Storage Estimation for Multidimensional Aggregates in the Presence of Hierarchies*

Amit Shukla **Prasad M. Deshpande**
Jeffrey F. Naughton **Karthikeyan Ramasamy**

{*amit,pmd,naughton,karthik*}*@cs.wisc.edu*
Computer Sciences Department
University of Wisconsin - Madison

Abstract

To speed up multidimensional data analysis, database systems frequently precompute aggregates on some subsets of dimensions and their corresponding hierarchies. This improves query response time. However, the decision of what and how much to precompute is a difficult one. It is further complicated by the fact that precomputation in the presence of hierarchies can result in an unintuitively large increase in the amount of storage required by the database. Hence, it is interesting and useful to estimate the storage blowup that will result from a proposed set of precomputations without actually computing them. We propose three strategies for this problem: one based on sampling, one based on mathematical approximation, and one based on probabilistic counting. We investigate the accuracy of these algorithms in estimating the blowup for different data distributions and database schemas. The algorithm based upon probabilistic counting is particularly attractive, since it estimates the storage blowup to within provable error bounds while performing only a single scan of the data.

*Work supported by an IBM CAS Fellowship, NSF grant IRI-9157357, and a grant from IBM under the University Partnership Program.

Proceedings of the 22nd VLDB Conference
Mumbai (Bombay), India, 1996

1 Introduction

Multidimensional data analysis, as supported by OLAP systems, requires the computation of several large aggregate functions over large amounts of data. To meet the performance demands imposed by these applications, virtually all OLAP products resort to some degree of precomputation of these aggregates. The more that is precomputed, the faster queries can be answered; however, it is often difficult to say *a priori* how much storage a given amount of precomputation will require. This leaves the database administrator with a difficult problem: how does one predict the amount of storage a specified set of precomputations will require without actually performing the precomputation? In this paper we propose and evaluate a number of techniques for answering this question.

To further clarify the problem we are considering, we begin with an example[1]. Consider a table of sales with the schema

```
Sales(ProductId, StoreId, Quantity)
```

with the intuitive meaning that each tuple represents some quantity of some product sold in some store. Furthermore, assume that we have some information about products captured in a table

```
Products(ProductId, Type, Category)
```

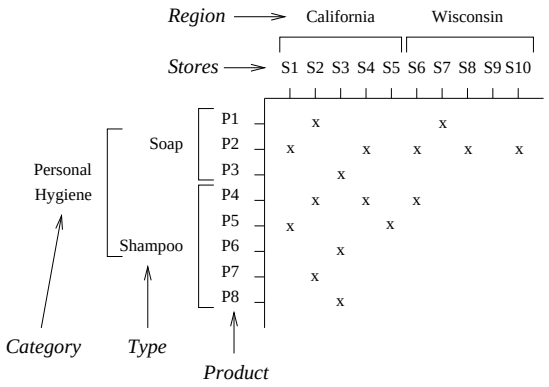
capturing for each product to which Type it belongs, and for each Type to which Category it belongs. Finally, suppose we have an additional table

```
Stores(StoreId, Region)
```

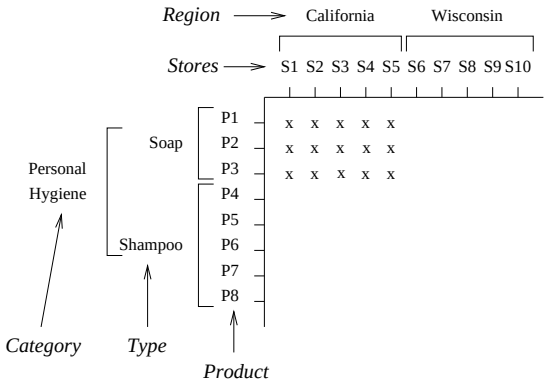
which captures for each store to which region it belongs.

This data set can be viewed conceptually as a two-dimensional array with hierarchies on the dimensions, as shown in Figure 1 (a).

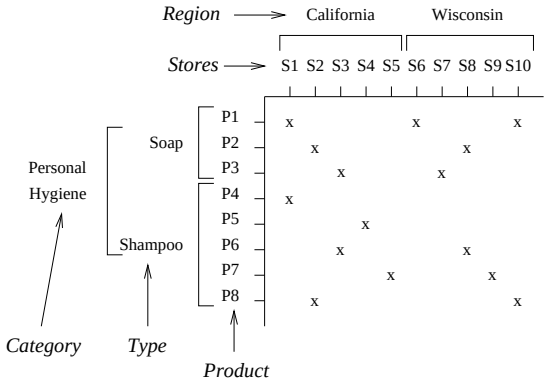
[1] This example first appeared in [AGS95]



(a)



(b) Database DB 1



(c) Database DB 2

Figure 1: Three sample multi-dimensional data sets. S_i's represent Stores and P_i's represent Products. Stores S1 - S5 are in California, and so roll up into the region California, while S6 - S10 are in Wisconsin, and roll up into the region Wisconsin. Products P1 - P3 are of type Soap, while products P4 - P8 are of type Shampoo. Soap and Shampoo are further grouped into the category Personal Hygiene. The **x**'s are sales volumes; entries that are blank correspond to (product, store) combinations for which there are no sales. (b) and (c) are sample multi-dimensional data sets which are used in an example.

There are a number of queries that can be asked of this data. For example, one may wish to know sales by product; or sales by type; or sales by product and region; or sales by store and type; and so forth. Each of these queries represents an aggregate computation. For example, sales by product in SQL is just:

```
select ProductId, SUM(Quantity)
from sales
group by ProductId;
```

If the sales table is large, this query will be slow. However, if this aggregate is precomputed, the query (and queries derived from it) can be answered almost instantly. Therefore, the task the DBA faces is to choose a set of queries to precompute and store. In this paper, we first consider the problem of estimating how much storage will be required if all possible combinations of dimensions and their hierarchies are precomputed. Furthermore, once we have described how to estimate this full precomputation the extension to precomputation of a subset is trivial.

A useful way to describe the full precomputation problem is to use the framework proposed by Gray et al. [GBLP96]: the *cube* operator. The cube operator is the n-dimensional generalization of the SQL group by operator. The cube on n attributes computes the group by aggregates for each possible subset of these dimensions. In our example, this is: (), (ProductId), (StoreId), (ProductId, StoreId). The SQL for these four group bys (in the above order) is:

```
select SUM(Quantity)
from sales;

select ProductId, SUM(Quantity)
from sales
group by ProductId;

select StoreId, SUM(Quantity)
from sales
group by StoreId;

select ProductId, StoreId, SUM(Quantity)
from sales
group by ProductId, StoreId;
```

When we consider the possibility of aggregating over hierarchies, we get a generalization of the cube, which we will refer to as the cube from here on. The cube as defined by [GBLP96], will be referred to as a cube without hierarchies. Again returning to our example, the cube with hierarchies will compute aggregates for (), (ProductId), (StoreId), (Type), (Category), (Region), (ProductId, StoreId), (ProductId, Region), (Type, StoreId), (Type, Region), (Category,

Table 1: The variation in the size of the cube with the data distribution. Figures 1 (b) and (c) show DB 1 and DB 2 respectively.

Group by	DB 1	DB 2
()	1	1
(Products)	3	8
(Type)	1	2
(Category)	1	1
(Stores)	5	10
(Regions)	1	2
(Products, Stores)	15	15
(Type, Stores)	5	15
(Category, Stores)	5	10
(Product, Region)	3	14
(Type, Region)	1	4
(Category, Region)	1	2
Size of Cube	42	84

StoreId), and finally (Category, Region). It is the presence of hierarchies in the dimensions that in general make the storage requirements of cubes with hierarchies far worse than that of cubes without hierarchies. Note that on this small example of only two dimensions the cube computed on the 16 tuples in Figure 1 (a) results in 73 tuples, while the cube without hierarchies has 34 tuples.

Furthermore, for a given database schema and a fixed number of data elements, the resulting size blowup on computing a cube can vary dramatically. Figure 1 (b) and (c) show two databases which illustrate the range of blowups that can occur. Each database has the same number of tuples (15), the same number of dimensions (2), and the same hierarchy on the dimensions. As the computation in Table 1 shows, even for a small database, and a small number of dimensions, the sizes of the cubes for the databases are very different.

Estimating the size of these blowups without computing the cube is the problem we are attacking in this paper. Computing the cube is a very expensive operation. For example, computing the cube for a schema of 5 dimensions each with two levels of hierarchies is equivalent to computing over 200 distinct SQL "group by" queries. One of the algorithms we propose, the one based on probabilistic counting, is especially attractive in that it estimates the cube size to within a provable error bound while only scanning the input data set once. The remainder of this paper is organized as follows. Section 2 discusses solutions to this problem. An evaluation of how well our algorithms work in practice is presented in section 3. Section 4 discusses extensions of the algorithm based on probabilistic counting. Finally, section 5 concludes and discusses future work.

2 Approximating the size of the Cube

This section presents three solutions that approximate the size of the cube.

2.1 An Analytical Algorithm

If the data is assumed to be uniformly distributed, we can mathematically approximate the number of tuples that will appear in the result of the cube computation using the following standard result. Feller [Fel57]:

> If r elements are chosen uniformly and at random from a set of n elements, the expected number of distinct elements obtained is $n - n(1 - 1/n)^r$.

This can be used to quickly find the upper bound on the size of the cube as follows.

To apply the uniform-assumption method, we need to know the number of distinct values for each attribute of the relation. Such statistics are typically maintained in the system catalog. Using the above result, we can estimate the size of a group by on any subset of attributes. For example, consider a relation R having attributes A, B, C, D. Suppose we want to estimate the size of the group by on attributes A and B. If the number of distinct values of A is n_1 and that of B is n_2, then the number of elements in $A \times B$ is $n_1 * n_2$. Thus $n = n_1 * n_2$ in the above formula. Let r be the number of tuples in the relation. Using these values we can estimate the size of the group by. This is similar to what is done in relational group by estimation.

A cube is a collection of group bys on different subsets of attributes. If we are computing a cube on k dimensions where dimension i has a hierarchy of size h_i then the total number of group bys to be computed is:

$$\prod_{i=1}^{k}(h_i + 1) \tag{1}$$

This figure is obtained by observing that in any group by at most one of the attributes in each hierarchy should be present. We can estimate the size of each of the group bys and add them up to give the estimated size of the cube.

Any skew in the data tends to reduce the size of the group bys reducing the size of the cube. Hence the uniform assumption tends to overestimate the size of the cube, and there is of course no way to know how far off it might be, since this method does not consult the database other than to gather crude cardinalities. It also requires counts of distinct values, without which it cannot be used. However, this method has the advantage that it is simple and fast.

2.2 A Sampling - Based Algorithm

In this section, we consider a simple sampling-based algorithm. The basic idea is as follows: take a random subset of the database, and compute the cube on that subset. Then scale up this estimate by the ratio of the data size to the sample size. To be more precise, we have the following. Let D and s be the database and a sample obtained from the database respectively. If $|s|$ is the sample size, $|D|$ the size of the database, and *CUBE(s)* is the size of the cube computed on the sample s, then the size of the cube on the entire database D is approximated by:

$$CUBE(s) * \frac{|D|}{|s|}$$

This is admittedly very crude. The approach of estimating the size of an operation by computing the operation on a subset of the data and then linearly scaling produces an unbiased estimator for some common relational algebraic operations such as join and select. Unfortunately, in this case, the estimate produced is biased, as estimating the size of the cube is more akin to estimating the size of a projection than it is to estimating the size of a join. However, once again the computation is simple, and has the potential advantage over the uniform assumption estimate of examining a statistical subset of the database (instead of just using cardinalities.) As we will see in Section 3, on many data sets, this simple biased estimator produces surprisingly good estimates.

2.3 An Algorithm Based on Probabilistic Counting

The key idea of the solution we propose in this section is based on an interesting observation made from Figure 1 (a). To compute the number of tuples formed by grouping Product type by Stores, we essentially group tuples along the Product dimension (to generate Product type), and count the number of distinct stores which are generated by this operation (See Figure 2). Hence, by estimating the number of distinct elements in a particular grouping of the data, we can estimate the number of tuples in that grouping. We use this idea to construct an algorithm that estimates the size of the cube based on the following probabilistic algorithm which counts the number of distinct elements in a multi-set.

2.3.1 The Probabilistic Counting Algorithm

Flajolet and Martin [FM85] propose a probabilistic algorithm that counts the number of distinct elements in a multi-set. It makes the estimate after a single pass through the database, and using only a fixed amount of additional storage. We present a description of their algorithm below.

For a non-negative integer y with L bits, $\mathrm{bit}(y,k)$ is defined to be the kth bit in the binary representation of y, such that $y = \sum_{k\geq 0} bit(y,k)2^k$. The function $\rho(y)$ represents the position of the least significant 1-bit in the binary representation of y.

$$\begin{array}{rcll} \rho(y) & = & \min_{k\geq 0}\ \mathrm{bit}(y,k) \neq 0 & \text{if } y > 0 \\ & = & \mathrm{L} & \text{if } y = 0 \end{array}$$

hash is a hashing function that transforms records into integers uniformly distributed over the set of binary strings of length L. That is, the range of *hash* is $0 \ldots 2^L - 1$. BITMAP$[0 \ldots L-1]$ is a bit vector. If M is the multi-set whose cardinality is sought, the basic algorithm comprises of the following sequence of operations:

for $i := 0$ **to** $L-1$ **do** BITMAP$[i] := 0$;
for all x **in** M **do**
begin
    index $:= \rho(hash(x))$;
    if BITMAP$[index] = 0$ **then**
        BITMAP$[index] := 1$;
end

If the values returned by hash(x) are uniformly distributed, the pattern 0^k1 appears with probability $2^{-(k+1)}$. The algorithm hinges on recording the occurrence of such patterns in the vector BITMAP$[0 \ldots L-1]$. Therefore, BITMAP[i] = 1 iff after execution, a pattern of the form 0^i1 has appeared among the hashed values of the data records. If n is the number of distinct elements, BITMAP[0] is accessed approximately $n/2$ times, BITMAP[1] approximately $n/4$ times, ... Thus, at the end of an execution, BITMAP[i] will almost certainly be *zero* if $i \gg \log_2 n$ and *one* if $i \ll \log_2 n$. The estimate formed from the above will typically within a factor of 2 from the actual size.

The simplest way to improve the accuracy of the estimate is to use a set $\mathbf{H}$ of m hashing functions, and computing m different BITMAP vectors. If R represents the position of the leftmost zero in the BITMAP, using m hashing functions we can obtain m estimates $R^{<1>}, R^{<2>}, \ldots, R^{<m>}$, where $R^{<i>}$ is obtained from hashing function i. We consider the average

$$A = \frac{R^{<1>} + R^{<2>} + \ldots + R^{<m>}}{m} \qquad (2)$$

When n distinct elements are present in a file, the random variable A has an expectation that satisfies

$$\mathbf{E}(A) \approx \log_2 \varphi n, \quad \varphi = 0.77351$$

Thus, 2^A can be expected to provide an estimate of n. The same effect can be achieved using *stochastic*

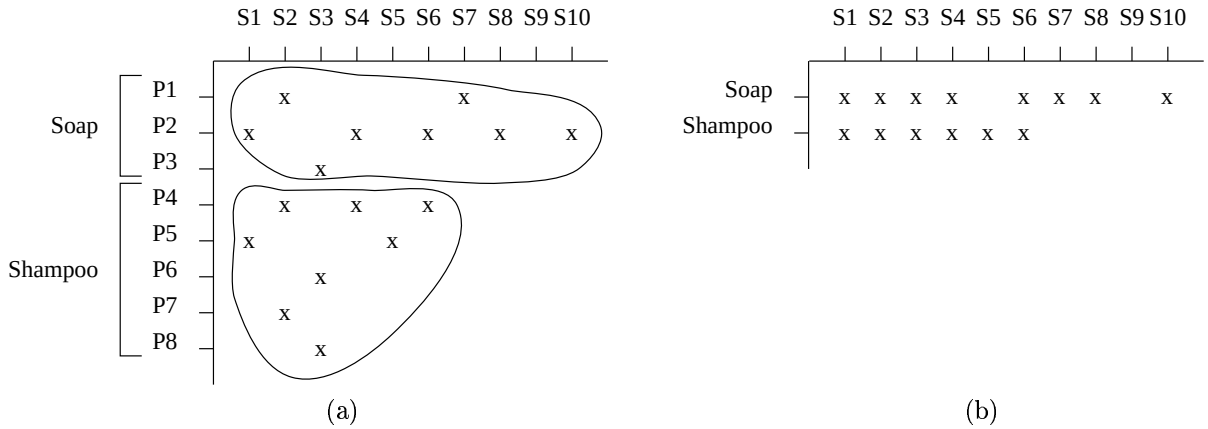


Figure 2: Grouping *Stores* by *Product Type*. (a) Before grouping, (b) after grouping.

Table 2: Parameters to the Probabilistic Counting Algorithm

Parameter	Value
Num. of Bitmaps	61
φ	0.77351
Len. of each bitmap	32

averaging. The idea is to use the hashing function to distribute each record into one of m lots, computing

$$\alpha = h(x) \ mod \ m \tag{3}$$

Only the corresponding BITMAP vector at address α is updated with the rest of the information contained in $h(x)$. At the end, we determine the $R^{<j>}$'s and compute their average A as before. Hoping for the distribution of records into lots to be even enough, we may thus expect that about n/m elements fall into each lot so that $(1/\varphi)2^A$ should be a reasonable estimate for n/m. Therefore, the estimate for n, the number of distinct values, is:

$$n = \frac{m}{\varphi}2^A \tag{4}$$

With the number of BITMAPs, $m = 64$, the standard error is about 10%, and with $m = 256$, the error decreases to about 5%. The parameters we used are summarized in Table 2.

2.3.2 Approximating the Size of the Cube

The following algorithm uses probabilistic counting to estimate the number of tuples resulting from computing the cube on the base data.

(0) Initialize the bitmaps to 0.
(1) **for** each tuple T in the database **do**
(2) **for** each combination C of hierarchies **do**
(3) $T' := P_C(T)$
(4) bitset(C, $\alpha(T')$, bit(T'))
(5) *count* := 0
(6) **for** each combination C of hierarchies **do**
(7) Add the estimate from C to *count*.

The function P_C takes a tuple and projects it on the combination of hierarchies C. $\alpha(T)$ is defined in equation 3, and *bit(T)* returns an integer representing the bit in the bitmap to be set. The bitmap update strategy is discussed in section 2.3.1. The function bitset(C, BM, b) sets the b^{th} bit in the BM^{th} bitmap for the combination of hierarchies C.

Example: To illustrate the working of the algorithm, consider a tuple (P1, S7, 10000) in the sample database shown in Figure 1 (a). This represents sales of 10,000 units of product P1 at store S7. From Figure 1 (a), product P1 is a Soap, and store S7 is in the region "Wisconsin". The combinations of hierarchies C, and the corresponding tuple generated are shown in Table 3. The bitmap associated with each of these combinations is updated in step (4) of the algorithm. □

The estimate of the number of distinct elements is given by equation 4. We now prove that if the bound on the error for a particular combination of dimensions is $\leq k$, then the error of the sum of two different combinations of hierarchies is also $\leq k$.

Lemma 1 *The error in the sum of two estimates is $\leq$ the error in a single estimate.*

Proof. Suppose that the two estimates have errors $\leq k$. Suppose estimate 1 and estimate 2 respectively have errors r_1 and r_2, they predict that the number of distinct values are E_1 and E_2, and the actual number of distinct

Table 3: The combinations of hierarchies for a tuple. Since `Quantity` is the data being aggregated upon, it is always projected out.

Group by	Projected tuple
()	()
(Products)	(P1)
(Type)	(Soap)
(Category)	(Personal Hygiene)
(Stores)	(S7)
(Regions)	(Wisconsin)
(Products, Stores)	(P1, S7)
(Type, Stores)	(Soap, S7)
(Category, Stores)	(Personal Hygiene, S7)
(Product, Region)	(P1, Wisconsin)
(Type, Region)	(Soap, Wisconsin)
(Category, Region)	(Personal Hygiene, Wisconsin)

elements are respectively N_1 and N_2.

$$r_1 = \frac{E_1 - N_1}{N_1}; r_2 = \frac{E_2 - N_2}{N_2} \tag{5}$$

The error in the combination of the two estimates is:

$$\frac{(E_1 + E_2) - (N_1 + N_2)}{N_1 + N_2}$$

This can be rewritten as:

$$\frac{(E_1 - N_1) + (E_2 - N_2)}{N_1 + N_2}$$

From Equation 5,

$$\frac{N_1 r_1 + N_2 r_2}{N_1 + N_2} \tag{6}$$

Since $r_1 \leq k; r_2 \leq k$, Equation 6 is:

$$\leq \frac{k(N_1 + N_2)}{N_1 + N_2}$$

which is $\leq k$. Hence we have proved that the error in the sum is bounded by the same constant k as the errors in the component estimates. □

Note that this algorithm, unlike the uniform estimate blowup and the simple sampling-based estimate, actually guarantees an error bound on its estimate. This comes at the cost of a complete scan of the base data table; however, even this scan is much cheaper than actually computing the cube, which in general requires multiple scans and sorts of the input table [AAD+96].

Table 4: The number of distinct elements in each of the dimensions. The total number of tuples in the base data = 60,000 [Schema 1]

Dimension num.	Dimension	Hierarchy	
		1	2
0	1000	200	50
1	10,000	500	–

Table 5: The number of distinct elements in each of the dimensions. The total number of tuples in the base data = 50,000 [Schema 2]

Dimension num.	Dimension	Hierarchy	
		1	2
0	1000	20	–
1	100	4	–
2	2000	50	–
3	10,000	500	10
4	750	250	25

3 Evaluating the Accuracy of the Estimates

In this section we compare the accuracy of the three approaches by comparing their estimates of the size of the cube with its actual size. Tables 4 and 5 contain the schemas and the number of distinct values of the dimensions and hierarchies of the two databases we used. For example, the data in Table 4 means that the database has two dimensions. Dimension 0 has a two level hierarchy, and dimension 1 has a one level hierarchy. Dimension 0 has 1000 distinct values, and its hierarchies have 200 and 50 values respectively, while dimension 1 has 10,000 distinct values, and its hierarchy has 500 values. The database is a combination of distinct values of all dimensions. A Zipfian distribution [Zipf49] was used to generate the database from the distinct values of each dimension. A Zipf value of 0 means that the data is uniformly distributed. By increasing Zipf, we increase the skew in the distribution of distinct values in the database. The mapping from the distinct values in a dimension to its hierarchies uses a uniform distribution.

Figure 3 shows for varying degrees of skew, the actual size of the cube, and estimates made by the probabilistic counting algorithm, by an analytical estimate using an uniform approximation, and by the sampling algorithm with three sample sizes (100, 500 and 1000 samples). Figure 4 provides a different perspective of

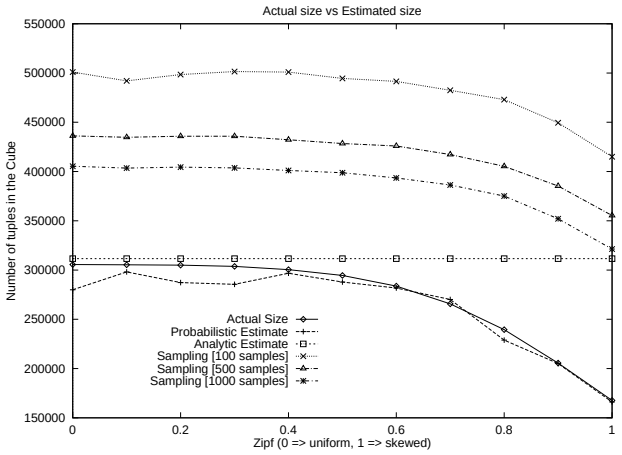


Figure 3: Estimates vs. the actual size of the cube for Schema 1.

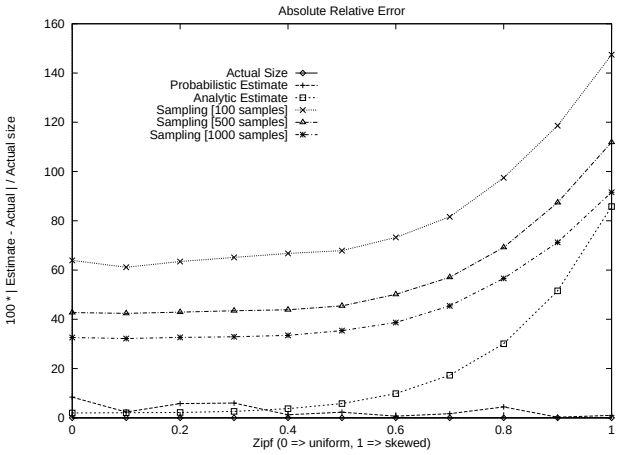


Figure 4: The error in the estimates from the actual size of the cube for Schema 1.

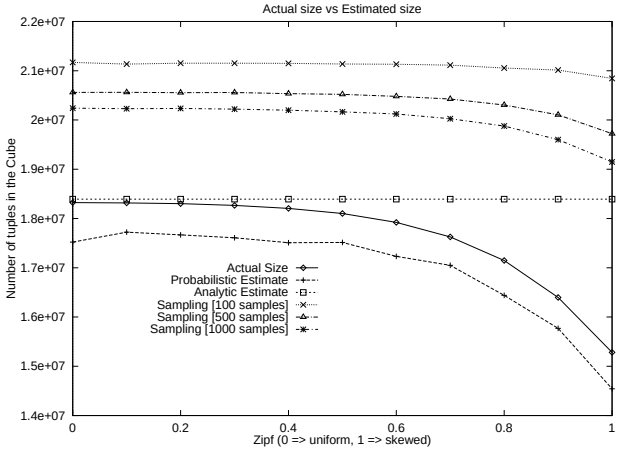


Figure 5: Estimates vs. the actual size of the cube for Schema 2.

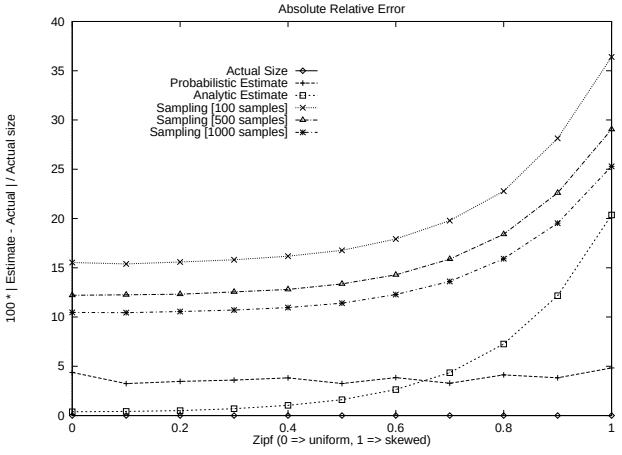


Figure 6: The error in the estimates from the actual size of the cube for Schema 2.

the data. For each degree of skew, we scale the actual size of the cube to 100, and then scale the other values relative to the actual cube size. Figures 5 and 6 present the same information for database schema 2.

The analytical algorithm based on the assumption that the data is uniformly distributed, provides an estimate that is very close to the actual size when the data is indeed uniformly distributed. However, when the skew in the data distribution increases, the size of the cube decreases. Since the estimate by the analytical algorithm is independent of the underlying data distribution, its prediction becomes more and more inaccurate. Hence, it tends to over-estimate the size of the cube.

The algorithm based on sampling picks tuples randomly from the database. It over-estimates the size of each group since it doesn't see enough duplicates. However, we expected it to do much worse, since applying the same algorithm to a single dimensional cube with a single level hierarchy is just estimating the size of a projection. This sampling algorithm is known to perform very poorly in general in that case. The reason the algorithm fails for projection estimation in general is that it is highly unlikely to see enough duplicates for an accurate estimate. Suppose, for example, we have a table of 1,000,000 tuples, with 500,000 distinct values. Then any reasonable sample size will be unlikely to see any duplicates, hence it will generate an estimate closer to 1,000,000 distinct values rather than 500,000. However, if the table in question in fact consists of all distinct values, the simple blowup sampling estimator we are using estimates the size perfectly! Now back to the cube, most of what we are estimating is for combinations of two or three attributes. Even if each attribute itself contains a large number of duplicates, these higher dimensional combinations contain very few duplicates. For these, the algorithm is close to correct, hence the overall estimate is not bad. To verify this, we carried out another set of experiments on a database with two dimensions, $D0$ and $D1$. Each dimension had 100 unique values, and the database consisted of 50,000 tuples. There was no hierarchy on either dimension. Let us call this schema 3. The number of distinct values was small, resulting in a lot of duplicates in the database. Now, the sampling based approach over-estimates the size of the cube by orders of magnitude (see Figures 7 and 8). Hence, we can conclude that the sampling based approach was performing well on the data sets associated with schemas 1 and 2 because the number of duplicates was too small.

The algorithm based on probabilistic counting estimates the size of the cube to within a theoretically predicted bound. The values of the parameters we used are shown in Table 2. The estimate is accurate under widely varying data distributions, ranging from uniform to highly skewed. It scans the database only once. It maintains storage proportional to the number of group bys that will be performed in order to compute the cube. The number of group bys is given by Equation 1. Therefore, using memory proportional to $\prod_{i=1}^{k}(h_i+1)$, and a single scan of the database we can accurately estimate the size of precomputed aggregates.

4 Extensions to the PCSA-based algorithm

In this section we look at how to estimate the size of a sub-cube. Estimation of the cube size after the addition or deletion of data is also discussed.

4.1 Estimating sub-cube sizes

The PCSA based algorithm in section 2.3.2 considers all combinations of hierarchies in order to generate an estimate of the cube. If the size of a sub-cube is desired, we can generate those combinations of hierarchies which make up the desired sub-cube. For example, in the cube computation presented in Table 1, if we compute the sub-cube which includes "Products" as one of the group by attributes, the following set S of group bys will be computed:

$$S = \{(\text{Products}), (\text{Products, Stores})\}$$

and lines (2) and (6) of the algorithm now read:

(2) **for** each $C \in S$ **do**

(6) **for** each $C \in S$ **do**

Since we estimate the size of each group by in order to estimate the cube size, we can trivially estimate the size of a single group by.

4.2 Incremental estimation

Data warehouses typically batch updates to the database. For example, loading weekly sales data into the warehouse can be done once a week. The addition of new data may changes the sizes of some of the group bys, and hence change the size of the new cube. This change in group by sizes can be estimated by updating the bitmaps used by the previous estimation. The estimate of the size of a group by is made from the corresponding bitmap array. So, the changes in the size of a group by can be captured by storing the bitmap array corresponding to a group by and updating it using the new data. To estimate the cube size, the bitmaps corresponding to every combination of group bys have to be stored. All additions of data to the database must also update these bitmaps.

The changes to the algorithm in section 2.3.2 are minimal. Step (0) becomes:

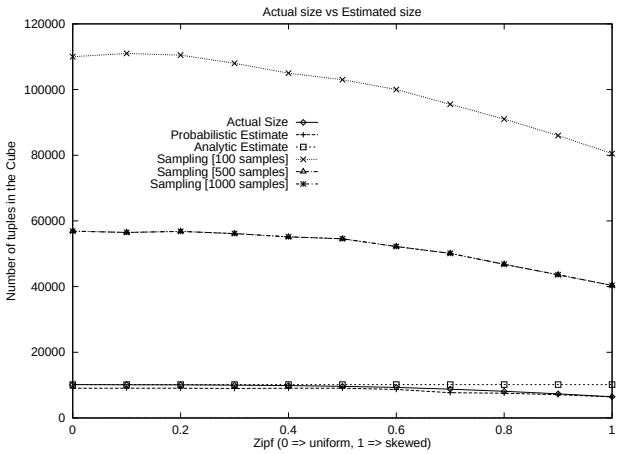


Figure 7: Estimates vs. the actual size of the cube for Schema 3.

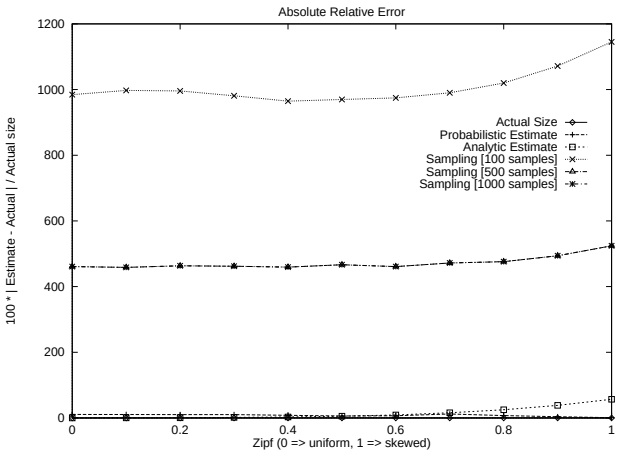


Figure 8: The error in the estimates from the actual size of the cube for Schema 3.

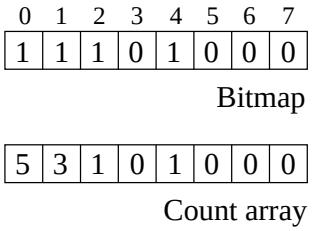


Figure 9: A bitmap array and its corresponding count array. Bits 0, 1, 2 and 4 of the bitmap are 1s. From the count array, the number of "hits" to these bits are 5, 3, 1 and 1 respectively.

(0) Load the bitmaps from disk.

If $|C|$ is the number of group bys in the cube, L is the length of each bitmap and m is the number of bitmaps per group by, the storage needed for the bitmaps is: $|C| * L * m$.

4.3 Estimation after data removal

Business may want to keep data that is fairly recent in its database (older data can be moved from to tertiary storage). For example, a business will want to keep sales data from the last 65 weeks (5 quarters) in its database. Usually blocks of data are discarded at the same time.

It is not sufficient to maintain bitmaps to estimate cube sizes with removal of data. For each bitmap, we have to store the number of "hits" for each bit (see Figure 9). Corresponding to a bitmap, we have an array of integers, the n^{th} element of which is the number of times tuples tried to set the n^{th} bit of the bitmap to 1. The algorithm is:

(0) Load the counter arrays from disk.
(1) **for** each tuple T being deleted **do**
(2) **for** each combination C of hierarchies **do**
(3) $T' := P_C(T)$.
(4) Decrement(C, $\alpha(T')$, bit(T'))

The function *Decrement* takes three arguments as input, the combination of hierarchies, the number of the counter array, and the index into the counter array respectively, and decrements the specified element of the counter array. α is defined in equation 3. The estimate can be formed using the count-arrays.

If $|C|$ is the number of group bys in the cube, L is the length of each count-array (equal to the length of a bitmap), m is the number of count-arrays per group by (equal to the number of bitmaps per group by), and the size of an integer is I, the storage needed for the count-arrays is: $|C| * L * m * I$.

5 Conclusions

Precomputing aggregates on some subsets of dimensions and their corresponding hierarchies can substantially reduce the response time to a query. However, precomputation in the presence of hierarchies results in a large increase in the amount of storage required to store the database. In this paper, we presented three strategies to estimate this blowup.

Comparing the algorithms based on their accuracy, we find that the algorithm based on sampling overestimates the size of the cube, and the estimate is strongly dependent on the number of duplicates present in the database. The algorithm based on assuming the data is uniformly distributed works very well if the data is uniformly distributed, but as the skew in the data increases, the estimate (which is independent of the skew) becomes inaccurate. In the experiments we carried out, the analytical estimate was more accurate than the sampling based estimate for widely varying

skew in the data. The algorithm based on probabilistic counting performs very well under various degrees of skew, always giving an estimate with a bounded error. Hence it provides a more reliable, accurate and predictable estimate than the other algorithms.

Analyzing the amount of work performed by the different algorithms we can see that the analytical approximation does not look at the data, and hence the amount of work done is dependent only on the schema, and not on the data. The algorithm based on sampling needs to see only a small subset of the database. Sampling may be relatively expensive depending on the page access pattern of the sampling strategy. Each tuple may need a page access, making the algorithm expensive. The algorithm based on probabilistic counting scans the entire database once and performs work proportional to the number of group bys for each tuple.

Which algorithm is best depends upon the desired accuracy, the amount of time available for the estimation, and the degree of skew in the underlying data. But in most cases, the algorithm of choice for a reasonably quick and accurate estimate of the size of the cube is the algorithm based on probabilistic counting.

References

[AAD+96] S. Agarwal, R. Agrawal, P.M. Deshpande, A. Gupta, J.F. Naughton, R. Ramakrishnan, S. Sarawagi. On the Computation of Multidimensional Aggregates. *Proc. of the 22nd Int. VLDB Conf.*, 1996.

[AGS95] R. Agrawal, A. Gupta, S. Sarawagi. Modeling Multidimensional Databases. *IBM Research Report*, IBM Almaden Research Center, San Jose, California, 1995.

[CCS93] E.F. Codd, S.B. Codd, C.T. Salley. Providing OLAP (On-Line Analytical Processing) to User-Analysts: An IT Mandate, E.F. Codd and Associates, 1993. *Available from http://www.arborsoft.com/papers/intro.html.*

[Fel57] W. Feller. *An Introduction to Probability Theory and Its Applications*, Vol. I, John Wiley and Sons, pp 241, 1957.

[FM85] P. Flajolet, G.N. Martin. Probabilistic Counting Algorithms for Database Applications, *Journal of Computer and System Sciences*, 31(2): 182-209, 1985.

[GBLP96] J. Gray, A. Bosworth, A. Layman, H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals, *Proc. of the 12th Int. Conf. on Data Engg.*, pp 152-159, 1996.

[HNSS93] P.J. Haas, J.F. Naughton, S. Seshadri, A.N. Swami. Selectivity and Cost Estimation for Joins Based on Random Sampling. *IBM Research Report RJ9577*, IBM Almaden Research Center, San Jose, California, 1993.

[HNSS95] P.J. Haas, J.F. Naughton, S. Seshadri, L. Stokes. Sampling-Based Estimation of the Number of Distinct Values of an Attribute, *Proc. of the 21st Int. VLDB Conf.*, 311–322, 1995.

[HRU96] V. Harinarayanan, A. Rajaraman, J.D. Ullman. Implementing Data Cubes Efficiently, *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 205–227, 1996.

[KT95] Kenan Technologies. An Introduction to Multidimensional Database Technology, *Available from http://www.kenan.com/*

[Str95] MicroStrategy Inc. The Case for Relational OLAP, *A white paper available from http://www.strategy.com/*

[Zipf49] G.K. Zipf. *Human Behavior and the Principle of Least Effort*, Addison-Wesley, Reading, MA, 1949.