Quasi-Cubes: A space-efficient way to support approximate multidimensional databases

Daniel Barbará *
George Mason University
ISSE Dept.
Fairfax, VA 22182
dbarbara@isse.gmu.edu

Mark Sullivan
Juno Online Services
120 West 45th Street, 39th floor
New York, NY 10036
sullivan@staff.juno.com

February 20, 1998

Abstract

A data cube is a popular organization for summary data. A cube is simply a multidimensional structure that contains at each point an aggregate value, i.e., the result of applying an aggregate function to an underlying relation. In practical situations, cubes can require a large amount of storage. The typical approach to reducing storage cost is to materialize parts of the cube on demand. Unfortunately, this lazy evaluation can be a time-consuming operation.

In this paper, we propose an approximation technique that reduces the storage cost of the cube without incurring the run time cost of lazy evaluation. The idea is to characterize regions of the cube by using statistical models whose description take less space than the data itself. Then, the model parameters can be used to estimate the cube cells with a certain level of accuracy. To increase the accuracy, some of the "outliers," i.e., cells that incur in the largest errors when estimated can be retained. The storage taken by the model parameters and the retained cells, of course, should take a fraction of the space of the full cube and the estimation procedure should be faster than computing the data from the underlying relations. We study three methods for modeling cube regions: linear regression, singular value decomposition and the independence approach. We study the tradeoff between the amount of storage used for the description and the accuracy of the estimation. Experiments show that the errors introduced by the estimation are small even when the description is substantially smaller than the full cube. Since cubes are used to support data analysis and analysts are rarely interested in the precise values of the aggregates (but rather in trends), providing approximate answers is, in most cases, a satisfactory compromise. Our technique allows systems to handle very large cubes that cannot either be fully stored or efficiently materialized on demand. Even for applications that can only accept tiny error in the cube data, our techniques can be used to present successive on-line refinements of the answer to a query so that a coarse picture of the answer can be presented while the remainder of the answer is fetched from disk (i.e., to support on-line aggregation).

1 Introduction

A *data cube* is a popular organization for summary data [11]. A cube is simply a multidimensional structure that contains at each point an aggregate value, i.e., the result of applying an aggregate function to an underlying relation. For instance, a cube can summarize sales data for a corporation, with dimensions "time of sale," "location of sale" and "product type".

Cubes are organized in a hierarchical fashion. At the base of the hierarchy are the aggregates computed from the underlying relation(s) (we call these *base data*, following the notation

*phone: 703-9931627, fax:703-9931638

of [9]). The aggregates in the lower levels of the hierarchy are used to construct the coarser-grain aggregates of the higher levels. At any level in the hierarchy, data can be thought as comprising a multidimensional matrix.

In our corporate sales example, the base data is the total retail sales by *days*, *stores* and *product*, where *product* is a list of products sold by the corporation. Higher levels of the hierarchy can be specified in terms of *weeks*, *cities* and *product type* (where *product type* can be higher classification of products such as "dolls", "VCRs," etc.), or *months*, *countries*, *product dept.* The base data (or any other level of the hierarchy in this cube) corresponds to a three-dimensional matrix.

The underlying relation is commonly referred to as *fact table* in the *star schema* [16]. The fact table contains all the attributes that determine the dimension of the cube, plus an attribute on which the aggregations are to be performed. Along with this table, there are *dimension tables* whose keys are foreign keys of the fact table and which describe each one of the dimension attributes.

A data cube can be implemented using an *eager* or a *lazy* materialization strategy (Widom uses these terms to refer to warehouse materialization, although her definition of *lazy* is slightly different from ours [26].). In an *eager* strategy, the entire cube is materialized at initialization time. The advantage of this strategy is that the materialized cube can be queried quickly. The primary disadvantage is that the cost of storing the materialized cube can be large. For instance, in our previous example, if we assume 10,000 stores, 365 days and 1,000 products, materializing just the base data requires storage for 3.65 billion aggregate values. The *lazy* strategy defers computation of the cube entries until users examine them. When the entries are needed, the system queries the underlying database to compute them. Sometimes, a hybrid strategy is used in which part of the cube is materialized (e.g., the base data) and the rest is computed on demand.

In this paper, we propose an approximation technique that reduces the storage cost of the cube without incurring the run time cost of lazy evaluation. The idea is to provide an incomplete description of the cube and a method of estimating the missing entries with a certain level of accuracy. The description, of course, should take a fraction of the space of the full cube and the estimation procedure should be faster than computing the data from the underlying relations. We study the tradeoff between the amount of storage used for the description and the accuracy of the estimation. We call these approximated cubes *Quasi-Cubes*.

The Quasi-Cube data structure has two components: model parameters and retained points. We use a model, such as linear regression, to describe parts of the data cube (e.g., columns, planes). The model allows us to generate an entire cube from a small set of parameter values. However, the real cube entries will not fit the values predicted by the model exactly, so additional storage is used to improve accuracy. During cube construction, we calculate, for each point in the cube, the difference between the value predicted by the model and the cell's true value. Then, we rank the cube points by this estimation error and select the highest ranking entries as "retained points" that will be stored rather than computed. Specifying the most poorly predicted points improves the model's accuracy. The more points one is willing to retain, the lower the estimation errors will get.

Since cubes are used to support data analysis and analysts are rarely interested in the precise values of the aggregates (but rather in trends), providing approximate answers is, in most cases, a satisfactory compromise. Our technique can allow systems to handle very large cubes that cannot either be fully stored or efficiently materialized on demand.

We conducted a set of experiments to evaluate Quasi-Cubes. The experiments were conducted over synthetically generated data and real data sets. Several metrics were used to evaluate the techniques: errors incurred in the estimation, space savings achieved by Quasi-Cubes, time to build a Quasi-Cube compared to the time required to build a conventional cube, and performance of queries over a Quasi-Cube. Our results show that for a small fraction of retained points (10 %) the maximum estimation errors range from 1 to 20 % depending of the distribution of the data in the set. (For the real dataset, the results were specially encouraging.) For some datasets (including the real dataset), one can achieve savings of 80 % or more of the original space needed for a conventional cube. For some queries, drastic reductions of 50 % or more of the running time can be achieved by tolerating maximum errors of 10 %. While building models for the data imposes overhead over building conventional cubes, the overhead is tolerable and moreover, it only impacts the CPU time.

For several reasons, Quasi-Cubes can be especially useful in data warehouses:

- Data warehouses can be very large. The underlying relation(s) might be archived in tertiary storage after summarizing the data. That makes them, for all practical purposes, not available on-line, forcing the eager strategy for supporting cubes. The eager cube for such a large warehouse will either have very coarse-grained aggregates or be very large itself.

- The process of generating entries in the cube may be complex. For instance, when the data comes from a variety of sources and combining it requires some data manipulation. (Some stores keep track of sales by the hour and another by days). The process of combination is called "data scrubbing" [26] and can be a costly operation. Lazy evaluation can be prohibitively expensive in these environments, but Quasi-Cubes remain a reasonable low-storage-cost alternative to eager cubes.

- The data warehouse is usually dynamic. The underlying relation(s) are extended as data is added to the warehouse, so the cube must be either recalculated (at coarser granularity) or storage must be added for the expanding cube. Query evaluation strategies must be incremental or the cube generation (lazy or eager) becomes more and more expensive over time. On the other hand, Quasi-Cubes expand incrementally fairly easily by making incremental changes to the model parameters and replacing the model's "retained points" as higher-ranked ones arrive. Of course, representing more data over time with the same amount of storage lessens the accuracy of the results.

The paper is organized as follows. In Section 2 we describe the basics of Quasi-Cubes. In Section 3, we present some experimental results that demonstrate that the method is practical. Section 4 discusses some additional benefits that come from modeling data regions. Section 5 summarizes the related work and finally Section 6 offers some conclusions and future avenues of research in the area.

2 Quasi-Cubes

In this section, we explain the basics of constructing and querying Quasi-Cubes. The idea behind our method is to divide the cube in regions and use a model to describe each region. The model provides a more concise description of the region, occupying less space than the

100	70	20	190
100	90	40	230
100	110	80	290
100	130	60	290
400	400	200	

Figure 1: Sample Aggregates Matrix

100	70	20
100	90	40
100	110	80
100	130	160

Figure 2: The estimated cube, using the models of Equation 1

group of cells being modeled. The model can be used to estimate cells in the region. Doing so, one introduces errors in the estimation (with regards to the original values). To keep the errors low, the cell values with the highest errors are stored rather than modeled. The accuracy of the method improves as more cells are retained, establishing a tradeoff between the space devoted to the specification of the Quasi-Cube and the errors obtained.

To illustrate the point, consider the simple two-dimensional cube of aggregates shown in Figure 1. (Throughout the section, we will use the matrix of Figure 1 as a running example.) In general, we call an original base data cube, like that one, the *aggregates cube*. The entries in the example matrix are aggregates of retail sales computed from an underlying relation. The horizontal dimension is retail stores and the vertical dimension quarters of the year. (We keep this cube small for illustration purposes.) The sum of the entries for this cube is 1,000.Figure 1 also has one more column at the right and one more row at the bottom than Figure 1. Each entry in the extra column contains the sum of the other matrix entries in its row. The entries in the last row contain the sums of the values in their respective columns.

To illustrate the notion of modeling, let us partition the cube in 1 into three regions, each corresponding to a column. Now we can describe each one of these regions with a model that approximates the original values. For instance, the columns in this example can be described by the three models shown in Equation 1. (These models are not very realistic, they are just shown for illustrating purposes.)

$$\begin{aligned} c_{i1} &= 100 \\ c_{i2} &= 10 \times (5 + 2i) \\ c_{i3} &= 10 \times 2^i \end{aligned} \tag{1}$$

In each of these models, c_{ij} is the estimated value of the cell with coordinates i, j. Notice that the space needed to describe the models is significantly less than that needed to store the entire cube. (In practice, however, we would like to describe all the regions of the cube with the same *type* of model, to avoid having to store a description of which model is used in each case.) Figure 2 shows the values obtained when using the models to reconstruct the cube. For the models chosen, both the first and second columns can be rebuilt without errors (the models fit the data perfectly). For the third column, however, one incurs in errors when estimating c_{43}. The estimated value is 160, while the real value of the cell is 60, for an absolute error of 100. The tradeoff that we face here is the following: if we are willing to store the real value of c_{43}, plus the description of the models, we can lower the error to 0. If, on the other hand, only the model descriptions are stored, the error is 100.

Sometimes it is useful to convert the cells in the cube into probability entries by dividing the cell values by total sum of the entries. The matrix of Figure 3 is the probability matrix corresponding to Figure 1. Of course, for the probability matrix in Figure 3, the sum of

0.1	0.07	0.02	0.19
0.1	0.09	0.04	0.23
0.1	0.11	0.08	0.29
0.1	0.13	0.06	0.29
0.4	0.4	0.2	

Figure 3: The probability matrix, derived from the matrix of Figure 1

0.076	0.076	0.038	0.19
0.092	0.092	0.046	0.23
0.116	0.116	0.058	0.29
0.116	0.116	0.05	0.29
0.4	0.4	0.2	

Figure 4: Completing the matrix from the marginal distributions

entries is one. The row and column sums are shown in the extra column and row, respectively. (These values are commonly known as *marginal distributions* [27].) We shall see shortly how the marginal distributions are used in our method.

The first decision to be made is that of which model to use and how to divide the cube in regions. In the next two subsections, we show two different models that we have experimented with: the independence model and linear regression. We have to point out that in practice the independence model performs very poorly and is kept here just for illustration purposes. The linear regression approach, however, has given us very promising results, as we shall see later.

2.1 Independence Assumption

The idea behind this method is to divide the cube in two-dimensional planes and follow the simple *mutual independence approach* [22] which assumes total independence between the two dimensions. (The cube can be divided in other higher-dimensional cubettes and the independence approach can be straightforwardly applied to them.) In this technique, we estimate each cube entry using the product of its corresponding row and column marginal distributions. The matrix in Figure 4 is constructed in this way from the marginal distributions of the matrix in Figure 3. As we can see, some of the entries are estimated correctly (particularly the last row of the matrix), but most are not. The average error for this estimation is 0.0123.

We can achieve better results by storing more information. For instance, in addition to the marginal distributions, we could retain **one cell** from the original matrix (see Figure 5). (We choose, of course, the cell for which the largest estimation error is incurred.) Because the cell is specified instead of calculated, it does not contribute to matrix error. Applying the mutual independence approach to the new matrix yields the results shown in Figure 6. (The exact procedure will be made clear shortly.) This new estimated matrix has an average error of 0.0094.

Thus, by retaining the "outliers" that are farthest from fitting the independence assumption, we reduce the error. Specifying more data points makes the estimation process more

0.1	?	?	0.19
?	?	?	0.23
?	?	?	0.29
?	?	?	0.29
0.4	0.4	0.2	

Figure 5: Marginal distributions and one specified entry

0.1	0.076	0.038
0.0851	0.092	0.046
0.1074	0.116	0.058
0.1074	0.116	0.05

Figure 6: Completing the unspecified entries

$totalrow = 0.0$
$totalcol = c_j$
for $i = 1, ..., m$
  If i, j is not in the list of specified entries
    $totalrow += r_i$
  else
    $totalcol -= P_{ij}$
for $i = 1, ..., m$
  If i, j is not in the list of x, y
    $p_{ij} = totalcol * \frac{r_i}{totalrow}$

Figure 7: Algorithm IND to estimate unspecified entries in column j

accurate (although, in this example, there is no sense in retaining more than eleven values - seven for the marginal distribution, four matrix entries - since the matrix itself has only twelve entries). This is the essence of our method: trade the number of specified matrix entries for accuracy. The hope is to find a point at which substantial portions of the matrix can be left unspecified while maintaining acceptable levels of accuracy.

We now present the algorithm used to estimate entries using the mutual independence approach. Algorithm IND in Figure 7 shows the pseudo-code for this algorithm. The algorithm solves a column j of the matrix. The notation p_{ij} refers to the data point calculated for row i, column j. The notation r_i refers to the marginal distribution of row i. The notation c_j refers to the marginal distribution of column j. Specified entries are denoted by P_{ij}. The dimensions of the matrix are m (rows) and n (columns). The symbols r_i and c_j denote row and column marginals respectively.

Several facts about IND are worth noticing. First, the algorithm acts by computing a column at a time. In the case in which no entries in the column have been specified, the algorithm simply converges to the multiplication of marginal distributions. For the general case, the variable *totalcol* is computed as sum the products of the marginal distributions for the unspecified entries in the column. Then, the estimated values are computed by distributing the remainder of the column sum (what is left after taking the specified values out) according to this products. Notice that, in order to compute any particular entry it necessary to compute its column.

Algorithm IND uses the simplest model to calculate missing entries: the assumption of independence between the column and row variables. In that sense, it is very economical, since it does not require the specification of any parameters (other than the row and column marginals). In practice, however, using Algorithm IND leads to large errors: the variables are not necessarily independent and the estimates may differ considerably from the real values. Consider the example of Figure 8 (we use only one column to illustrate the point, the last column displays the row marginals). If we were to use Algorithm IND to estimate the two entries in the given column of matrix 8, assuming that no entries are specified, the results would be as shown in Figure 9. We can see that the errors incurred in this case are large.

Additionally, when we use the independence approach, selecting the points to retain can

0.24	0.6
0.36	0.4
0.6	

Figure 8: Example of a column that illustrates non-independence

0.36	0.6
0.24	0.4
0.6	

Figure 9: Guessing using the independence assumption

be a difficult task. The problem is that each time an entry is specified, the row and column marginals change. This changes the estimation errors in the rest of the entries. Selecting the k points that minimize the errors in the estimation algorithm IND implies checking every subset of size k in the matrix. Since this is impractical for moderate to large matrices, one has to resort to heuristics. The best heuristic we found was to select in every column those points that, when selected alone, make the errors incurred by IND decrease the most. We show in the experiments section a set of graphs based on this heuristic.

Finally, an additional drawback of the independence approach is that in order to estimate any entry in the matrix, if the entry is not among those specified by the Quasi-Cube description, the entire column needs to be estimated.

2.2 Using Regression

Using independence gives a simple, concise way of estimating entries, but can lead to intolerable errors. To solve this, we introduce the use of linear regression as a model to describe the entries for a particular column. In other words, (for a two-dimensional plane) we will use a equation of the form:

$$p_{ij} = b_0 r_i + b_1 \tag{2}$$

to model column j of the matrix. In this case, r_i is the marginal distribution for column i. The known values of p_{ij} and r_i are used to produce the best estimate of b_0 and b_1 by linear regression techniques [27]. Notice that it is really not necessary to use probabilities in the case of regression: we can also use the cell value directly and make r_i the marginal sum for column i. The value of the cell can be simply recovered by multiplying p_{ij} by the sum of all the aggregates in the cube.

In the case of the column shown in the matrix of Figure 6, the regression would result in an equation of the form:

$$p_{ij} = -6.0 r_i + 6.0 \tag{3}$$

If we apply Equation 3, using the row distribution marginals of that matrix, we obtain the correct entries 0.36 and 0.24 for the column. Unfortunately, in practice we do not always get the correct answers, but using regression results in better estimates for the unknown entries in a column.

For every column in the matrix we will keep three parameters: the values of $b_0(j)$ and $b_1(j)$ and the value of the so-called "standard deviation for the unexplained error", easily calculated by the regression procedure. We denote this quantity by $stdev(j)$. The value of $stdev(j)$ is used to further correct the estimates by adding or subtracting it from the value we get by applying Equation 2. In order to know whether to add or to subtract this value, we keep a

for $i = 1, ..., m$
If i, j is not in the list of x, y
$p_{ij} = b_0(j) r_i + b_1 + sign(i,j) stdev(j)$

Figure 10: Algorithm *REG* to estimate unspecified entries in column j, using regression parameters

bit vector of signs. We call this vector $sign$. The corresponding sign of row i and column j is denoted by $sign(i, j)$.

As in the independence case, we will store specified entries in the Quasi-Cube to improve the accuracy of the estimates. We select as specified points those that will minimize the errors incurred by the estimation algorithm. Unlike the independence case, the rank that we use to determine whether a matrix entry would make a good specified point is calculated independently for each matrix entry. Choosing to retain one point does not affect whether or not another should be chosen.

Summarizing, the Quasi-Cube is described by the following data:

- The marginal distributions per row (r_i).
- Three values per column: $b_0(j)$, $b_1(j)$ and $stdev(j)$.
- A bit vector: $sign$, which will be used to correct the estimates.
- A set of k specified entries.
- The total value of the aggregates $total$ (to reconstruct the aggregates matrix).

Algorithm *REG* in Figure 10 is the procedure we use in the warehouse to estimate any of the entries in the matrix. Although the algorithm shows how to estimate an entire column of entries, individual entries can be estimated separately.

In practice, cubes are often sparse, i.e., many of the cells have a zero value. It would be unwise to subject these cells to an estimation process, since it is unlikely that the estimated values would be exactly zero. Moreover, many current OLAP products take advantage of the sparseness to effectively reduce the size of the cube by using specialized data structures that retain only the non-zero cells, Incorporating empty cells in the Quasi-Cube would eliminate the storage savings achieved by our technique. Thus, we need a method to mark empty cells and to involve only non-zero cells in the modeling process. This is easily achieved by keeping a bitmap description of the cube that contains 0's for the zeros and 1's for the positive cells. This bitmap can be heavily compressed, and therefore will not take much space. To avoid paying the overhead of decompression during retrieval one can rely on methods of doing "AND" and "OR" operations on compressed bitmaps [10].

In any given column, the linear regression would be performed over the non-zero cells of the column. When answering a query, the system would consult first the bitmap to see if the needed point is zero, and in the case it is not, would check if it is a retained point. If none of these cases apply, then the point would be estimated using the stored parameters.

We now need to extend the method to higher dimensional cubes.

2.2.1 Higher Dimensional Matrices

There are two ways of dealing with cubes of dimension $\eta > 2$:

- **Multiregression** For each value s in the domain of one of the dimensions, represent the sub-cube formed by the other $\eta - 1$ dimensions by a multiregression on the marginal distributions for each of the dimensions. For instance, for a three-dimensional matrix, one could: For each point s in the third dimension, represent the corresponding plane by a multiregression of the form:

$$p_{ijs} = b_0 c_j + b_1 r_i + b_2 \tag{4}$$

  where c_j is the marginal distribution for column j in plane s and r_i is the marginal distribution for row i in plane s. With this method, each value in the domain of the third dimension would be described by four values: b_0, b_1, b_2, and $stdev(s)$.

- **Divide-and-conquer** Regard the η dimensional cube as a set of q sub-cubes of dimension $\eta - 1$. Do this recursively until the sub-cubes have two dimensions. Then represent the two dimensional matrices using the method we described in this section.

  Again, for a three-dimensional cube, we could describe each plane s, as a two dimensional matrix, representing each column j in the plane with a simple regression of the form of Equation 2.

Multiregression as a strategy becomes more complex as the number of dimensions increases. However, for the same percentage of specified points, the description of the Quasi-Cube is more compact when using the multiregression approach than in the case of divide-and-conquer.

Another issue in the selection of the method to be employed for higher dimensions is the space and time needed to build the Quasi-Cube. In the case of multiregression, the space is usually greater than in the divide-and-conquer approach, since we need to keep around all the points involved in the sub-cube on which the multiregression is to be performed. However, the divide-and-conquer approach requires the computation of regression parameters for **every plane** in the cube, a process that can potentially slow down the Quasi-Cube construction.

2.3 Singular Value Decomposition

Singular Value Decomposition (SVD) [20, 25] is a technique that has been used to approximate matrices, perform statistical analysis [15], text retrieval [8] and dimensional reduction [7]. Recently, Korn et al [17], published a technique to use SVD to compress large matrices into a format that supports approximate queries.

The formal definition of SVD follows. Given an $N \times M$ matrix of reals X, we can express it using Equation 5, where U is a column-orthonormal (its columns are mutually orthogonal unit vectors) $N \times k$ matrix, $\bigwedge$ is a diagonal $k \times k$ matrix of the eigenvalues λ_i of X, and V is a column-orthonormal $M \times k$ matrix.

$$X \;=\; U \times \bigwedge \times V \tag{5}$$

Equation 5 can be alternatively written as the *spectral decomposition* form, shown in Equation 6, where u_i and $V - i$ are column vectors of U and V respectively, and λ_i the diagonal elements of the matrix $\bigwedge$.

$$X = \lambda_1 u_1 \times v_1^t + \lambda_2 u_2 \times v_2^t + ... + \lambda_k u_k \times v_k^t \tag{6}$$

The matrix X can be approximated by selecting the κ largest values of the set of λ_is. Doing this, we need $N\kappa$ data elements from the U matrix, κ entries for the eigenvalues and κM data elements from the V matrix. So, the total number of parameters used to represent the original matrix X is given by Equation 7.

$$P_{svd} = N\kappa + \kappa + \kappa M \tag{7}$$

Korn et al [17] propose a method, called SVDD, that gives better performance than just approximating the matrix by truncating the number of eigenvalues used. In SVDD, the entries that give the greatest estimation errors are retained (as we do in the previous two methods), to guarantee that the error incurred by the estimation is kept low.

As we shall see in the experiments section, SVDD can be successfully used to model planes of a cube, giving excellent error bounds. To use SVDD for cubes of more than two dimensions, we have to apply the divide-and-conquer method presented in Section 2.2.1. SVDD, however, has the following drawbacks:

- The running time of the algorithm is greater than that of linear regression or the independence approach. Although approximate, randomized solutions have been proposed for SVD [19], these solutions are only asymptotically superior and their benefits have not been studied in practice yet.
- There is no known method to extend SVD to higher-dimensional matrices. Therefore, we are forced to use divide-and-conquer to deal with cubes of more than 2 dimensions. Given the running time of the application of SVD for each plane of the cube, the overall running time to model the cube may be prohibitive in real life cases.
- There is no way of taking advantage of spareness in SVD. Even if a cell is 0, it will have to participate in the computation. This is a serious drawback, since, as we saw in the linear regression case a better performance can be achieved if the modeling does not have to be performed including cells that have 0 as their values. (Of course, cells with zero value can be marked as such using a bitmap, as we did for linear regression, thus eliminating the need to estimate them, but the whole plane of cells will be used to find the eigenvalues.)
- Besides having to use zero-valued cells in the computation, the estimated values obtained in sparse matrices are worse than those obtained for dense matrices (as we will show in the experiments section).

2.4 Implementing Quasi-Cubes

The previous subsections described several algorithms for reconstructing cubes from incomplete descriptions. In this section, we outline the implementation details of Quasi-Cubes.

We assume that the cell values need to be computed from the fact table. We aim for a construction technique that requires one pass over this relation. On that pass, the entries that would cause the greatest errors when applying Algorithm *REG* are selected to be part of the specification. We assume in this section that we are using the regression technique to model regions of the cube. Notice that, when using regression, unlike the independence approach, retaining an entry does not change the errors incurred on the estimation of the entries left unspecified. The selection procedure is as follows:

1. Select the size of the regions to be modeled.
2. For each region:
   (a) Compute the cell values in the region by aggregating tuples in the fact table.
   (b) Compute the regression parameters, assuming no items are retained. (I.e., all non-zero cells in the column enter in the regression calculation.) Compute the sign vectors as well.
   (c) Estimate all the cells in the region by using regression.
   (d) Compute the errors incurred by the procedure as the modulo of the difference between the estimated value and the known value for the entry. For each entry, try adding and subtracting the value $stdev(j)$, and keep the smaller of the two errors, also recording the sign (if the addition results in the smaller error, the sign is positive, otherwise it is negative) in $sign(i,j)$.
   (e) For each cell, if the error is bigger than or equal to a predefined threshold value, store the cell value.

A lot of the work of building a Quasi-Cube and a conventional cube is the same. In both we need to calculate of the aggregates (cell values) from the underlying facts table first. After that, there is a tradeoff between conventional cubes and Quasi-Cubes. In Quasi-Cubes we increase the CPU overhead by computing the regression parameters (or the parameters of whatever model we are using), but we store only a fraction of the cell values. In the case of a conventional cube, all the cell values will be stored. We will report the overhead incurred by the construction of the Quasi-Cube in the experiments section.

Temporary space is also a concern when building a Quasi-Cube. We need enough space in main memory to hold the cells of the region we are currently processing. In practice, since cubes are sparse, this requirement is not very strict. All we need to store temporarily is the non-zero cells and their position in the region. The position can be stored as an array of integers that correspond to a predetermined way of traversing the cube. I.e., the integer x could be translated to a set of cube coordinates by the successive application of the algorithm shown in Equation 9, where $coord(i)$ is the position in the ith coordinate and $dim(i)$ the cardinality of that dimension.

$$\begin{aligned} coord(i) &= \frac{x}{\prod_{j=0}^{i-1} dim(j)} \text{ if } i \neq 0 \\ & \quad x \text{ if } i = 0 \\ x &= x \ \% \prod_{j=0}^{i-1} dim(j) \end{aligned} \tag{8}$$

For instance in a three dimensional cube with cardinalities 2, 3 and 5 respectively, the cell $(1,1,4)$ would occupy the position $x = 4 \times 3 \times 2 + 1 \times 2 + 1 = 27$. Transforming x into coordinates would result in the following operations:

$$
\begin{aligned}
coord(2) &= \frac{27}{6} = 4 \\
x &= 27 \ \% \ 6 = 3 \\
coord(1) &= \frac{3}{2} = 1 \\
x &= 3 \ \% \ 2 = 1 \\
coord(0) &= 1
\end{aligned} \tag{9}
$$

We take advantage of the fact table being sorted according to one of the dimensions (this is a common occurrence: for instance, if time is one of the dimensions, the table is naturally sorted on the time values). Then we select the regions to be modeled as be subsets (not necessarily proper subsets) of the regions described by each value in the domain of the sorted dimension (a range of values works equally well). We bring all the tuples corresponding to that value to memory and perform the aggregations there. Notice that the regions can be defined as the cubettes formed by the particular value (or values) of the sorted dimension or as subsets of those cubettes, if further restrictions on the other dimensions are imposed. If the regions are proper sub-cubettes (i.e., proper subsets of the cubettes), then the tuples of the fact table corresponding to the value(s) of the sorted dimension stay in main memory until all the sub-cubettes corresponding to this value(s) have been processed. The sub-cubettes are built following an algorithm similar to the one presented in [23]. We make a pass through the set of tuples in memory. For each tuple of them we compute the coordinates of the cell that the tuple contributes to. If the coordinates do not fall inside of the sub-cubette being processed, the next tuple is selected. Otherwise, if there is a counter already allocated for this cell (i.e, these coordinates), we increment it using the value of the aggregate attribute for that tuple. (The presence of a counter for this cell means that a previous processed tuple corresponded to the same cell.) If not, then we create a new counter and initialize it with the value of the aggregate attribute for that tuple. Notice that the final number of counter corresponds exactly to the number of non-zero cells in the sub-cubette. Along with the counters we keep an array of positions that identify the cell that each counter represents. This algorithm performs very well for sparse cubes, since the number of counters per sub-cubette will be small.

Notice that we also need the temporary space in main memory to store the tuples of the fact table that are going to be modeled next. Either the number of tuples that correspond to the region fit in main memory, or we bring them in blocks of tuples that do fit in memory. Since we only execute one pass over the set of tuples, in the later case we can discard a block of tuples as soon as it is processed and bring the next one to memory.

Once the description of the Quasi-Cube is completed, we need to store it in a compact form. The regression parameters and row marginals are stored contiguously. They are few enough that they can be cached in main memory at run time. The specified entries may be too large to cache, however, so a more careful organization is needed. We group them into variable-sized segments, each segment containing selected points for a region of the cube. An index of regions is built indicating where in the disk the regions are stored. Like the regression parameters the index can fit in main memory at run time. Alternatively, we could simply use a global index that stores each retained cell using its coordinates in the cube. There have been many proposals to index cube data ranging from the usage of standard R-trees and bit-mapped indices ([21]) to specialized structures ([14]).

As for the sign data, it can be stored as a bitmap, similar to the one used to indicate whether cells are zero or non-zero. Both bitmaps can be heavily compressed to reduce their storage requirements.

2.5 Space Savings

This subsection deals with the gains in space achieved by Quasi-Cubes. The space taking by a Quasi-Cube can be broken into the following items:

- Space for the retained cells. This can vary according to the way retained cells are stored. A simple structure (such as the one explained in Section 2.4) would require two values per cell: one for the actual entry and one for the offset. A more elaborate structure may include an index (such a B-tree) which will add space overhead to the space needed for the cells alone. Let us call this space S_r.
- Space for the model parameters. In the case of regression that space amounts to the marginal sums (or distributions) plus the estimated parameters of the regression model. For SVD, we need to store k columns of the matrices U and V, plus k eigenvalues. Let us call this space S_p.
- Space taken by bitmaps. The bitmap needed to deal with the sparseness will occupy some space (although this space will not be very significant compared with the other requirements). Also, in the case of regression, the sign vectors can be encoded with a bitmap whose space overhead will be roughly the same taken by the sparse bitmap.

2.6 Incremental Updates

Since the data in a warehouse is bound to change incrementally, we need to discuss how to reflect those changes in the Quasi-Cube, without having to rebuild it entirely.

We discuss how to do this when using the regression method. For SVD it is not clear how to incrementally update the model. The first, and easiest possibility is that the cells that we use in the regression procedure are immutable. For instance, if one is using the divide-an-conquer approach and the row marginals are kept for a dimension such as time, we do not need to worry about the entry points ever changing. (E.g., sales of a given product for a given store at a given day in the past will never change.) However, new rows will be incrementally added to the cube (e.g., the next day of sales aggregates). In order to deal with these new values, one can proceed by estimating the new points using the regression parameters computed for the original column. One of two things can happen: the error is acceptable and then we can simply discard the point, or the error is too high and we keep the point as a selected one.

For cases in which the old aggregates themselves can change, we can follow a similar procedure: compute the estimate of the changed point and compare it with the new value. If the error incurred is bigger than the level of error that one wants to support the new cell needs to be retained (if it was not retained before). Otherwise, the cell will be correspondingly estimated when needed. This, may, in the long run, cause deterioration of the model parameters (as new points are incorporated, the old parameters may not be able to properly model the column anymore), causing the system to retain, perhaps unnecessarily, many cells. So, an eventual recomputation of the affected models is advisable. The updating of the model can be achieved by using techniques similar to those described in [4] to update polynomial models for selectivity estimation. The techniques use a method called recursive least-square-error [28] to avoid a lot of expensive recomputation.

2.7 Query processing

Once the Quasi-Cube is constructed, we can use it to answer queries. Any query will request a chunk of the matrix entries which spans one or more of our regions. For each cell in the query, we use the index to decide if the entry is in the list of retained points or not. (For some recent work in how to index cubes see [14, 21].) If it is not, we use the model parameters to compute it. The I/O required to fetch the retained values dominates the performance of the query. Obviously, the performance is going to be heavily affected by the match between the choice of region dimension and the query workload.

3 Evaluation

In this section, we present the results of a series of experiments in which we partially specified matrices and used the estimation algorithm to reconstruct them. We will measure the accuracy of the estimation along with the space savings achieved by the method. Before presenting the experiments themselves, however, we first describe the metrics used to compare the estimated matrix to the original one.

3.1 Comparison Metrics

In what follows, $g_{\hat{i}}$ is the original entry in the original cube, $\hat{i}$ is the vector of coordinates that defines the cell position. $total = \sum e_{\hat{i}}$ is the sum of all the cell values in the cube, $p_{\hat{i}}$ is the corresponding value in the original probability matrix (i.e., $p_{\hat{i}} = \frac{e_{\hat{i}}}{total}$), and $e_{\hat{i}}$ is the estimated cell probability value. (Notice that for retained entries, $e_{\hat{i}} = p_{\hat{i}}$.) Finally, N is the number of non-zero cells in the cube.

The first comparison metric is the normalized average absolute error. To define this metric we first define the average absolute error as:

$$AVGERR = \frac{\sum |total * (e_{\hat{i}} - p_{\hat{i}})|}{N} \tag{10}$$

$AVGERR$ is then the expected value of the absolute differences between the entries in the two matrices. To make the value of the average absolute error comparable across different matrix sizes, we normalize it by dividing it by the average entry value in the aggregates matrix ave, given by:

$$ave = \frac{\sum g_{\hat{i}}}{N} \tag{11}$$

The value ave, of course, depends on the distribution of values. Dividing $AVGERR$ by ave we obtain a metric which we call $\bar{A}$.

$$\bar{A} = \frac{AVGERR}{ave} \tag{12}$$

We are also interested in the cumulative error, which is defined as

$$CUMERR = \sum |e_{\hat{i}} - p_{\hat{i}}| \tag{13}$$

The value $CUMERR$ has two useful interpretations. First is the total error incurred by the estimation process: if this value is low we know the estimation is a good one. The second interpretation can be found if we multiply and divide the term inside the sum of Equation 13 by $p_{\hat{i}}$. That is,

$$CUMERR = \sum (\frac{|e_{\hat{i}} - p_{\hat{i}}|}{p_{\hat{i}}}) p_{\hat{i}} \tag{14}$$

The term $(\frac{|e_{\hat{i}} - p_{\hat{i}}|}{p_{\hat{i}}})$ is simply a random variable that corresponds to the fractional error. That is, the error incurred as a fraction of the probability entry $p_{\hat{i}}$. Multiplying that term by the corresponding probability and adding over all the values, we get the expected value of the fractional error. Then, $CUMERR$ can be also interpreted as the expected fractional error.

Finally, we are also interested in the normalized maximum error that can be incurred, given by the following equation:

$$\bar{M} = \frac{\max_{i,j}(|total * (e_{\hat{i}} - p_{\hat{i}})|)}{ave} \tag{15}$$

3.2 Experiments

The experiments presented in this section were conducted on a Sun UltraSparc 2 machine with 256 Mbytes of RAM, running SunOS 5.5.1 (Solaris).

3.2.1 Error Metrics

We wanted to evaluate the errors incurred by our method using both synthetic data and real data. Synthetic datasets allowed us to experiment with a variety of data distributions and see

Uniform (0-200)
Normal (mean 100, std dev 20)
Normal (mean 100, std dev 10)
Normal (mean 100, std dev 2)
Correlated Row/Col ($2 * X + Y \pm 5$))
Bimodal
Zipf(0.8,1)
Zipf(0.8,10)

Table 1: Distribution of the entries in our test matrices

how they affect our method. The real dataset allowed us to check our method for real-world data.

For the synthetic data cubes, the aggregate values in the cube were drawn from one of the five different data distributions listed in Table 1. The first four are standard probability distributions in which each cell is calculated independently from its neighbors. Note that our technique performs better when there is more structure to the data, so Uniform Distribution is the worst case. The three Normal Distributions differ only in the standard deviation. The correlated distribution gives a matrix in which the aggregate value is roughly a linear combination of the row and column values. The final distribution is a bimodal one, composed of two normal distributions of means 105 and 55 respectively, with standard deviations of 2 for each of them. Ten percent of the points are generated with the distribution of mean 55 and the remaining 90 % of the points obey the other distribution. The overall mean (as in the other distributions) is 100. The last two are Zipf distributions. In both, 80% of the time, a number from a given set is generated, of the remaining 20%, 80% of the time, a number from a second set is generated, and so on. The first set in the first Zipf distribution contains a single number N, the second contains the number $N + 1$, and so on. In the second Zipf distribution, the first set contains 10 numbers: $N, N + 1, ..., N + 9$, the second set another 10: $N + 10, N + 11, ..., N + 19$, and so on. In both distributions N is chosen so that the mean of the distribution is 100.

Figures 11,12 and 13 show the results for our three metrics. Each graph gives the accuracy metric on the Y axis. The X axis in each case is the fraction of the original cube retained. For each data point, we generated six different cubes (using different seeds for the data generators) and computed the mean value. In all cases, the standard deviation over the six runs was much less than 1% of the mean. First, we tested the results over two-dimensional cubes of dimension 100, 200, 500, 1,000 and 10,000. The results were indistinguishable for different matrix sizes (remember our metrics are scaleless), so we present only the 10,000x10,000 results in the figures. We also tested our results for a three dimensional matrix of size $100 \times 100 \times 100$, using the multiregression method. The results are indistinguishable from those obtained for two dimensional matrices. We also performed experiments using the divide-and-conquer approach on cubes with several dimensions. In each case, the results were indistinguishable from the previous ones. The biggest cube we experimented with had 10 dimensions, 8 of them with a domain of 5 values and 2 of them with a domain of 1,000 values (the complete cube would have $390,625 \times 10^6$ points and occupy 2.84 Terabytes of storage). Finally, we perform again all the tests for cubes where only 10the cells were non-zero. For all the sizes and dimensions mentioned above, the results remain unchanged. (Of course, the X-axis in

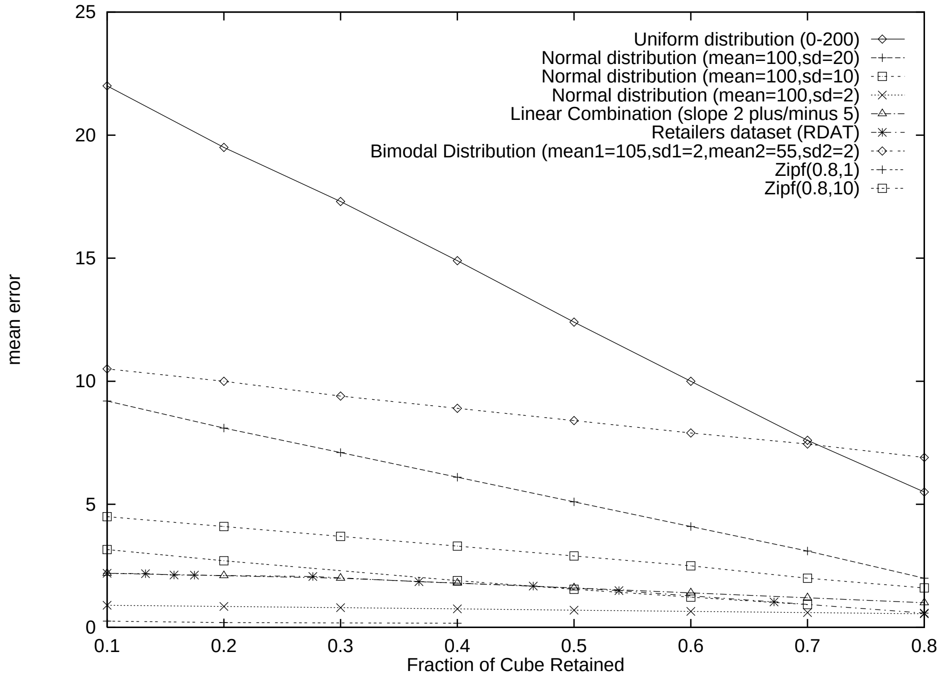

Figure 11: Mean Error for Estimated Cube Entries ($\bar{A}$), using regression. The X axis gives the fraction of the matrix that is specified in the Quasi-Cube. The Y axis is the error in the entry value as a percentage of the mean entry value.

those cases represents the percentage of retained points with respect to number of non-zero cells.)

The graphs of the regression Quasi-Cubes each have roughly the same shape. The mean error, maximum error and total error probability decrease as the data becomes less random and as more of the matrix is specified in the Quasi-Cube.

For the normal and correlated matrices, the cube is easily characterized by the regression algorithm and the Quasi-Cube performs well. The normal with the smallest deviation has a mean error of about 1% which, not surprisingly, does not decrease much as more of the matrix is specified. For the input matrices with larger standard deviation, the mean error decreases linearly as more of the matrix is specified. For the normal with standard deviation of twenty, the mean error varies between about 9% and 3%. Even in our worst case, the Uniform Distribution, the mean error never goes beyond about 23%. As more of the matrix is specified, the mean error drops to about 15% when half of the matrix is specified and 5% when 80% is specified. For the correlated matrix, the correlation is predicted exactly by the linear regression and the ± 5 noise we added to each point in the matrix determines the mean error. In all cases, the maximum error is roughly twice as much as the mean error in each of these cases, hence it drops more quickly as more of the matrix is specified. We see that in the Bimodal distribution, the maximum error drops significantly when going from 10% to 20% of the matrix being specified. In this case, the second part of the distribution (mean 55 with 10% of the points) drags the regression coefficients, causing some of the points in the first part of the distribution (mean 105, 90% of the points) to be wrongly estimated. After roughly 20% of the entries have been specified, most of the points obeying the smallest distribution are already retained, but the effect of them on the regression coefficients lingers, making the

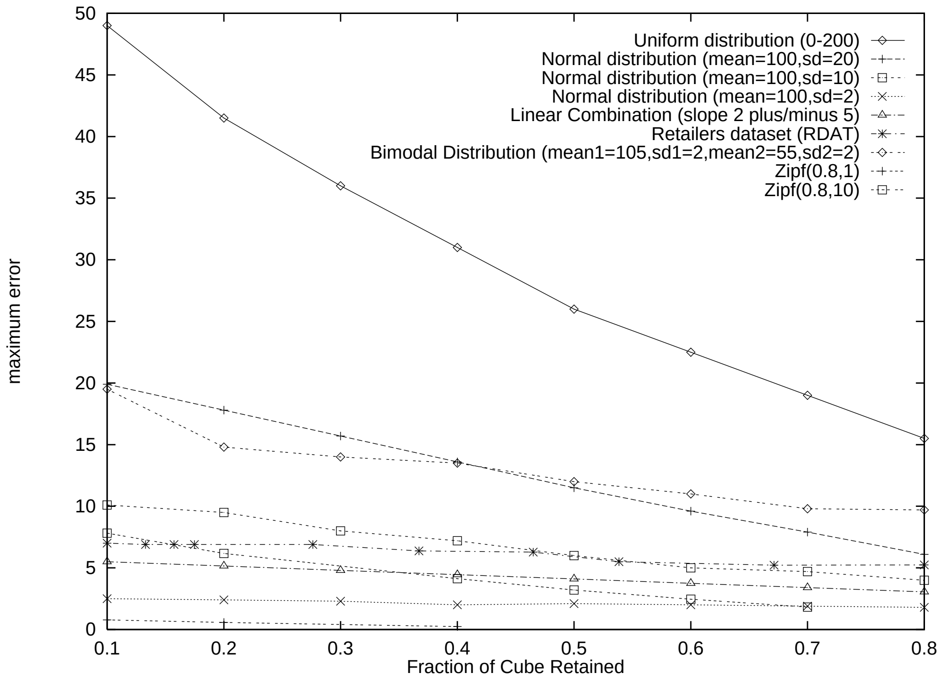

Figure 12: Maximum Error for Estimated Cube Entries ($\bar{M}$), using regression. The X axis gives the Fraction of the matrix that is specified in the Quasi-Cube. The Y axis is the maximum error in the entry value as a percentage of the mean entry value.

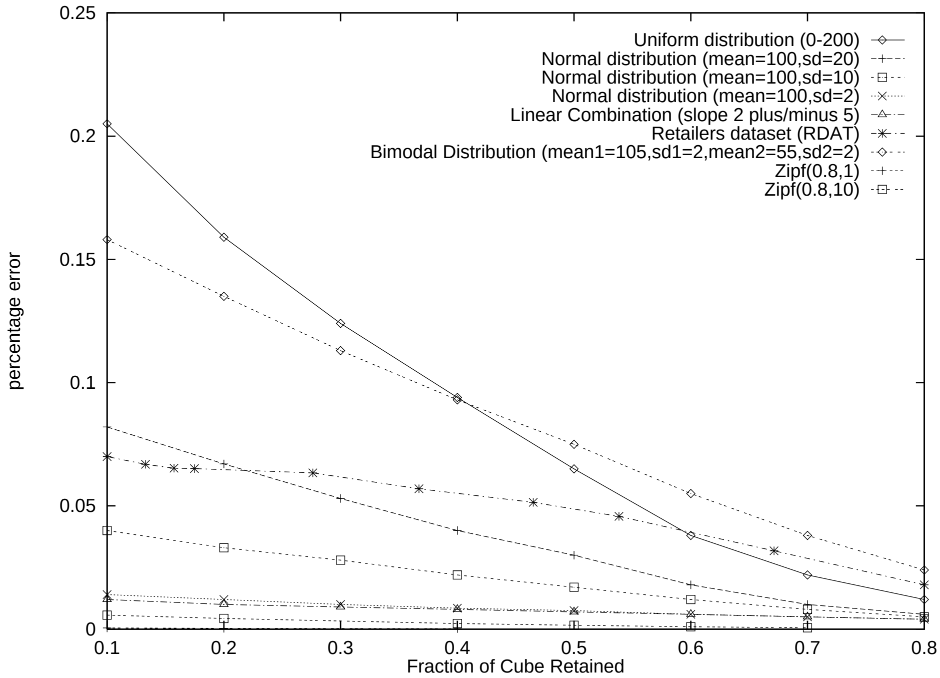

Figure 13: Cumulative Error ($CUMERROR$), using regression. The X axis gives the fraction of the matrix that is specified in the Quasi-Cube. The Y axis is the cumulative error in the entry values of the probability matrix corresponding to the Quasi-Cube.

drop of the error less pronounced than in the other curves. Nevertheless, even with this badly behaved distribution our method performs extremely well: the mean error never exceeds 11 %. The curves corresponding to the Zipf distribution show the method does extremely well: the maximum error for Zipf(0.8,1) and Zipf(0.8,10) are 1% and 7the fact that most of the numbers generated by this distributions are clustered in a small region, making the linear regression model fit the data very well. Although the distributions have a long tail, the values in the tail are not many and can be easily retained.

The curve(s) labeled RDATA in Figures 11,12 and 13 corresponds to a real data set which contains data from a survey of retailers. From each tuple of the dataset seven attributes were used (the rest of the data in the tuple is textual). The first six are dimension attributes (such as product, store) and the seventh one is the attribute aggregate attribute (i.e., sale amount). The corresponding cardinalities of the attributes are 597, 42, 966, 1764, 4, 69. The size of each tuple is 102 bytes and the dataset had a total of 764,993 tuples (i.e., 78,029,286 bytes). The complete cube for this data set would have 11,792,568,586,176 cells, but since the data is sparse only 335,324 cells are non-zero. Quasi-cubes of various error levels were built for this dataset in a manner that will be described in Section 3.2.3. We modeled 597 cubettes (one per each value of the first attribute) of size $42 \times 966 \times 1764 \times 4 \times 69$ each one, using multiregression on the five dimensions. Here we wanted to report the error levels vs. the fraction of points in the cube that were retained, to compare that performance with that of the synthetic data. As can be seen in the figures, the errors obtained with this dataset are small (considerably less than the worse case represented by the uniform distribution).

Unlike regression, SVD gives different error results for different matrix sizes and shapes. In Figures 14,15 and 16 we show the results of using SVD to model planes of size 1000×100 in the cubes. (I.e., we tried cubes of several dimensions in which two of the dimensions had cardinalities 1000 and 100 respectively.) Figures 17, 18 and 19 show the results obtained when modeling planes of size 100×100 in the cubes. All these results were obtained by generating cubes using the distributions shown in Table 1, in the same way we did for the regression experiments. For each cube, we use the divide-and-conquer approach, modeling planes of the cube. In each case the number of eigenvalues used is 10 % of the number of rows in the matrix. Square matrices always perform worse than rectangular ones for SVD. (It is also true that for rectangular ones we obtain a better space reduction, since we only need $N \times k + k + k \times M$ parameters for the estimation, and $M < N$.)

As we can see in the figures, the percentage of points needed to be retained to achieve the same level of errors we got with the regression approach is smaller. Although the errors for the other distributions are very encouraging, we should point out that, given the increased running time of SVD when compared with regression (we will show the figures later on in this section), we were not able to try SVD in cubes as big as we used for the regression approach. We tried different cubes from two dimensional ones (of sizes 100×100, and 1000×100), to cubes with many dimensions, obtaining similar results in every case. We also tried sparse cubes (in which the fraction of non-zero cells was 10% of the total number of cells) obtaining similar results. (The biggest cube we tried had 5 dimensions 3 of them with 5 values, one with 100 values and one with 1,000 values, for a total of 12.5 million cells.) Finally, Figure 20 reveals a serious drawback of the SVD approach. The curves show the maximum error as a function of the percentage of points retained for three planes of size 100×100 with spareness $0\%, 50\%$ and 90% respectively. The measurements show that as the matrices become more sparse, the results are worse. I.e, the percentage of points needed to retain the same level of

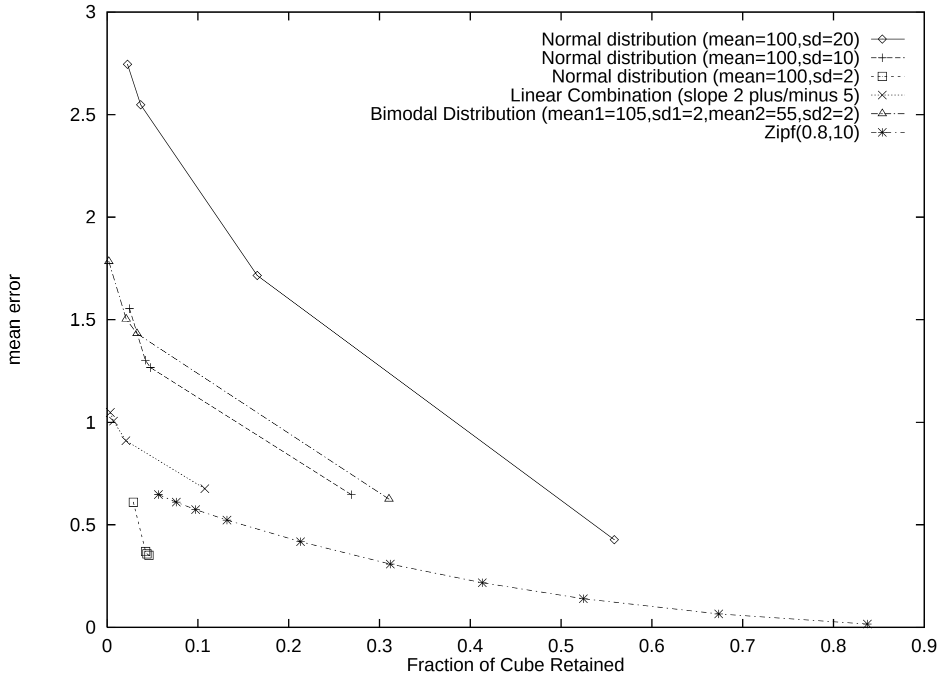

Figure 14: Mean Error for Estimated Cube Entries using SVD in cubes with planes of size $1000 \times 100(\bar{A})$. The X axis gives the fraction of the matrix that is specified in the Quasi-Cube. The Y axis is the error in the entry value as a percentage of the mean entry value.

error increases drastically. In other words, SVD, does not deal very well with sparse cubes.

We did not conduct tests over RDATA using SVD for the following reasons:

1. We would have had to model the cube plane by plane. Due to the spareness of the dataset, each plane has very few non-zero cells (usually 1 or 0). SVD is ineffective in that case, since the estimation of the cells gives high errors and the algorithm ends up retaining most of the cell values.

2. The overhead imposed by using SVD is considerably higher than that caused by regression. This along with the need of modeling the cube by planes, would have resulted in an extremely large time to build the Quasi-Cube for RDATA.

When Quasi-Cubes are constructed using the independence assumption, the mean error is slightly worse than in the regression approach. However, the maximum error is significantly worse (see Figure 21). Also, for some of the distributions, the behavior is erratic (bimodal, correlated): the error can go up as more points are specified. This is due to the property that we mentioned in Section 2: when using the independence approach, retaining a cell has a impact on the errors incurred in estimating the others. The worst part about the independence Quasi-Cube is that the more "interesting" the data in the underlying database, the worse the Quasi-Cube will behave. The graphs of Figure 21 were obtained using the heuristic described in Section 2.1.

3.2.2 Space Savings

Figure 22 shows how much storage is saved by Quasi-Cubes using the regression approach. In this graph, the X axis shows the maximum acceptable error. The Y axis shows what percent

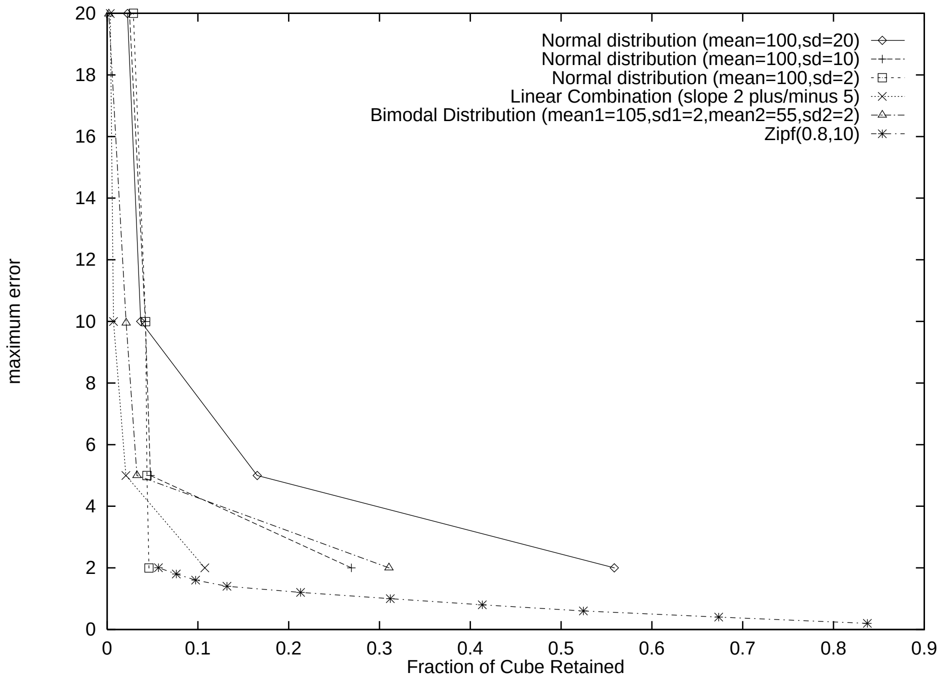

Figure 15: Maximum Error for Estimated Cube Entries, using SVD in cubes with planes of size 1000×100 ($\bar{M}$) The X axis gives the Fraction of the matrix that is specified in the Quasi-Cube. The Y axis is the maximum error in the entry value as a percentage of the mean entry value.

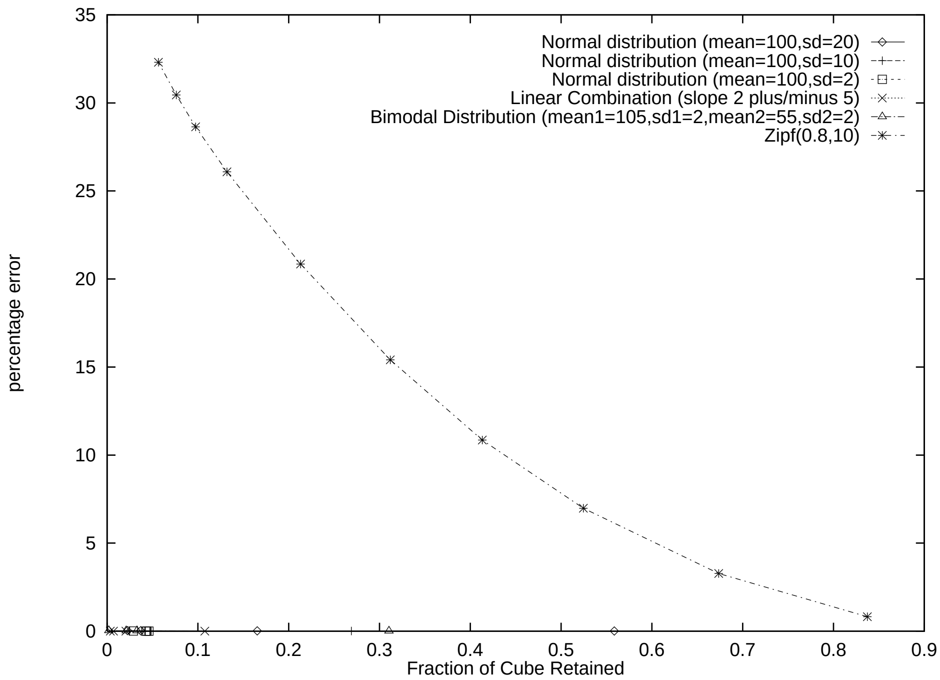

Figure 16: Cumulative Error using SVD in cubes with planes of size 1000×100 ($CUMERROR$). The X axis gives the fraction of the matrix that is specified in the Quasi-Cube. The Y axis is the cumulative error in the entry values of the probability matrix corresponding to the Quasi-Cube.

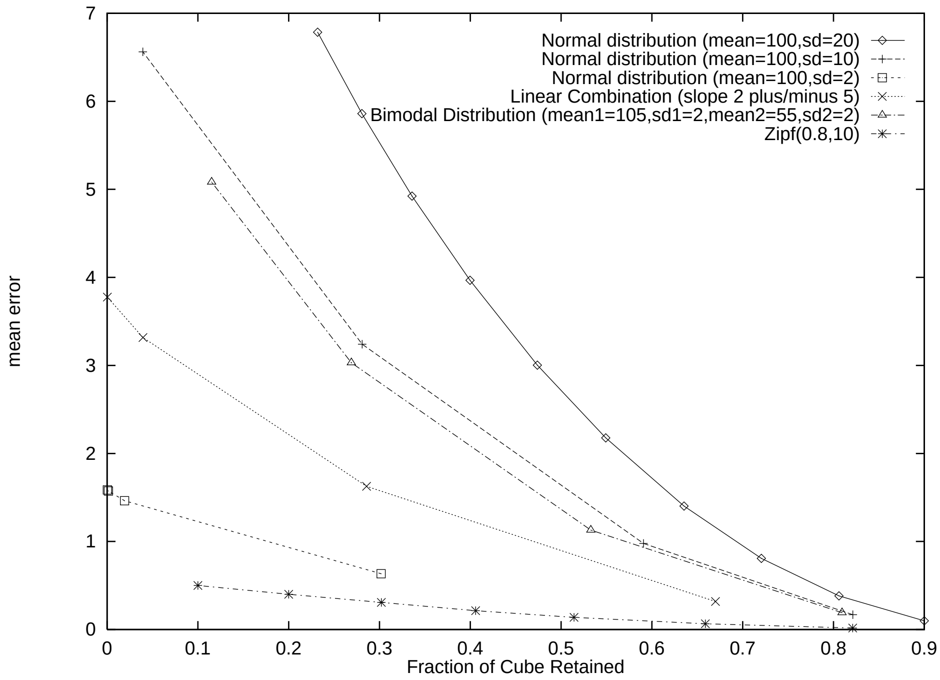

Figure 17: Mean Error for Estimated Cube Entries using SVD in cubes with planes of size $100 \times 100(\bar{A})$. The X axis gives the fraction of the matrix that is specified in the Quasi-Cube. The Y axis is the error in the entry value as a percentage of the mean entry value.

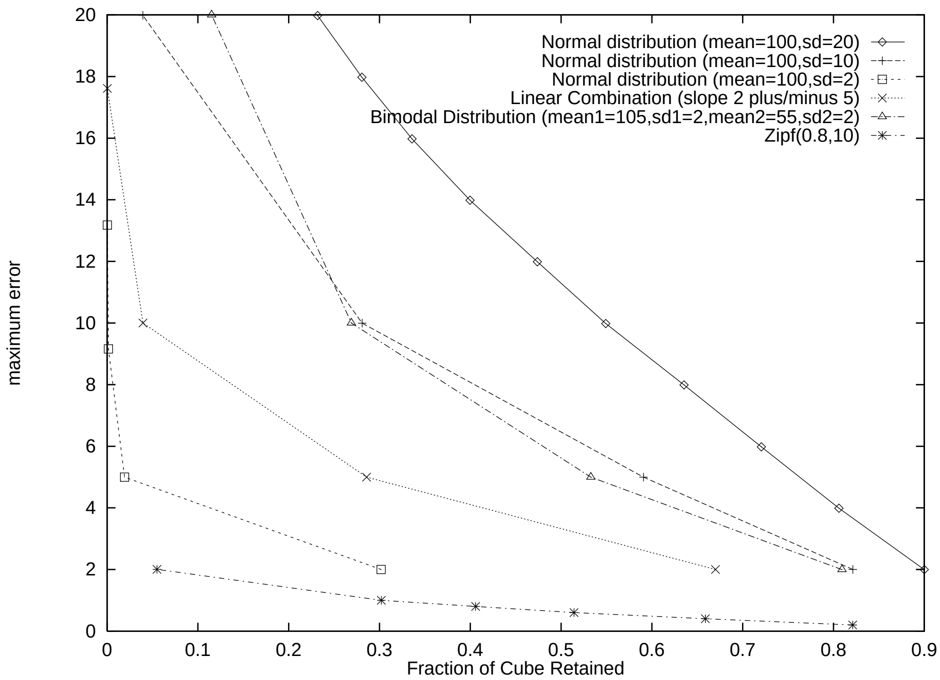

Figure 18: Maximum Error for Estimated Cube Entries, using SVD in cubes with planes of size 100×100 $(\bar{M})$ The X axis gives the Fraction of the matrix that is specified in the Quasi-Cube. The Y axis is the maximum error in the entry value as a percentage of the mean entry value.

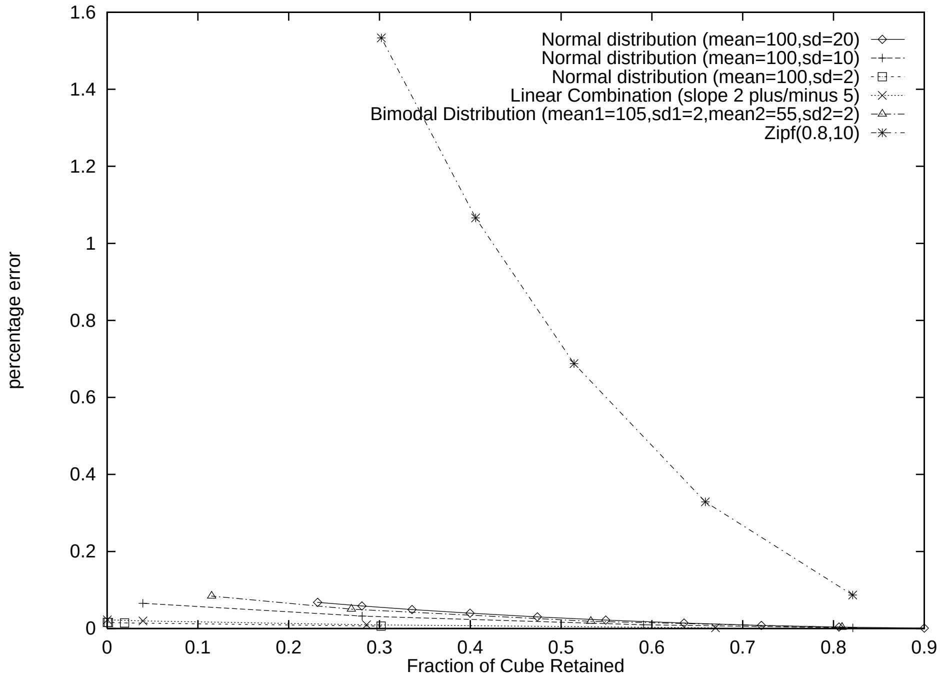

Figure 19: Cumulative Error using SVD in cubes with planes of size 100×100 ($CUMERROR$). The X axis gives the fraction of the matrix that is specified in the Quasi-Cube. The Y axis is the cumulative error in the entry values of the probability matrix corresponding to the Quasi-Cube.

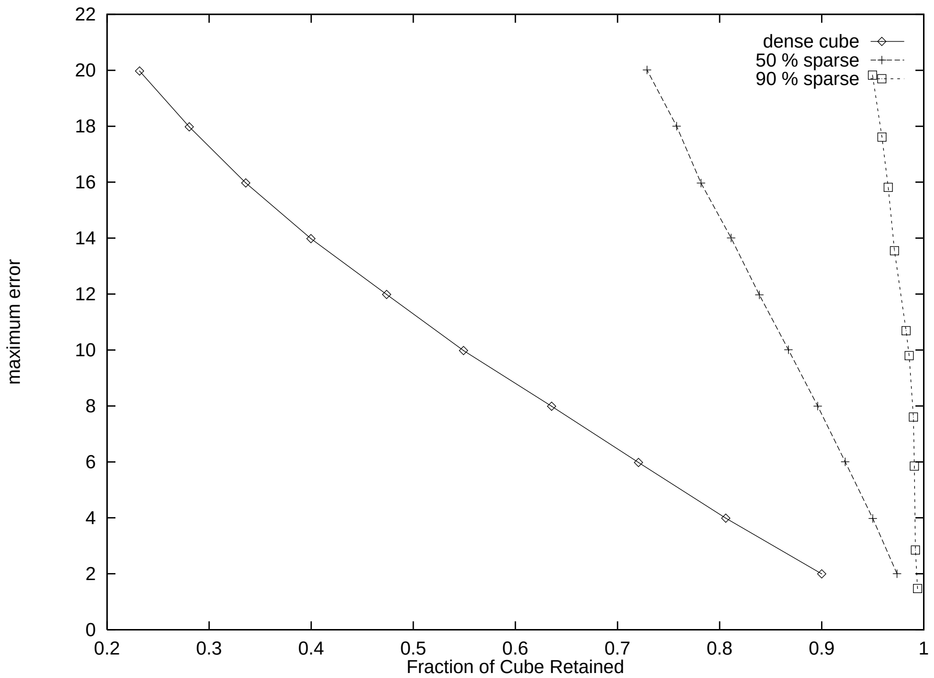

Figure 20: Maximum Error for Estimated Cube Entries, using SVD in cubes with planes of size 100×100 ($\bar{M}$) for three different levels of spareness.The term $x\%$ sparse means that x out of each 100 cells are zero. The X axis gives the Fraction of the matrix that is specified in the Quasi-Cube. The Y axis is the maximum error in the entry value as a percentage of the mean entry value.

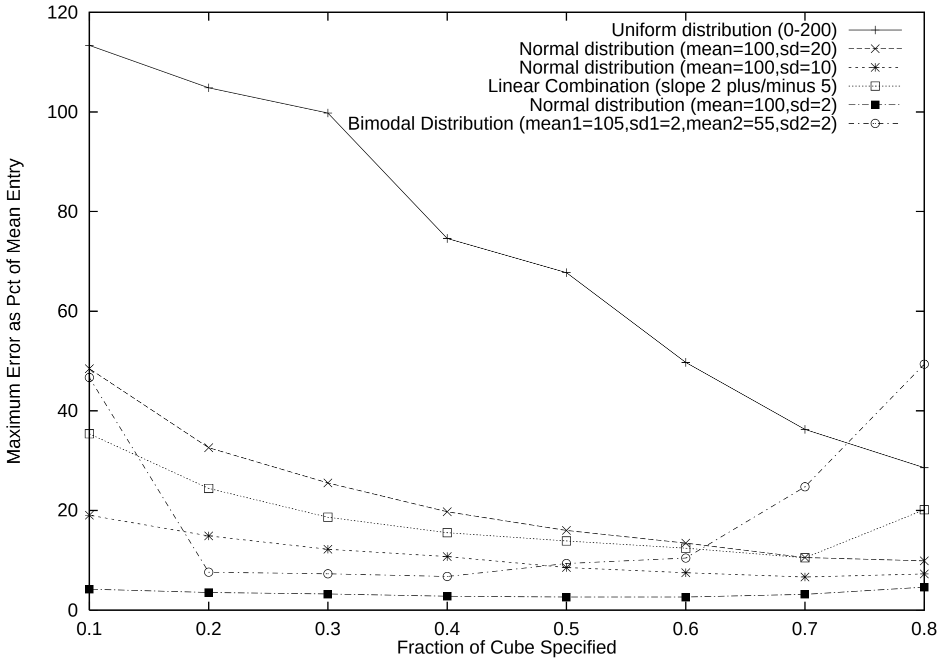

Figure 21: Maximum Error under the Independence Assumption. The X axis gives the fraction of the aggregate matrix that is specified in the Quasi-Cube. The Y axis is the maximum error for any entry. In these experiments we used the independence assumption to select specified points and to estimate the unspecified ones

of the storage required by a full cube one saves using the Quasi-Cube. When the maximum acceptable error is 3% and the input matrix is normally distributed (mean 100, sd 2), the space savings is 0.85. Thus, a Quasi-Cube for this data will use 15% of the storage required by a full data cube.

Figure 23 shows the space savings obtained when using SVD on cubes with planes of dimension 1000×100. Notice that the gains observed in terms of fraction of retained points needed for a given error (Figure 15) are somewhat offset by the need of keeping more parameters than in the regression approach. Nevertheless, the overall space savings for SVD are greater than those obtained by the regression approach.

3.2.3 Building Quasi-Cubes

To measure the overhead imposed by modeling sections of the cube in the total time of building the Quasi-Cube, we implemented a prototype that follows the description of Section 2.4. The system can build regions of the data cube in memory by aggregating the appropriate tuples from a fact table. Regions are computed by computing the aggregate values for each cell in the region, using the tuples in the fact table in the manner explained in Section 2.4. To improve performance, regions are characterized as containing all the tuples for one (or more) values of the first dimension attribute. Since the fact table is ordered by the values of the first dimension, it is possible to load at once all the tuples that contain an specific value(s) for that attribute and proceed to compute the aggregates in memory. Once the region is computed, the modeling process over that region takes place. Those cells whose errors are larger than a prespecified error are retained. For the rest of the cells, the errors are evaluated

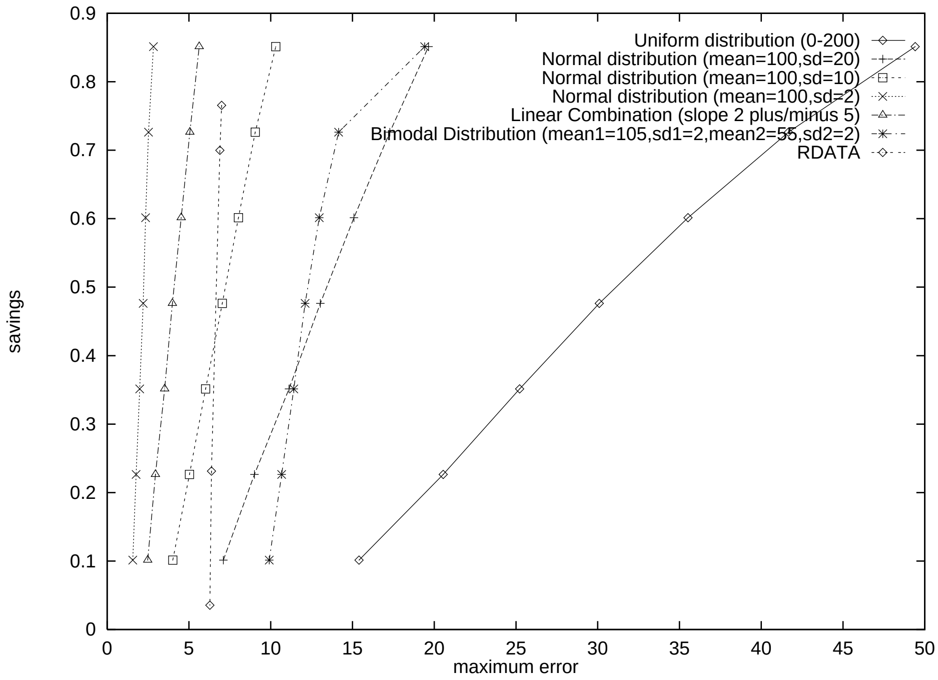

Figure 22: Space Savings (f_s) using the regression approach. The X axis gives the maximum error. The Y axis is the space savings obtained by the Quasi-Cube as a fraction of the total space taken by the cube.

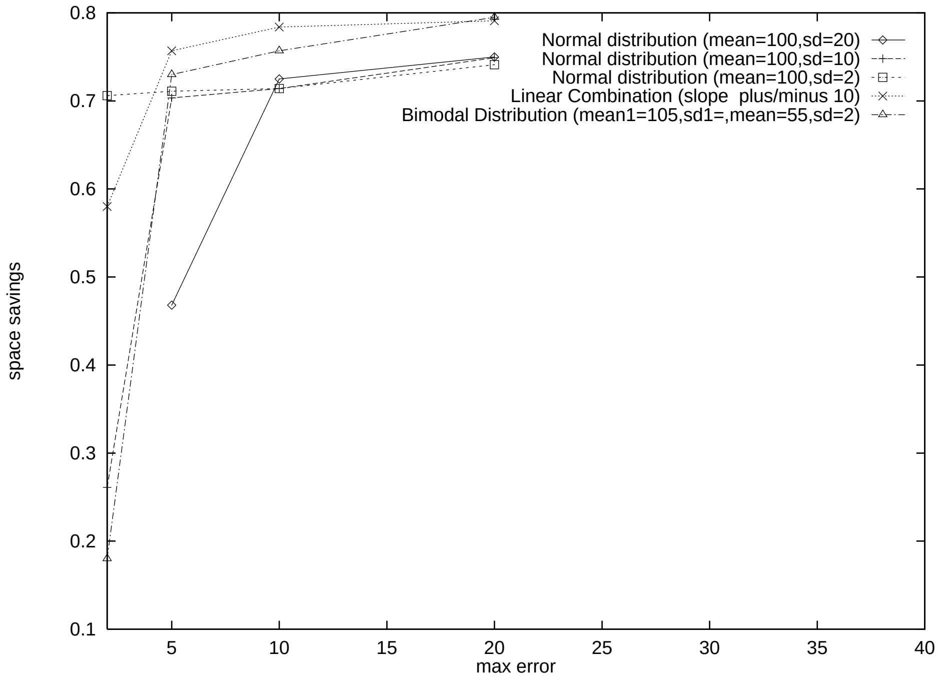

Figure 23: Space Savings (f_s) using SVD, for a cube with planes of dimension 1000×100. The X axis gives the maximum error. The Y axis is the space savings obtained by the Quasi-Cube as a fraction of the total space taken by the cube.

Dimensions	Time to build the Quasi-Cube	Time to build the conventional Cube
$597 \times 42 \times 966 \times 1764 \times 4 \times 69$	4,580 sec	74.84 sec
$1 \times 42 \times 966 \times 1764 \times 4 \times 69$	6.0 sec.	0.1 sec.

Table 2: Time to build Quasi-Cubes and cubes

and the metrics are updated accordingly. This process continues until all the regions have been modeled. Alternatively, to allow comparison with regular cubes, the system allows the building of entire cubes, i.e., computing aggregates to figure out every cell in each region but not modeling the region.

We used this software for building Quasi-Cubes for the RDATA dataset using regression as the modeling tool (with the error results reported in the previous section). Table 2 shows the performance of this system. The second column lists the time it takes to build a Quasi-Cube, while the third column list the time to build the whole cube (without modeling regions). The first row in the table corresponds to the experiment we ran using RDATA. For the second row, we took only a subset of RDATA, corresponding to the first value of attribute 1 and built a smaller Quasi-Cube (and cube) of the dimensions shown in the table. The overhead imposed by the regression method, although considerable (61:1) is manageable: a Quasi-Cube of the larger size ($597 \times 42 \times 966 \times 1764 \times 4 \times 69$) takes approximately 1.5 hours to be built in its entirety. We also see that the time to build Quasi-Cubes scales well with the size and dimensions of the dataset. The time to build the small Quasi-Cube is not exactly $\frac{1}{597}$ of the time needed to build the larger one, simply because the region we chose contains less than the average number of non-zero cells of regions in the larger cube (286 non-zero cells were present in the smaller cube, while the average per region in the larger cube is 561.68). The process of building Quasi-Cubes is CPU bound. Only 45.22 seconds were spent waiting for I/O when building the large Quasi-Cube. The system does not require anymore I/O than the necessary to bring the relation (in steps) to memory. As stated previously, we did not conduct the tests over RDATA using SVD.

We also conducted an experiment to compare the running time of the regression and SVD techniques. The results are shown in Table 3. The running times shown are to model planes of the size indicated in each column, once the whole matrix was in memory. (They were obtained by measuring the time needed to model 1,000 planes and dividing that time by 1,000.) The spareness of the matrix is shown in the second column of the table (e.g., a value of 90% means that ninety percent of the cells of the matrix were 0). Two things are made clear by this table. First, the running times of SVD are clearly several orders of magnitude greater than those of the regression technique. Secondly, unlike regression, where the running time decreases with the spareness of the matrix, SVD running time is independent of how sparse the matrix is.

3.2.4 Queries

We wanted to understand the behavior of queries on the proposed method. In order to do that, we indexed the retained values using a B-tree. As the key of the B-tree we used the concatenated coordinate values that define the position of the retained cell in the cube. (I.e., in a tridimensional cube, the cell i, j, k will have the concatenated key ijk.) Then we ran a series of queries and measure their response time. Each query's response time was measured for a number of different error thresholds (recall that the error threshold selected determines

Algorithm	size	spareness (%)	running time (msec.)
Regression	100×100	10%	1.9
	100×100	50%	1.0
	100×100	90%	0.5
	200×200	10%	130
	200×200	50%	50
	200×200	90%	40
	500×500	10%	370
	500×500	50%	350
	500×500	90%	240
	500×10	10%	7
	500×10	50%	5
	500×10	90%	1
	500×20	10%	9
	500×20	50%	6
	500×20	90%	2
SVD	100×100	10%	1,590
	100×100	50%	1,590
	100×100	90%	1,590
	200×200	10%	12,500
	200×200	50%	12,500
	200×200	90%	12,500
	500×500	10%	229,400
	500×500	50%	229,400
	500×500	90%	229,400
	500×10	10%	190
	500×10	50%	190
	500×10	90%	190
	500×20	10%	500
	500×20	50%	500
	500×20	90%	500

Table 3: Comparison of running times for building models using regression and SVD

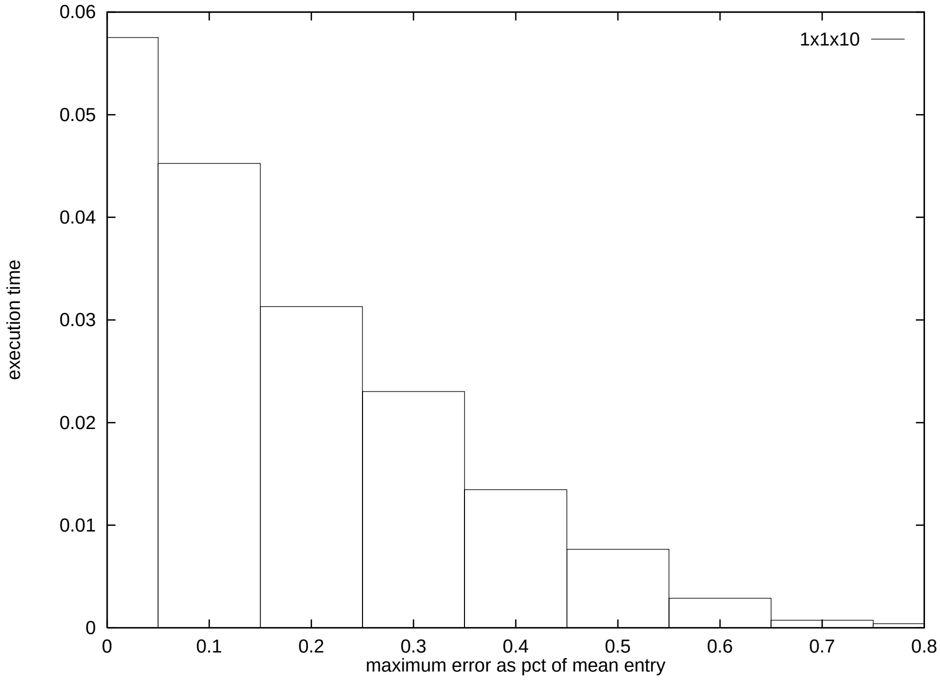

Figure 24: Response time for a query of size $1 \times 1 \times 10$ over a Quasi-Cube of dimensions $1000 \times 1000 \times 10$. The Y axis gives the response time in seconds (this value is the average of 100 queries). The X axis is the maximum error in the entry value as a percentage of the mean entry value.

the points that need to be retained). The results are shown in Figures 24, 25 and 26 which correspond to queries of sizes $1 \times 1 \times 10$, $1 \times 1000 \times 1$ and $500 \times 500 \times 5$ performed over a Quasi-Cube of size $1000 \times 1000 \times 10$. (For a given query size, we generated different queries by moving the cubettes defined by the query around the cube. We averaged the results of 100 different queries in each case.) In each case we can appreciate the tradeoff between errors and response time: tolerance for errors translates into drastically better performance. For instance, in the largest query ($500 \times 500 \times 5$), the difference between the perfect's answer performance (18 sec) and the performance obtained with 20 % of error is almost an order of magnitude. These measurements suggest that our technique is a very good way of supporting "Online Aggregation" [13], by providing quick, approximate answers to queries and refining the answers as the user looks at them. It would be enough to order the retained points according to the level of error they incur when estimated, and successively substitute the estimations by the real values (going to the disk). We plan to address this issue in future research.

4 Other Benefits of Modeling

Beside achieving space compression and helping support multiresolution queries and on-line aggregation, modeling parts of the cube gives a lot of useful information that helps to understand patterns in the data and to answer analytical questions.

For instance, the parameters in a regression model tell us to which dimensions the aggregates are more sensitive to. Consider, for instance a regression model built for sales data on

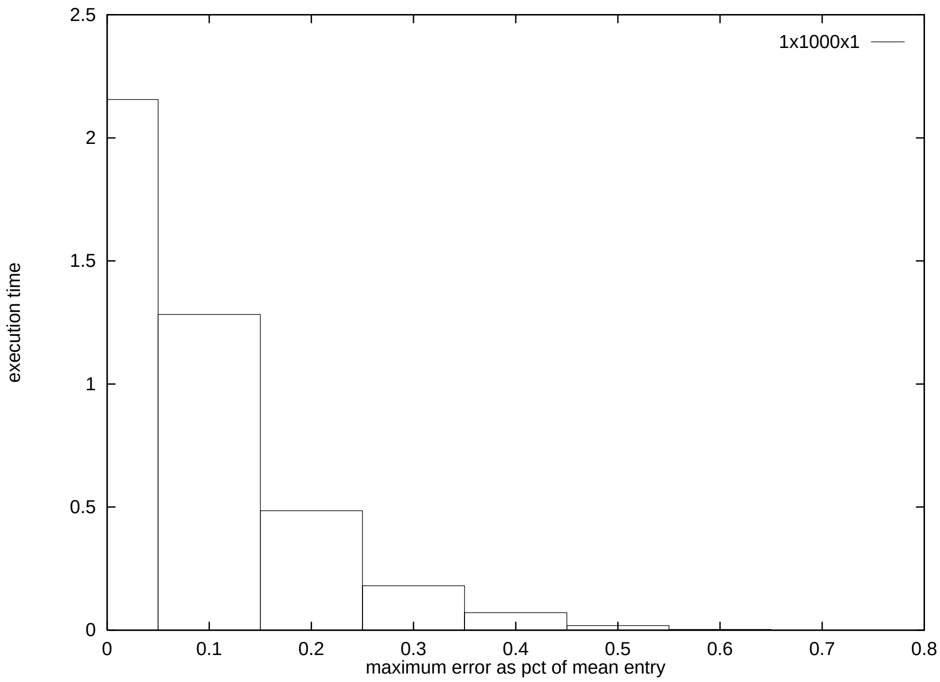

Figure 25: Response time for a query of size $1 \times 1000 \times 1$ over a Quasi-Cube of dimensions $1000 \times 1000 \times 10$. The Y axis gives the response time in seconds (this value is the average of 100 queries). The X axis is the maximum error in the entry value as a percentage of the mean entry value.

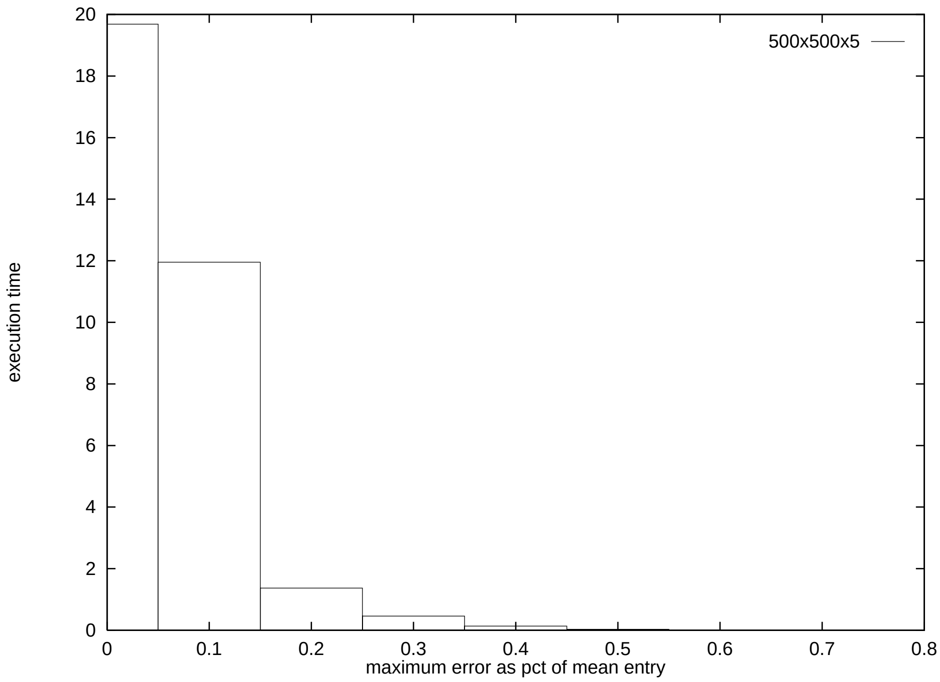

Figure 26: Response time for a query of size $500 \times 500 \times 5$ over a Quasi-Cube of dimensions $1000 \times 1000 \times 10$. The Y axis gives the response time in seconds (this value is the average of 100 queries). The X axis is the maximum error in the entry value as a percentage of the mean entry value.

three dimensions: date, average temperature on that day and dollars invested in publicity. Assume one of such models has been created for each region in the country. The resulting model has the form shown in Equation 16.

$$S_r = \alpha_r D + \beta_r T + \gamma_r P_r \quad (16)$$

where each parameter has been indexed using the region that corresponds to the model. A model that results in $\gamma_r >> \beta_r \; \gamma_r >> \alpha_r$ is indicating right away that publicity has a deeper effect in sales than any of the other factors involved. Moreover, comparing models for different regions, one can see how the corresponding parameters fare, perhaps making meaningful conclusions along the way. (E.g., publicity has a more direct effect in sales in the Northeast than in the Midwest.)

Regression models are not by any means the only ones that offer interesting information. Using SVD, for instance one may uncover hidden correlations among the different dimensions of the problem. Using the previous example, one could perform SVD in planes of date vs. publicity (keeping the rest of the dimensions fixed) and discover for which days publicity has the most significant effect. This can be achieved easily by considering the approximation shown in Equation 17, after selecting the k largest eigenvalues in Equation 5, substituting X by S (sales), U by D (date) and V by P (publicity).

$$S_k = D_k \times \bigwedge_k \times P_k \quad (17)$$

The rows of $P_k^T \times \bigwedge_k$ above represent the publicity described by the dates in which it has the most effect on sales.

Finally, the outliers are valuable pieces of information. They tell the analyst that this cells do not quite conform to the model that describes the rest of the cells well. An outlier can show, for instance, an exceptionally large sales value for a combination of dimension values that is worth looking into to understand what event brought the unexpected increase in sales.

Thus, modeling parts of the data cube can be also regarded as a first approach to mine information from the data warehouse.

5 Related Work

Multidimensional data bases are a central concept in the field commonly known as On-Line Analytical Processing (OLAP). OLAP systems are typically implemented either directly on top of relational systems or as a combination of a relational system and a proprietary multi-dimensional database (MDDB) [5]. In both approaches the underlying database is stored in a relational system. In a fully relational implementation (ROLAP), data cube queries are transparently converted into queries on the underlying relational database. Clever indexing schemes and careful query optimization are used to improve the performance of these queries. In the MDDB approach, the cube data is extracted from the database, converted into multi-dimensional arrays, and clustered so that common cube queries require minimal I/O. There are products in both the relational arena [24] and the MDDB world (Sinper's Spreadsheet Connector) that materialize only parts of the data cube. However, ours is the first approach that uses approximate cubes to save storage.

The Stanford University Data Warehousing Project studies methods to achieve efficient implementations of complete data cubes. A summary of research problems is presented in

[26]. Harinarayan, Rajaraman and Ullman [12] describe a near optimal algorithm to decide which parts of the cube should be materialized to obtain best performance and lowest space utilization.

The problem of completing partially specified tables given constraints on the sum by columns and rows has been studied in the domain of statistics and linear algebra [22]. S. Abad [1] uses three methods: mutual independence, maximum correlation and minimum correlation to combine statistical tables. The three estimates provide bounds for the answers. However, our experiments show that providing just the column and row sums does not give sufficient accuracy for moderate and large matrices. Also, we have observed that the errors incurred by using the mutual independence approach are bigger than those incurred by our regression method.

6 Conclusions

The data warehousing community has found data cubes to be a useful way to present summary descriptions of large databases to users. In this paper, we have shown a technique for presenting essentially the same information as a standard data cube but with significantly reduced storage cost. The Quasi-Cube structure uses a concise data representation consisting of a fraction of the standard cube entries and a set of model parameters. Queries of the Quasi-Cube estimate the missing entries with a reasonable level of accuracy using linear regression.

Our experiments have shown that Quasi-Cubes based on linear regression work better than ones based on the independence model or SVD. While the mean error is roughly the same for both methods, linear regression does a better job in keeping the maximum error down. We also should point out that the major disadvantage of the independence method is that the optimal selection of specified points is a difficult task. Our results hold over several different kinds of data distributions, including some that are badly behaved, such as the bimodal distribution. The results are also independent on the size of the matrix. Although SVD achieves better error rates than regression for dense cubes, the main problems with this method are its overhead, the fact that the method cannot take advantage of spareness and performs poorly with very sparse cubes and the need to always model the cubes plane by plane.

Altogether, we believe that our approach is a very practical way of reducing the space needed to store cubes and that Quasi-Cubes are a viable alternative to the methods used currently in practice to do OLAP. We have shown that our results are robust over a variety of data distributions and sizes of the cube, holding even for cubes with a large number of dimensions and points (10 dimensions and over 390 billion entries).

Although the building of a Quasi-Cube imposes an overhead in running time over the building of a traditional cube (mainly due to the need of computing regression parameters), we have seen that this overhead is manageable. It is also a fact that in building Quasi-Cubes instead of cubes, we are trading I/O time (in queries) and disk space for CPU time (in building the Quasi-Cube). This is a reasonable trade since CPU speeds tend to grow faster than I/O speeds. Moreover, the regression and estimation of errors is a highly parallelizable operation, opening the door for more speedups.

Sometimes, one of the points of doing OLAP is to find extraordinary numbers, e.g., very high sales of a product. For this applications, it would be convenient to retain the entries that satisfy the criterion of "extraordinary." This can be simply incorporated into our proposal by making sure that every point is tested for qualification. If a point qualifies, it is not discarded (but rather, retained), regardless of the error incurred by its estimate. By doing so, however, one pays the price of an increased number of retained points in the Quasi-Cube, therefore reducing the storage gains.

Even if the application cannot tolerate the smallest errors our method provides a systematic way of ordering cube cells according to the error they incur when estimated. Cells can be classified in bins each one corresponding to an error level. Cells belonging to the same bin can be cluster together in the disk. When a query is posed, a quick, approximate answer can be given by fetching the cells needed for the answer that correspond to the highest error level, while estimating the rest of the cells in the answer. This answer can be refined by bringing cells from the next level and substituting their estimated values by the real ones. Successive levels can be brought from the disk until either the user is satisfied with the answers accuracy or the answer contains no error. This process is similar to that presented in [13], with one exception: while their technique is based in sampling the fact table, our technique is based in pre-classifying the cells according to the model used for the Quasi-Cube construction. We plan to experiment with this technique in the future and evaluate its performance and merits.

Two additional uses of our technique are worth noticing. First, our methods can serve as the basis to implement Online aggregation [13], a method by which quick, approximate answers to queries are initially given to the user, while the system keeps refining them in an online fashion. We are currently implementing a system that uses Quasi-Cubes to support Online aggregation. Secondly, the models utilized to characterize parts of the cube offer a rich set of parameters that are very useful to the analyst, as we discussed in Section 4.

Furthermore, there are other modeling techniques that need to be studied in the context of compressing data cubes. Among these we want to study in the future wavelets [6, 18] and loglinear models [2] (specially suited for categorical data, i.e. data whose sale consists of a series of categories). A preliminary study of the characteristics of modeling techniques can be found in [3].

In practice, some data distributions may arise for which our methods will not perform as well as shown here. Perhaps Quasi-Cubes based on models more complex than linear regression can be used in those cases. However, even if Quasi-Cubes cannot replace normal data cubes in all instances, it is still a useful technique for data warehouse designers.

References

[1] S. Abad-Mota. Approximate Query Processing with Summary Tables in Statistical Databases. In *Proceedings of the 3rd Int'l Conference on Extending Database Technology, Vienna, Austria*, March 1992.

[2] A. Agresti. *An Introduction to Categorical Data Analysis.* John Wiley, New York, 1996.

[3] D. Barbará, 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.G. Sevcik. The New Jersey Data Reduction Report. *Bulletin of the Technial Committee on Data Engineering*, 20(4):3–45, December 1997.

[4] C.M. Chen and N. Roussopoulos. Adaptive Selectivity Estimation Using Query Feedback. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Minneapolis, Minnesota*, May 1994.

[5] G. Colliat. OLAP, Relational and Multidimensional Database Systems. *SIGMOD Record*, 25(3), September 1996.

[6] I. Daubechies. Orthonormal Bases of Compactly Supported Wavelets. *Communications on Pure and Applied Mathematics*, 41:909–996, 1988.

[7] R.O. Duda and P.E. Hart. *Pattern Classification and Scene Analysis.* Wiley, New York, NY, 1973.

[8] S. Dumais. Latent semantic indexing (LSI) and trec-2. In *Proceedings of the Second Text Retrieval Conference, Gaithersburg, Maryland*, March 1994.

[9] C. Dyreson. Information Retrieval from an Incomplete Data Cube. In *Proceedings of the 22nd International Conference on Very Large Data Bases, Bombay, India*, September 1996.

[10] E.L. Glaser, P. DesJardins, D. Caldwell, and E.D. Glaser. Bit string compressor with boolean operation processing capability. U.S. Patent # 5036457, July 1991.

[11] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. In *Proceedings of the International Conference on Data Engineering, New Orleans*, 1996.

[12] V. Harinarayan, A. Rajaraman, and J.D. Ullman. Implementing Data Cubes Efficiently. In *Proceedings of the ACM-SIGMOD Conference, Montreal, Canada*, 1996.

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

[14] T. Johnson and D. Shasha. Some Approaches to Index Design for Cube Forests. *Bulletin of the Technical Committee on Data Engineering*, March 1997.

[15] I.T. Jolliffe. *Principal Component Analysis.* Springer Verlag, 1986.

[16] R. Kimball. *The Data Warehouse Toolkit: How to Design Dimensional Data Warehouses.* John Wiley, New York, 1996.

[17] F. Korn, H.V. Jagadish, and C. Faloutsos. Efficiently Supporting Ad Hoc Queries in Large Datasets of Time. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Tucson, Arizona*, May 1997.

[18] S. Mallat. A Theory for Multiresolution Signal Decomposition: the Wavelet Representation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(7):674–693, July 1989.

[19] C.H. Papadimitriou, P. Raghavan, H. Tamaki, and S. Vempala. Latent Semantic Indexing: A Probabilistic Analysis. In *Proocedings of the ACM Conference on Principles of Database Systems (PODS), Seattle (to appear)*, 1998.

[20] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes in C, The Art of Scientific Computing.* Cambridge University Press, Cambridge, MA, 1996.

[21] S. Sarawagi. Indexing OLAP data. *Bulletin of the Technical Committee on Data Engineering*, March 1997.

[22] E. Seneta. *Non-negative matrices and Markov Chains*. Springer-Verlag, New York, 1980.

[23] D. Srivastava and K. Ross. Fast Computations of Sparse Cubes. In *PRoceedings of the 23rd International Conference on Very Large Data Bases, Athens, Greece*, August 1997.

[24] Inc. Stanford Technology Group. Designing the Data Warehouse on Relational Databases. White Paper.

[25] G. Strang. *Linear Algebra and its Applications*. Academic Press, 1980.

[26] J. Widom. Research problems in data warehousing. In *Proceedings of the 4th Int'l Conference on Information and Knowledge Management (CIKM)*, November 1995.

[27] R.J. Wonnacott and T.H. Wonnacott. *Introductory Statistics*. John Wiley, New York, 1985.

[28] P. Young. *Recursive estimation and time-series analysis*. Springer-Verlag, New York, 1984.