nonzero coefficients. We assume as in practice that the array is very sparse, that is, $N_z \ll N$. (The dense case was covered previously in [VWI98].) We use sparse techniques to do the wavelet decomposition directly based upon the sparse (ROLAP) representation of S. In Section 5.2, we give the details of the our efficient wavelet decomposition algorithms and analyze their I/O performance.

2. *Thresholding and Ranking.* We keep only C wavelet coefficients, for some $C \ll C'$ 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. We order (rank) the C wavelet coefficients according to their importance in the context of accurately answering typical aggregation queries. The C ordered coefficients compose our compact data cube. The issue of how to choose proper thresholding method and how to define the (relative) importance of a wavelet coefficient is the key for the accuracy of our approximation method and will be addressed in Section 5.3.

3. *Reconstruction.* In the on-line phase, an aggregation query is processed by using the k most significant wavelet coefficients, for some $k \leq C$, to reconstruct an approximate answer. The choice of k depends upon the time the user is willing to spend. More accurate answers can be provided upon request by using more coefficients to refine the previous approximations. The efficiency of the reconstruction step, in terms of both I/O performance and CPU time, is crucial, since it affects the query response time directly. We give our efficient query answering algorithm in Section 6.

5. Constructing the Compact Data Cube

5.1. Wavelet Decomposition

Wavelets are a mathematical tool for the hierarchical decomposition of functions in a space-efficient manner. 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.

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 which will be used throughout the paper. (A detailed treatment of wavelets can be found in any standard reference on the subject, e.g., [JS94, SDS96].) Suppose we have a one-dimensional "signal" of $N = 8$ data items:

$$S = [2, 2, 0, 2, 3, 5, 4, 4].$$

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

$$[2, 1, 4, 4].$$

That is, the first two values in the original signal (2 and 2) average to 2, and the second two values 0 and 2 average to 1,

and so on. Clearly, some information is lost in this averaging process. To recover the original signal from the four averaged values, we need to store some *detail coefficients*, which capture the missing information. Haar wavelets store one half of the pairwise differences of the original values as detail coefficients. In the above example, the four detail coefficients are $(2-2)/2 = 0$, $(0-2)/2 = -1$, $(3-5)/2 = -1$, and $(4-4)/2 = 0$. 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
8	$[2, 2, 0, 2, 3, 5, 4, 4]$	
4	$[2, 1, 4, 4]$	$[0, -1, -1, 0]$
2	$[1\frac{1}{2}, 4]$	$[\frac{1}{2}, 0]$
1	$[2\frac{3}{4}]$	$[-1\frac{1}{4}]$

We define the *wavelet transform* (also called *wavelet decomposition*) of the original eight-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

$$\widehat{S} = [2\tfrac{3}{4}, -1\tfrac{1}{4}, \tfrac{1}{2}, 0, 0, -1, -1, 0]. \qquad (3)$$

The individual entries are called the *wavelet coefficients*. The wavelet decomposition is very efficient computationally, requiring only $O(N)$ CPU 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 eight values, and so does the transform. Given the transform, we can reconstruct the exact signal by recursively adding and subtracting the detail coefficients from the next-lower resolution.

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.

5.2. Building the Compact Data Cube

The goal of this step is to compute the wavelet decomposition of the multidimensional array S, obtaining a set of C' wavelet coefficients. In this section, we present two algorithms to deal with the difficult and important case in which the underlying data are very sparse. (Dense data can been handled using the algorithm discussed in [VWI98].)

Our algorithms takes the sparse representation (2) of array S as input, which we assume is in *dimension order* $\langle D_1, \ldots, D_d \rangle$; that is, the indices of the entries change most rapidly along the rightmost dimension D_d, next most rapidly along dimension D_{d-1}, and so on.[1] The array entries for which the values in the initial set of dimensions D_1, ..., D_k are fixed form a $(d-k)$-dimensional hyperplane, which we denote by $\langle D_{k+1}, \ldots, D_d \rangle$. Without loss of generality, we assume that D_d is the dimension with the smallest domain size, which improves performance in practice.

5.2.1. ALGORITHM I: DECOMPOSITION WITH SEPARATE TRANSPOSITION STEP

We use a concrete example to illustrate the compact data cube construction process. Suppose S is a three-dimensional array for which $|D_2| \times |D_3| \leq M - 2B$, but $|D_1| \times |D_2| \times |D_3| > M - 2B$. We do the wavelet decomposition in two passes. We partition the three dimensions into two groups: $\{D_1\}$ and $\{D_2, D_3\}$. All tuples with a fixed dimension D_1 value are contiguous in the input S and form a $\langle D_2, D_3 \rangle$ hyperplane. The first pass is done by reading in all $\langle D_2, D_3 \rangle$ hyperplanes, one by one. Each hyperplane is guaranteed to fit in internal memory. An ordinary two-dimensional wavelet decomposition is performed on each hyperplane and the result, still using the sparse representation, is written out using an output double buffer. After all $\langle D_2, D_3 \rangle$ hyperplanes have been processed, we obtain an intermediate array S', which is the result of applying the wavelet decomposition to S along dimensions D_2 and D_3. The elements of array S' are still stored in the dimension order $\langle D_1, D_2, D_3 \rangle$. We reorganize them so that they are stored according to the dimension order $\langle D_2, D_3, D_1 \rangle$. We then do the wavelet decomposition of S' along D_1. We scan S' and read in the $\langle D_1 \rangle$ hyperplanes, one by one, and an ordinary multidimensional wavelet decomposition is performed on each of them (in this particular example, a one-dimensional decomposition). The output of this pass constitutes the final result of the algorithm.

In general, we partition the d dimensions into g groups, for some $1 \leq g \leq d$. Let the jth group be $G_j = \{D_{i_{j-1}+1}, D_{i_{j-1}+2}, \ldots, D_{i_j}\}$, where $i_0 = 0$ and $i_g = d$. The requirement that G_j must satisfy is that either

$$\left| D_{i_{j-1}+1} \right| \times \left| D_{i_{j-1}+2} \right| \times \cdots \times \left| D_{i_j} \right| \leq M - 2B \qquad (4)$$

or else G_j is a singleton group (i.e., $i_{j-1} + 1 = i_j$). We can form the groups one by one in a greedy manner: Given groups $G_1, \ldots, G_{j-1}$, we choose i_j to be the largest integer in the range $(i_{j-1}, d]$ such that (4) still holds, or else $i_j = i_{j-1} + 1$.

Algorithm I for constructing the compact data cube consists of g passes. The groups are processed in reverse order, one per pass. In the $(g-j+1)$st pass, each hyperplane in the jth group G_j is read in, one by one, and processed (i.e., the ordinary multidimensional wavelet decomposition is performed), and the results are written out to be used for the next pass.

One problem is that the density of the intermediate results will increase from pass to pass, since performing wavelet decomposition on sparse data usually results in more nonzero coefficients. The number of nonzero coefficients can increase by a factor of $\log |D_i|$ when doing the wavelet decomposition along dimension D_i. We may thus have to process more and more entries from pass to pass, even though a lot of entries are very small in magnitude. The natural solution to this problem is truncation. In each pass, after obtaining the

---

[1] Our definition of dimension order corresponds to the C programming language array declaration syntax.

intermediate array S', we truncate S' by cutting off entries with small magnitude and keep roughly only N_z entries. We then use the truncated S' as the input for next pass. This process keeps the sparsity of all the intermediate results unchanged. The truncation operation is reasonable because it is in line with the wavelet decomposition method itself; that is the most significant wavelet coefficients contribute the most to the reconstruction of the original signal.

However, it is too expensive to do truncation after all intermediate entries are written out in a multipass process. Instead, we do truncation in each pass via an on-line learning process. We keep on-line statistics of the distribution of the values of the intermediate wavelet coefficients during each pass and dynamically maintain a *cutoff value*. Any entry with its absolute value below the cutoff value will be thrown away on the fly. The cutoff value is adjusted periodically when more coefficients are generated and the statistics change. For example, if too many entries have been cut off, the cutoff value will be decreased. On the other hand, if too few entries have been thrown away, we need to increase the cutoff value. This self-adjusting procedure works well in practice.

After the $(g-j+1)$st pass, for each $1 < j \le g$, in which G_j is processed, a transposition is performed on the output of the pass in order to regroup the cells according to the dimension order required by the next pass. (There is no need to do a transposition after the last pass, namely, when $j = 1$.) The data can be transposed in $\log_{M/B}(|D_{i_{j-1}+1}| \times |D_{i_{j-1}+2}| \times \cdots \times |D_{i_j}|)$ distribution passes, based upon the values of the indices in G_j, and thus the number of I/Os is

$$O\left(\frac{N_z}{B} \log_{M/B}\left(\left|D_{i_{j-1}+1}\right| \times \left|D_{i_{j-1}+2}\right| \times \cdots \times \left|D_{i_j}\right|\right)\right). \tag{5}$$

The first distribution pass of each transposition can be done during the wavelet decomposition procedure (at the cost of reserving half the internal memory for buffer space), which speeds up performance in practice.

After all g passes are done, we obtain the final wavelet decomposition, which consists of $C' \approx N_z$ coefficients. (In the next section we describe the final thresholding process to reduce the number of coefficients from C' to C.) We denote the value of a coefficient by v. Each coefficient, with its index, is of the form

$$c = (i_1, \ldots, i_d, v). \tag{6}$$

The method described above for how the groups are formed is rather conservative and is related to how dense arrays are processed in [VWI98]. The following result follows from (5) with little algebra:

Theorem 1 *For internal memory of size M and block size B, we consider an array S of size $N = \prod_{1 \le i \le d} |D_i|$, where $|D_i|$ is the size of dimension D_i, having a total of N_z nonzero entries. The I/O complexity of Algorithm I (using the truncation procedure) is*

$$O\left(\frac{N_z}{B} \log_{M/B} \frac{N}{B}\right).$$

In practice we can do better than the conservative bound in Theorem 1 by using Algorithm I with a more liberal group partitioning and a smaller number of groups. For example, we might want relax condition (4) and partition the dimensions so that the jth group satisfies

$$density(S) \times \left|D_{i_{j-1}+1}\right| \times \left|D_{i_{j-1}+2}\right| \times \cdots \times \left|D_{i_j}\right| \le \frac{M - 2B}{2}, \tag{7}$$

for $1 \le j \le g$. The value of $density(S) = N_z/N$ is typically a very small fraction in practice. The new partition results in a much smaller value of g than the one in Theorem 1. Often we get $g = 2$, and thus only two passes (and one intermediate transposition) are needed. However, it may no longer be desirable to do the transposition via the distribution approach of (5); instead we can do the transposition by sorting, which uses $O(\frac{N_z}{B} \log_{M/B} \frac{N_z}{B})$ I/Os. (See [Vit99] for a proof in the I/O model that transposition is equivalent to sorting.) If all the processed hyperplanes individually fit into internal memory, the resulting I/O bound for Algorithm I will be

$$O\left(\frac{N_z}{B} \log_{M/B} \frac{N_z}{B}\right), \tag{8}$$

which is optimal.

The tradeoff for the liberal partitioning strategy is that from time to time, certain hyperplanes may not fit in internal memory and their wavelet decomposition may require multiple passes. But the number of such hyperplanes requiring extra time is usually small, and the recomputation is localized to the hyperplanes. Overall we can get great savings in Algorithm I by using a smaller g value, as our experiments indicate in Section 7. We use (7) as a guideline for determining the groups, and we find that $g = 2$ as long as

$$M \ge 2\sqrt{density(S) \times N_z} + 2B.$$

5.2.2. Algorithm II: Decomposition without Separate Transposition Step

One problem with Algorithm I when using a conservative group partitioning is that we need to perform a transposition operation to reorder the array entries between passes, for example, after processing one group and before proceeding to the next.

In this section, we present another decomposition algorithm, called Algorithm II, that uses buffering and knowledge of the domain sizes to avoid an explicit transposition step between passes. The tradeoff is that Algorithm II must use a conservative group partition, and thus may require more passes g than in Algorithm I. The input to the algorithm is the sparse representation of array S. We assume the input is in dimension order $\langle D_1, \ldots, D_d \rangle$. As before, we partition the d dimensions into g groups, for some $1 \le g \le d$, in a greedy fashion. Let the jth group be $G_j = \{D_{i_{j-1}+1}, D_{i_{j-1}+2}, \ldots, D_{i_j}\}$, such that either

$$\left|D_{i_{j-1}+1}\right| \times \left|D_{i_{j-1}+2}\right| \times \cdots \times \left|D_{i_j}\right| \le \frac{M}{2B+1}, \tag{9}$$

or if $j = 1$ then

$$|D_1| \times |D_2| \times \cdots \times \left|D_{i_j}\right| \le M - 2B, \tag{10}$$

or else G_j is a singleton group (i.e., $i_{j-1} + 1 = i_j$).

The algorithm consists of g passes. We process the g groups in reverse order, one per pass. Let us illustrate the process with the following concrete example. Suppose S is a six-dimensional array and we partition the six dimensions according to the above procedure. Let the partition be $\{D_1, D_2\}$, $\{D_3, D_4\}$, and $\{D_5, D_6\}$.

At the beginning of the first pass, we reserve two types of buffers in internal memory: a *processing buffer* and *output buffers*. We have one processing buffer whose size is $|D_5| \times |D_6|$. We have $|D_5| \times |D_6|$ output double buffers, each of size $2B$. Each output buffer has a unique $b\_id \in \{0, 1, \ldots, |D_5| \times |D_6| - 1\}$.

We then read in all $\langle D_5, D_6 \rangle$ hyperplanes, one by one, into the processing buffer, and perform the ordinary multidimensional wavelet decomposition. The results of the decomposition are then subjected to the cutoff value. For those coefficients whose magnitudes are bigger than the cutoff value,

we do not write them to disk. Instead, for a coefficient $c = (i_1, \ldots, i_6, v)$, we write it into the output buffer with $b\_id = i_5 \times |D_6| + i_6$. When half of an output double buffer becomes full, we write its data to disk.

After we are done with all the $\langle D_5, D_6 \rangle$ hyperplanes, we are finished with the first pass.

In the second pass, we read into the processing buffer the blocks created during the previous pass, in the order of increasing b_id value (and for each b_id, in the order the blocks were created). The important observation here is that the resulting order is the dimension order $\langle D_5, D_6, D_1, D_2, D_3, D_4 \rangle$, which is needed for doing the $\langle D_3, D_4 \rangle$ hyperplane decompositions, and thus we avoid the need for a separate transposition step. The transposition is done for free as a result of the buffering mechanism. We can then process similarly as in the previous pass, except that now the number of output buffers becomes $|D_3| \times |D_4|$ and the size of processing buffer becomes $|D_3| \times |D_4|$.

When performing the decomposition for the dimensions in the last pass (when processing G_1), we no longer need the output buffers, and we can write the decomposition results out directly.

In the previous example, all the individual dimension sizes satisfied the condition

$$|D_i| \leq \frac{M}{2B+1}, \qquad (11)$$

except possibly for D_1. We call D_i a *big dimension* if its size does not satisfy (11). All big dimensions form singleton groups. So far we have not described how to process big dimensions. If there is only one big dimension, namely, D_1, then we can perform the wavelet decomposition along D_1 in a linear pass since there is no need for transposing the data via the output buffers.

However, when there are multiple big dimensions, Algorithm II as described above no longer works (except for D_1). Let us suppose that D_i is a big dimension, for some $i \neq 1$. The simplest approach is to use the technique of Algorithm I, in which the wavelet decomposition is computed in $O(N_z/B)$ I/Os, followed by a distribution-based transpose operation, which takes $O(\frac{N_z}{B} \log_{M/B} |D_i|)$ I/Os. The first level of the distribution can be incorporated into the pass that does the wavelet decomposition, yielding an improvement in practice.

Putting everything together, we find that the total wavelet decomposition of array S requires $O(\log_{M/B} \frac{N}{B})$ passes over the data, each pass using $O(N_z/B)$ I/Os, and we get the following result:

Theorem 2 *For internal memory of size M and block size B, we consider an array S of size $N = \prod_{1 \leq i \leq d} |D_i|$, where $|D_i|$ is the size of dimension D_i, having a total of N_z nonzero entries. The number of I/Os needed for the wavelet decomposition of S using Algorithm II is*

$$O\left(\frac{N_z}{B} \log_{M/B} \frac{N}{B} \right).$$

The I/O bounds in Theorems 1 and 2 are a tremendous improvement over the bound obtained by Vitter et al. [VWI98], which is larger by a multiplicative factor of $1/density(S) = O(N/N_z)$. In practice, the approach of Algorithm I is generally preferable, since it can accommodate a more liberal group partitioning strategy, which often results in a much smaller g value, typically $g = 2$, and the optimal I/O bound of (8).

5.3. Thresholding and Ranking

Given the storage limitation for the compact data cube, we can only "keep" a certain number of the C' wavelet coefficients. Let C denote the number of wavelet coefficients that we have room to keep; the remaining wavelet coefficients will be implicitly set to 0. Typically we have $C \ll C'$, so that the C coefficients can fit into one or a few disk blocks. 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 of the query. We use the following five different error measures for the error e_i of approximating query q_i:

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

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. For example, suppose the exact answer to a query is $v_i = 10$. The approximate answers $\widehat{v_i} = 5$ or $\widehat{v_i} = 20$ each have the same modified relative error, namely, $5/5 = 10/10 = 1$. In contrast, in terms of relative error, the approximation $\widehat{v_i} = 5$ has a relative error of $5/10 = 0.5$, and the approximation $\widehat{v_i} = 20$ has a relative error of $10/10 = 1$. The approximation $\widehat{v_i} = 0$ (which is a terrible approximation for OLAP purposes) has a relative error of only $10/10 = 1$, while the modified relative error is $10/1 = 10$. 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 $e_i = e_i^{\mathrm{abs}}$. We define the overall error for the Q queries by one of the following error measures:

	Notation	Definition
1-norm average error	$\|e\|_1$	$\dfrac{1}{Q} \sum_{1 \leq i \leq Q} e_i$
2-norm average error	$\|e\|_2$	$\sqrt{\dfrac{1}{Q} \sum_{1 \leq i \leq Q} e_i{}^2}$
infinity-norm error	$\|e\|_\infty$	$\max_{1 \leq i \leq Q} \{e_i\}$

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

$$\|e\|_p = \left(\frac{1}{Q} \sum_{1 \leq i \leq Q} e_i{}^p \right)^{1/p}.$$

The first step in thresholding is weighting the coefficients in a certain way (which corresponds to using a particular basis, such as an orthonormal basis, for example). In particular, for the Haar basis, normalization is done by dividing the wavelet coefficients $\widehat{S}(2^j), \ldots, \widehat{S}(2^{j+1} - 1)$ by $\sqrt{2^j}$, for $0 \le j \le \log N - 1$.

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 for the set of singleton queries:

$$\left\{ Sum(i_1 : i_1, \ldots, i_d : i_d) \ \middle| \ 0 \le i_j < |D_j|, \text{ for each } 1 \le j \le d \right\}.$$

That is, if we want to minimize the average absolute error in approximating all the individual cells in S, the best choice is to keep the C largest (in absolute value) wavelet coefficients [SDS96].

But our goal here is to approximate d'-dimensional range-sum queries, where usually $d' \ll d$. If a coefficient c_i is more likely to contribute to a query than another coefficient c_j, we would like to give c_i a higher weight, even its absolute value is smaller than that of c_j. From Lemma 16 in Section 6, we can observe the following fact: For a coefficient $c = (i_1, \ldots, i_d, v)$, the bigger the value $\sum_{j=1}^{k} [i_j = 0]$, the more likely c is going to contribute to a d'-dimensional range-sum query.[2] We therefore define the weight function w for coefficient c as

$$w(c) = \sum_{j=1}^{k} [i_j = 0].$$

In doing thresholding, we pick the C'' ($C < C'' < C'$) largest wavelet coefficients in absolute value, and among those we pick the C wavelet coefficients with the largest weight with respect to function w. (We break ties using the absolute value.) We rank the C coefficients by ordering them according to their weights in decreasing order to get our compact data cube. Let us denote by R the compact data cube computed for a d-dimensional array S. We can view R as a one-dimensional array of length C, with each entry being a wavelet coefficient value and its indices in the sparse representation of form

$$R[j] = (i_{j_1}, \ldots, i_{j_d}, v_j), \qquad 1 \le j \le C. \qquad (12)$$

The entries are ranked according to their importance in decreasing order.

6. Answering On-Line Queries

In this section, we show how to answer on-line aggregation queries using the compact data cube constructed in the previous section. Let's consider a range-sum query $Sum(l_1 : h_1, \ldots, l_{d'} : h_{d'})$.

An advantage of the partial-sum approach of [VWI98] is that we need to reconstruct only $2^{d'}$ values, not 2^d values, of the partial-sum data cube in order to answer this query, which requires processing $\min\{k, 2 \prod_{1 \le i \le d'} \log |D_i|\}$ wavelet coefficients in the worst case, where k is the specified number of stored coefficients in the compact data cube to use for the approximation. If we abandon the partial-sum data cube and use the compact data cube, one big concern is that we may lose this speed advantage. It turns out that we will not. In fact, our algorithm for answering queries is even faster, both theoretically and in practice: In the partial-sum scenario using the logarithm transform in [VWI98], a wavelet coefficient

---

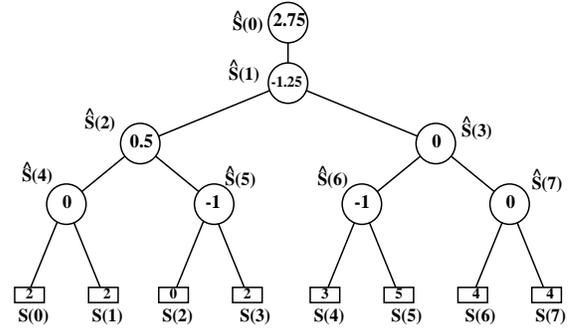2 We use the notation $[i_j = 0]$ to denote 1 if $i_j = 0$ and 0 if $i_j \neq 0$.

Figure 1: Error tree for $N = 8$.

may be involved in the approximation of several of the $2^{d'}$ values, so the CPU time complexity is $O(2^{d'} d' k)$, whereas in our new approach the CPU time complexity for the standard implementation is only $O(d'k)$ with a very small constant factor of proportionality (roughly 2) in terms of the number of arithmetic operations. (A more complicated approach yields an $O(d'k)$-time algorithm for the former case, but only if the logarithm transform is not used.)

To fully understand the on-line query processing algorithm, we examine the relationship between wavelet coefficients and a range-sum query by using the *error tree* structure introduced in [MVW98]. The error tree is built based upon the wavelet transform procedure. Figure 1 is the error tree for the example in Section 5.1. Each internal node is associated with a wavelet coefficient value, and each leaf is associated with an original signal value. (For purposes of exposition, the wavelet coefficients are unnormalized, but in the implementation the values are normalized and the algorithm is modified appropriately.) Internal nodes and leaves are labeled separately. Their labels are in the domain $\{0, 1, \ldots, N - 1\}$ for a signal of length N. For example, the root is an internal node with label 0 and its node value is 2.75 in Figure 1. For convenience, we shall use "node" and "node value" interchangeably.

The construction of the error tree exactly mirrors the wavelet transform procedure. It is a bottom-up process. First, leaves are assigned original signal values from left to right. Then wavelet coefficients are computed, level by level, and assigned to internal nodes.

As the figure shows, the (unnormalized) value of each internal node i is denoted by $\widehat{S}(i)$, and the value of each leaf j is denoted by $S(j)$. We use $left(i)$ and $right(i)$ to denote the left and right child of any node i, and we use $leaves(i)$ to denote the set of leaves in the subtree rooted at i. The average value of the nodes in $leaves(i)$ is denoted by $ave\_leaf\_val(i)$. For any leaf i, we use $path(i)$ to denote the set of internal nodes (or the node values) along the path from i to the root. For any two leaves $l \le h$, we use $S(l : h)$ to denote the range-sum between $S(l)$ and $S(h)$, that is,

$$S(l : h) = \sum_{i=l}^{h} S(i). \qquad (13)$$

Below are some useful facts that are helpful for understanding our algorithm.

Lemma 1 *For any nonroot internal node i, we have*

$$\widehat{S}(i) = \frac{ave\_leaf\_val\big(left(i)\big) - ave\_leaf\_val\big(right(i)\big)}{2}.$$

Lemma 2 *The reconstruction of any signal value depends only upon the values of those internal nodes along the path*

from the corresponding leaf to the root. That is, the reconstruction of any leaf value $S(i)$ depends only upon the nodes in $path(i)$.

Consider the range sum (13). It is an algebraic sum of many internal nodes. For example, for $l = 0$, $h = 1$,

$$S(0) + S(1) = \left(\widehat{S}(0) + \widehat{S}(1) + \widehat{S}(2) + \widehat{S}(4)\right)$$
$$+ \left(\widehat{S}(0) + \widehat{S}(1) + \widehat{S}(2) - \widehat{S}(4)\right).$$

Note that the two terms of $\widehat{S}(4)$ cancel out each other, so $\widehat{S}(4)$ does not contribute to the final summation. In general, any original signal $S(i)$ can be represented as the algebraic sum of the wavelet coefficients along the path $path(i)$. A nonroot internal node contributes positively to the leaves in its left subtree and negatively to the leaves in its right subtree. For a range sum, the contributors may cancel each other, and we have the following result:

Lemma 3 *A nonroot internal node x contributes to the range sum (13) only if $x \in path(l) \cup path(h)$. In particular, the contribution of x to (13) is*

$$\left(\left|left\_leaves(x, l:h)\right| - \left|right\_leaves(x, l:h)\right|\right) \times \widehat{S}(x),$$

where

$$left\_leaves(x, l:h) = leaves\left(left(x)\right) \cap [l, h]; \quad (14)$$
$$right\_leaves(x, l:h) = leaves\left(right(x)\right) \cap [l, h]. \quad (15)$$

Mathematically, we can write any range sum in terms of all the wavelet coefficients as

$$S(l:h) = \sum_{x} \left(\left|left\_leaves(x, l:h)\right| - \left|right\_leaves(x, l:h)\right|\right) \widehat{S}(x),$$

where the summation is over all internal nodes x. In our algorithm, however, we do not evaluate all the terms. We quickly determine the nonzero contributors and evaluate their contribution.

Let us relook at the example in Section 5.1; its error tree is shown in Figure 1. The original signal S can be reconstructed from $\widehat{S}$ by the following formulas:

$$
\begin{array}{rcllll}
S(0) &=& \widehat{S}(0) + \widehat{S}(1) + \widehat{S}(2) & + \widehat{S}(4) & & \\
S(1) &=& \widehat{S}(0) + \widehat{S}(1) + \widehat{S}(2) & - \widehat{S}(4) & & \\
S(2) &=& \widehat{S}(0) + \widehat{S}(1) - \widehat{S}(2) & & + \widehat{S}(5) & \\
S(3) &=& \widehat{S}(0) + \widehat{S}(1) - \widehat{S}(2) & & - \widehat{S}(5) & \\
S(4) &=& \widehat{S}(0) - \widehat{S}(1) & + \widehat{S}(3) & & + \widehat{S}(6) \\
S(5) &=& \widehat{S}(0) - \widehat{S}(1) & + \widehat{S}(3) & & - \widehat{S}(6) \\
S(6) &=& \widehat{S}(0) - \widehat{S}(1) & - \widehat{S}(3) & & + \widehat{S}(7) \\
S(7) &=& \widehat{S}(0) - \widehat{S}(1) & - \widehat{S}(3) & & - \widehat{S}(7)
\end{array}
$$

For example, $S(2)$ depends only upon $path(2) = \{\widehat{S}(5), \widehat{S}(2), \widehat{S}(1), \widehat{S}(0)\}$. If we want to compute the range sum $S(2:5)$, we can see that although $\widehat{S}(1)$ contributes to each of $S(2)$, $S(3)$, $S(4)$, and $S(5)$, its total contribution cancels out, and the net effect is that it does not contribute at all. Similarly, $\widehat{S}(5)$ and $\widehat{S}(6)$ are gone, and we have

$$S(2:5) = 4\widehat{S}(0) - 2\widehat{S}(2) + 2\widehat{S}(3).$$

The formula can also be verified by using Lemma 3.

We can extend the above observation to the multidimensional case. For example, for a two-dimensional array with $|D_1| = |D_2| = 8$, we can answer the range-sum query $Sum(4:7, 0:7)$ (note that D_2's range is the special range all_2)

using the following formula that involves only coefficients $\widehat{S}(0,0)$ and $\widehat{S}(1,0)$:

$$Sum(4:7, 0:7) = 8 \times 4 \times \left(\widehat{S}(0,0) - \widehat{S}(1,0)\right).$$

Lemma 4 *In the reconstruction process, a wavelet coefficient $c = (i_1, \ldots, i_d, v)$ contributes to the range sum $Sum(l_1:h_1, \ldots, l_{d'}:h_{d'})$ only if $i_{d'+1} = \cdots = i_d = 0$. Its contribution is*

$$v \prod_{j=1}^{d} \left(\left|left\_leaves(i_j, l_j:h_j)\right| - \left|right\_leaves(i_j, l_j:h_j)\right|\right)$$

$$= v \prod_{j=1}^{d'} \left(\left|left\_leaves(i_j, l_j:h_j)\right| - \left|right\_leaves(i_j, l_j:h_j)\right|\right)$$

$$\times \prod_{j=d'+1}^{d} |D_j|.$$

To answer a query of form $Sum(l_1:h_1, \ldots, l_{d'}:h_{d'})$ using k coefficients of the compact data cube R, we use the following algorithm:

$AnswerQuery(R, k, l_1, h_1, \ldots, l_{d'}, h_{d'})$
  $answer = 0$;
  for $i = 1, 2, \ldots, k$ do
    if $Contribute(R[i], l_1, h_1, \ldots, l_{d'}, h_{d'})$ then
      $answer = answer +$
        $Compute\_Contribution(R[i], l_1, h_1, \ldots, l_{d'}, h_{d'})$;
  for $j = d' + 1, \ldots, d$ do
    $answer = answer \times |D_j|$;
  return $answer$;

Function $Contribute(R[i], l_1, h_1, \ldots, l_{d'}, h_{d'})$ returns *true* if the $R[i]$ contributes to the range-sum query, and it returns *false* otherwise. The actual contribution of $R[i]$ to the specified query is computed by function $Compute\_Contribution(R[i], l_1, h_1, \ldots, l_{d'}, h_{d'})$.

Based upon the regular structure of the error tree and the preceding lemmas, we have devised two algorithms to compute the above two functions. Each algorithm has CPU time complexity of about $2d'$. For reasons of brevity, we defer the details to the full paper.

Theorem 3 *For a given aggregation query of form*

$$Sum(l_1:h_1, \ldots, l_{d'}:h_{d'}),$$

the approximate query answer can be computed based upon the top k coefficients in the compact data cube using a $\left((d+1)k\right)$-space data structure in $2d' \times \min\left\{k, 2 \prod_{1 \leq i \leq d'} \log |D_i|\right\}$ CPU time.

Proof Sketch: We only need to process the first k coefficients in R. Each of the coefficients is a $(d+1)$-tuple of the form (6), so the space complexity is $(d+1)k$. The CPU time complexity follows easily from that of the two functions. An alternate mechanism is to process only the coefficients needed, which are at most $2 \prod_{1 \leq i \leq d'} \log |D_i|$. □

Often the first k coefficients of the compact data cube reside in internal memory. If instead they are on disk, they occupy $\lceil dk/B \rceil$ disk blocks, which is typically one or a small constant, so they can be retrieved with a constant number of I/Os. In terms of CPU time, the quantity $2 \prod_{1 \leq i \leq d'} \log |D_i|$ is almost always larger than k in practice, so the faster and simpler way to evaluate the approximation takes $O(d'k)$ CPU time. By comparison, the CPU running time is $2^{d'}$ times faster than the algorithm in [VWI98]!

Our algorithm has the useful feature that it can progressively refine the approximate answer with no added overhead.

If a coefficient contributes to a query, its contribution can be computed independently of the other coefficients. Therefore, to refine a query answer, the contribution of a new coefficient can be added to the previous answer in $O(d')$ CPU time, without starting over from scratch.

7. Experiments

7.1. Data Description

In many OLAP applications, the data have high dimensions 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 both real-world data and synthetic data of high dimension. For brevity, we report the accuracy of our approximate query answers for only real data. To analyze the speed of our compact data cube construction algorithm, we report the running time of our algorithm on tunable synthetic datasets.

We obtained our real-world data from the U.S. Census Bureau using their Data Extraction System (DES) [Bur]. Our data source is the Current Population Survey (CPS) and our extracted file is the March Questionnaire Supplement–Person Data File. The 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 ten-dimensional array with more than 16 million cells. The density of the array is about 0.001; there are 15,985 nonzero elements. In this setting we can imagine that several sparse data sets are approximated, and each data set must be approximated using very little space.

Our synthetic datasets are generated using our own data generation model described in the Appendix.

7.2. Efficiency of the Compact Data Cube Construction Algorithm

We implemented our compact data cube construction algorithms using the *Transparent Parallel I/O Programming Environment* (TPIE) system [VV96, Ven97, Ven94]. TPIE is a collection of templated functions and classes to support high-level and efficient implementations of external memory algorithms. The basic data structure in TPIE is a *stream*, representing a list of objects of an arbitrary type. The system includes I/O efficient implementations of algorithms for scanning, merging, distributing, and sorting streams, which are building block for our algorithms.

We did our experiments on a Digital Alpha workstation running Digital UNIX 4.0, with 512MB of internal memory. Since the sizes of the raw data sets used in our experiments are relatively small (44MB to 1GB), we restricted the amount of internal memory used by our program to be in the range from 1MB to 10MB. For all the runs using Algorithm I, the logical block transfer size used by TPIE streams was 256KB (32 times the physical disk block size 8KB) in order to achieve a high transfer rate. Smaller logical block sizes resulted in slightly longer run times. However, for all the runs using Algorithm II, we used a smaller logical block transfer size of 16KB in order to keep the number g of passes small.
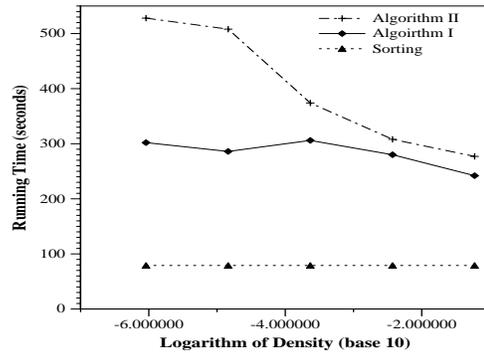

Figure 2: Construction time vs. density.

In the first group of experiments, we use synthetic data and measure the elapsed time of the compact data cube construction algorithms as a function of the data density N_z/N. We fix the number of dimensions d, the number of nonzero entries N_z, and the internal memory size M.

Figure 2 depicts the results from one set of the experiments, in which we fix $d = 10$ and $N_z = 10^6$. The size of each data item is 44 bytes, and the internal memory size parameter M is set to 190650 (corresponding to 8MB). The size of the sparse representation of the raw data is 44MB. By changing the dimension size parameters $|D_i|$, we obtain multidimensional arrays with different sizes N in the range from $16 \times N_z$ to $2^{20} \times N_z$. This corresponds to the densities in the range from 0.06 to 10^{-6}. We partition the dimensions according to (7). For all data sets, we have $g = 2$, although the ones with small density have very big values of N. For example, the data set with density of 10^{-6} has array size $N \approx 2^{40}$.

We ran our compact data cube wavelet decomposition algorithms against five data sets. The results are shown in Figure 2. To make the plot clear, we plot the logarithm of density on the x axis. The x coordinate $x = -i$ corresponds to a density of 10^{-i}. As we can see from the figure, the running time of Algorithm I for the five data sets varies slightly (in the range from 242 seconds to 306 seconds) as the density changes. The differences in running time are mainly caused by the effect of the on-line cutoff. For some runs, slightly more than N_z coefficients are written out during the first pass, which causes a longer running time, whereas it is the other way around for some other runs. The running time of Algorithm II decreases significantly as the density increases. The reason for the decrease is that N_z is fixed, and a data set with small density corresponds to a big N value. Since Algorithm II cannot apply the more liberal group partitioning
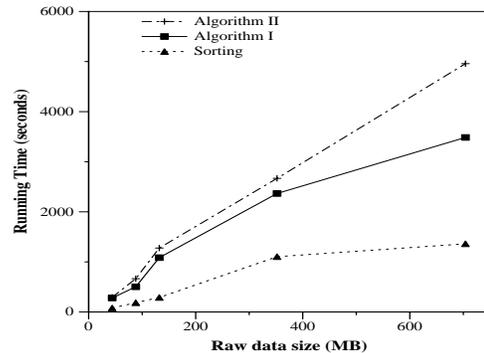

Figure 3: Construction time vs. input raw data size.

of (7), the resulting g value is large. On the other hand, Algorithm I takes advantage of (7) for partitioning the groups and performs noticeably better than Algorithm II for very sparse data.

Our methods require less time and storage space than do other methods. For example, for the data set with the smallest density $1/10^6$, if we use the partial-sum data cube approach of [HAMS97], we will need to process and store a partial-sum cube that contains 10^{12} nonzero cells and takes up 4GB storage space if we use the MOLAP (array) representation, even though the raw data size is only 44MB. If we use the compact partial-sum data cube approach in [VWI98], the final compact partial-sum data cube is much smaller in size than that of the raw data, but there will be time and space problems during the construction stage because of the need to compute the dense partial-sum data cube during the wavelet decomposition.

In the second group of experiments, we measure the elapsed time in terms of the raw data size. In each set of the experiments in this group, we fix the number of dimensions d, the density N_z/N, and the amount of internal memory M. By changing N_z and N proportionally, we obtain data sets with the same density but different size.

Figure 3 plots the result of one set of the experiments, in which we use $d = 10$, data item size of 44 bytes, $M = 190650$ (corresponding to 8MB), and a density of 0.001. The value N_z varies from 1 million to 16 million, corresponding to a raw data size from 44MB to 704MB. From Figure 3, we can see that the running time of both algorithms scales almost linearly with respect to the input data size.

In all the experiments, the initial order of the data is in the order needed for the first pass of the data cube construction algorithms. We also graph in Figures 2 and 3 the time it takes to sort each data set using TPIE, so that the speed of the data cube construction algorithms can be assessed in terms of the time it takes to sort a similarly sized file. Note however that the TPIE sorting routine has an extra speed advantage in that it has been carefully optimized. Algorithm I was fairly easy to program since it makes use of the TPIE scan and sorting operations. However, it can be optimized further by performing the first pass of each transposition step during the actual wavelet decomposition, as suggested in Section 5.2.1. Algorithm II should also be further optimized; our implementation did not make use of double buffering.

7.3. Accuracy of the Approximate Answers

In this section, we compare the accuracy of our method with that of the traditional random sampling method in answering typical range-sum queries. The simplest way of using random sampling is, during the off-line phase, to take a random sample of a certain size from the raw data. When a query is presented in the on-line phase, the query is evaluated against the sample, and an approximate answer is given in the obvious way: If the answer of the query using a sample of size t is s, the approximate answer is $s \times N_z/t$. The new sampling-based summary statistics proposed in [GM98] cannot be applied here to any advantage since our raw data do not contain duplicate tuples. We chose not to do any comparisons with traditional histogram methods [PIHS96, PI97], because as we mentioned in Section 1, they are too inefficient to construct for high-dimensional data that cannot fit in internal memory.

The relative effectiveness of random sampling and that of our method are fairly constant over a wide variety of synthetic data sets and range-sum query sets. Our compact data cube

generally provides more accurate results than that of a random sample of the same size. If the locations of the nonzero entries are uniformly distributed in the multidimensional array, random sampling may perform better. But uniform distributions in real-life data warehouses are rare.

We measure the accuracy of the methods by using both the real data and synthetic data. For the brevity of this paper, we present the results from a typical set of our real-data experiments. (The other experiments were qualitatively similar.) In the experiments, we specify partial ranges on $d' = 3$ of the $d = 10$ dimensions and average our results over the following d'-dimensional range-sum queries:

$$\left\{ Sum(l_1\!:\!h_1, l_2\!:\!h_2, l_3\!:\!h_3) \;\middle|\; h_i = l_i + \Delta,\; 0 \le l_i \le h_i < |D_i|,\right.$$
$$\left. \text{for each } 1 \le i \le 3 \right\}$$

where Δ is a nonnegative integer constant.

Figure 4 plots the accuracy of our method in comparison with random sampling for different error metrics and various storage space sizes k. We used $\Delta = 10$. The absolute errors are normalized by the largest exact answer $L = 766327$ for the query set. The sampling results are the averages for five different runs. The storage size is measured in terms of the number k of wavelet coefficients (for our method) or the number of sample points (for sampling) used in answering the queries. The wavelet coefficients and the sample points are each of form (6), which is a $(d + 1)$-tuple; since $d = 10$ for our data set, each wavelet coefficient (sample point) is represented by a tuple of 11 numbers.

As shown in Figure 4, the accuracy of our compact data cube is noticeably better than that of random sampling. For example, when using only $k = 50$ coefficients in our compact data cube method, the average relative error for the query set is only 17%, and the average relative error is about 10 times better than for random sampling.

8. Conclusions

In this paper, we present an efficient and effective technique based upon a multiresolution wavelet decomposition that yields an approximate and space-efficient representation of the underlying multidimensional data in OLAP applications.

Our compact data cube construction algorithms are very efficient in term of I/O complexity, especially when the raw data are very sparse. In the on-line phase, by using the hierarchical structure of the wavelet decomposition, we obtain an efficient algorithm for answering typical OLAP aggregation queries. Experiments show our compact data cube provides excellent approximation for on-line range-sum queries with limited space usage and little computational cost

One feature of the approach of Vitter et al. [VWI98] for dense multidimensional arrays that we have not incorporated into the sparse approach presented in this paper is the use of special transforms for further improvements in relative accuracy. The logarithm transform in [VWI98] is based upon the use of the partial-sum data cube. It would be interesting to explore alternative transforms or perhaps compression approaches to the partial-sum data cube. It is possible that such transforms are effective mainly in lower dimensions.

We are considering alternative 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. In other joint work, we are developing dynamic efficient algorithms for maintaining the compact data
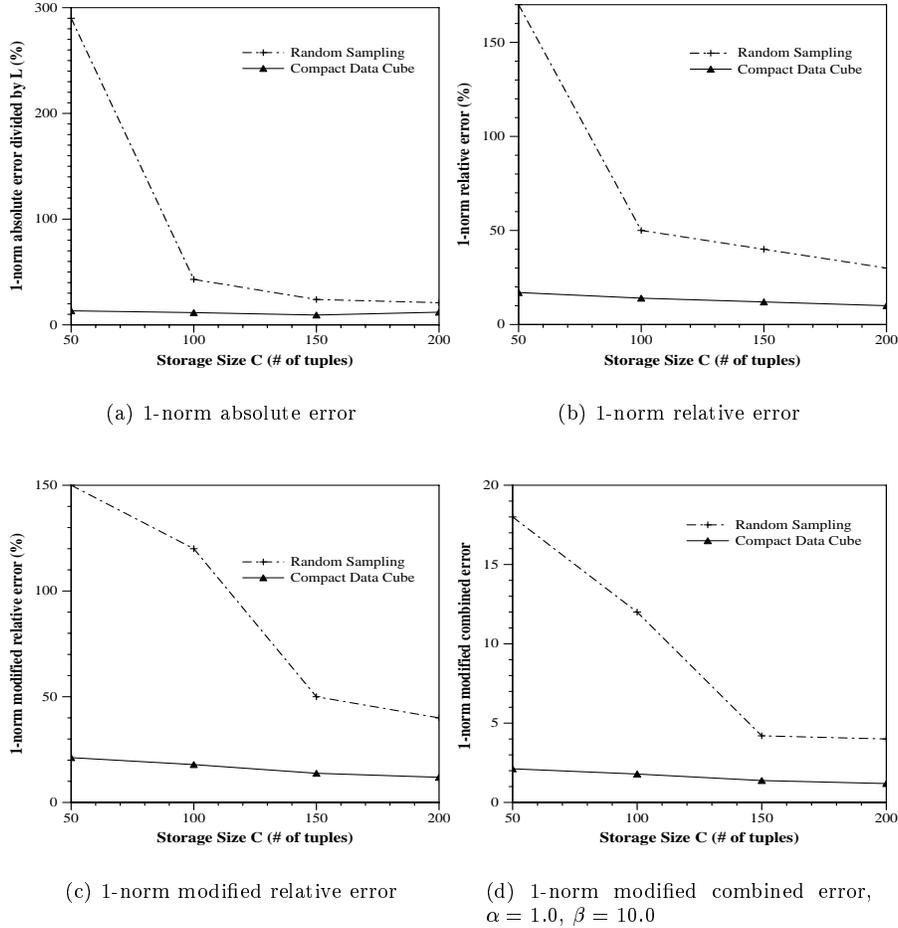
(a) 1-norm absolute error

(b) 1-norm relative error

(c) 1-norm modified relative error

(d) 1-norm modified combined error, $\alpha = 1.0$, $\beta = 10.0$

Figure 4: Accuracy of approximate query answers for the compact data cube and for random sampling.

Attribute	Value
d	the number of dimensions
$|D_i|$	the size of the ith dimension, $1 \leq i \leq d$
n_region	the number of dense regions in multidimensional space
$n1\_region$	the number of type 1 dense regions
$n2\_region$	the number of type 2 dense regions
T	the sum of all the nonzero values
Z	the Zipf parameter for the value distribution dense regions
z_min	the minimum Zipf parameter for type 2 dense regions
z_max	the maximum Zipf parameter for type 2 dense regions
V_min	the minimum volume of a dense region
V_max	the maximum volume of a dense region
$noise\_volume\_level$	% of the number of nonzero entries outside dense regions w.r.t to the total number of non-zero entries
$noise\_weight\_level$	% of the sum of the nonzero values outside dense regions w.r.t. T

Table 1: Description of the synthetic data.

cube, given updates in the underlying raw data. We are also working on applying our techniques to other operators (e.g., *projection*) other than *Sum*.

References

[AAD+96] S. Agarwal, R. Agrawal, P. Deshpande, J. Naughton, S. Sarawagi, and R. Ramakrishnan. On the computation of multidimensional aggregates. In *Proceedings of the 1996 International Conference on Very Large Databases*, Mumbai, India, 1996.

[AV88] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[BDF+97] D. Barbara, W. DuMouchel, C. Faloutsos, P. J. Haas, J. M. Hellerstein, Y. Ioannidis, H. V. Jagadish, T. Johnson, R. Ng, , V. Poosala, K. A. Ross, and K. C. Sevcik. The New Jersey data reduction report. *Bulletin of the Technical Committee on Data Engineering*,

20(4), 1997.

[Bur] U.S. Census Bureau. Census bureau databases. The online data are available on the web at http://www.census.gov/.

[FJS97] C. Faloutsos, H. V. Jagadish, and N. D. Sidiropoulos. Recovering information from summary data. In *Proceedings of the 1997 International Conference on Very Large Databases*, Athens, Greece, August 1997.

[GBLP96] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. 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)*, pages 131–139, 1996.

[GM98] 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, WA, June 1998.

[HAMS97] C.-T. Ho, R. Agrawal, N. Megiddo, and R. Srikant. Range queries in OLAP data cubes. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, Tucson, AZ, May 1997.

[HHW97] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, Tucson, AZ, May 1997.

[HRU96] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, Montreal, Canada, May 1996.

[JS94] B. Jawerth and W. Sweldens. An overview of wavelet based multiresolution analyses. *SIAM Rev.*, 36(3):377–412, 1994.

[MVW98] 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*, pages 448–459, Seattle, WA, June 1998.

[PC98] N. Pendse and R. Creeth. The OLAP report, 1998. The online report is available on the web at http://www.olapreport.com/Analyses.htm/.

[PI96] V. Poosala and Y. E. Ioannidis. Estimation of query-result distribution and its application in parallel-join load balancing. In *Proceedings of the 1996 International Conference on Very Large Databases*, Bombay, India, September 1996.

[PI97] 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.

[PIHS96] V. Poosala, Y. E. Ioannidis, P. J. Haas, and E. Shekita. Improved histograms for selectivity estimation of range predicates. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, Montreal, Canada, May 1996.

[SAC+79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, pages 23–34, 1979.

[SDS96] E. J. Stollnitz, T. D. Derose, and D. H. Salesin. *Wavelets for Computer Graphics*. Morgan Kaufmann, 1996.

[Ven94] D. E. Vengroff. A transparent parallel I/O environment. In *Proceedings of the 1994 DAGS Symposium on Parallel Computa tion*, July 1994.

[Ven97] D. E. Vengroff. *TPIE User Manual and Reference*. Duke University, 1997. The manual and software distribution are available on the web at http://www.cs.duke.edu/TPIE/.

[Vit99] J. S. Vitter. External memory algorithms and data structures. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*, DIMACS series. American Mathematical Society, to appear 1999. Available via the author's web page http://www.cs.duke.edu/~jsv/.

[VS94] 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.

[VV96] 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*, NASA Conference Publication 3340, Volume II, pages 553–570, College Park, MD, September 1996.

[VWI98] J. S. Vitter, M. Wang, and B. Iyer. Data cube approximation and histograms via wavelets. In *Proceedings of Seventh International Conference on Information and Knowledge Management*, pages 96–104, Washington D.C., November 1998.

[ZDN97] 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, AZ, May 1997.

Appendix: Synthetic Data Description

We use own data generation model and different combinations of parameters to generate a wide variety of synthetic datasets for our timing experiments. We model the distribution of the nonzero values in a multidimensional array by the parameters defined in Table 1.

The program will generate the sparse representation of a d-dimensional array whose size is determined by the parameters $|D_i|$s, for $1 \leq i \leq d$. The nonzero entries are mainly located in n_region dense regions. The center of each dense region is a randomly picked position in the d-dimensional array. The *volume* of any dense region (which is defined as the number of cells in the region) is a random number between V_min and V_max. Each region has a sum which is the summation of the values for all the cells contained in the region. The Zipf distribution parameter Z, together with $T(1 - noise\_weight\_level)$ and n_region, are used to generate values which are randomly assigned to each region as its sum.

A dense region can be either type 1 or type 2. A type 1 region has a distribution where all dimensions are independent of one other and obey the unbiased binomial distribution. To generate a type 2 region, we first use the Zipf distribution with parameter z (where z is uniformly chosen from $[z\_min, z\_max]$) to generate a set of values. The values are then assigned to the cells in the region in such a way that the closer a cell is to the center, the bigger the assigned value is.

We also consider the fact that besides the nonzero cells in the dense regions, there might be some isolated nonzero cells outside those dense regions. The number of such cells is defined by the parameters $noise\_volume\_level$ and the sum of these isolated nonzero values are defined by $noise\_weight\_level$. Their positions are generated in a random way.