Quasi-Cubes: Exploiting approximations in multidimensional databases

Daniel Barbará
Bell Communications Research
445 South St.
Morristown, N.J. 07960
dbarbara@bellcore.com

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

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 describe 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. 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.

Alternatively, the technique can be used to implement a multiresolution system in which a tradeoff is established between the execution time of queries and the errors the user is willing to tolerate. By only going to the disk when it is necessary (to reduce the errors), the query can be executed faster. This idea can be extended to produce a system that incrementally increases the accuracy of the answer while the user is looking at it, supporting on-line aggregation.

1 Introduction

A *data cube* is a popular organization for summary data [3]. 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 of [2]). 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.

A data cube can be implemented using an *eager* or a *lazy* materialization strategy [6]. 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 a big number of aggregate values. Even if the data is sparse and only 20% of the combinations have non-zero entries, the number of aggregate values is 0.73 billion. 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.

We have been experimenting with a strategy 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. There is a tradeoff between the amount of storage used for the description and the accuracy of the estimation. We call these approximated cubes *Quasi-Cubes.*

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.

Quasi-Cubes techniques can also be used for purposes other than saving space. For instance, one can organize the data to provide a *multiresolution* system in which a tradeoff is offered between errors and time to run a query. A user can speed up queries by accepting some level of inaccuracy in the answer. Also, this multiresolution system can be used to give users on-line answers whose accuracy improves as the system keeps working on the query, as has been described in [5].

The paper is organized as follows. In Section 2 we briefly describe the ideas behind incompletely specified cubes, or *Quasi-Cubes* and how they can be used to save storage. In Section 3, we talk about using these ideas to implement multiresolution systems. Section 4 offers some conclusions.

2 Quasi-Cubes

In this section, we explain the basics of Quasi-Cubes. The idea is to divide the cube in regions and use a statistical model to describe each region. The model can be used to estimate the entries in the region, but since the data does not fit the model perfectly, errors are introduced in the process. The entries that incur the highest errors are stored rather than modeled in an effort to increase the accuracy of the method. The more points that one is willing to retain, the better the accuracy of the method. Therefore, there is a tradeoff between the space devoted to the specification of the Quasi-Cube and the errors obtained.

The approach that has given us the best results is to view the cube as a set of two-dimensional planes. For each plane or matrix, we convert the entries into probabilities by dividing each one by the sum of all entries in the cube. After that, we compute the row marginals and store them. Then, for each column, we apply linear regression to model the entries on the column. In other words, we will use a equation of the form:

$$p_{ij} = b_0 r_i + b_1 \qquad (1)$$

to model column j of the matrix. 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 [7]

Notice that carefully selecting the granule of the cube in which one performs the modelling has a definite impact on how efficiently Quasi-Cubes can be generated. For instance, by slicing the cube into planes it is easy to generate all the entries in the plane by aggregating the corresponding tuples in the base relation and keep the plane around while the row marginals and

the column regressions are computed. (The plane is bounded to fit in main memory.) After this is done, the space dedicated to the plane can be reused to perform the computations in the next plane. In this way, the space overhead is kept very low. Moreover, if one has the appropriate support from the system (indexes over the base relation), it is possible to do all this and retrieve each tuple in the base relation just once. (So, one pass over the entire database suffices to compute the Quasi-Cube.)

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 1. In order to know whether to add or to subtract this value, we keep a bit vector of signs. We call this vector *sign*. The corresponding sign of row i and column j is denoted by $sign(i, j)$.

Estimating entries obviously produces errors. In order to keep the errors low, we want to save also the real entries that would incur in the larger errors if they were estimated. In other words, we select the outliers of the model as points to be retained and stored. By not having to estimate these points, we decrease the errors incurred by the process. Retaining these points can be achieved in one of two ways:

- Predetermine how many entries k are to be retained (stored) by deciding on a fraction of the total entries on the cube. Select the k points with the largest errors and store them.
- Select an error cutoff point and store entries whose errors are bigger than the cutoff point.

Again, notice that none of the two methods requires a large amount of space overhead. In the first method, only k points need to be retained at any given point in time. (If an entry exhibits an error less than the minimum error in the current set of k points, then it is simply discarded; otherwise it displaces the entry with the minimum error so far.) In the latter method, if an entry has an error less than the cutoff point, it is discarded. Otherwise, the point is retained. The number of points retained is then a fraction of the total number of entries in the cube, proportional to the level of error selected.

Since cubes are often sparse, we need a way to perform the procedure described above using only the positive entries, since the space savings must be achieved over the fraction of the cube that contain non-zero entries. (Commercial products that materialize the cube use specialized data structures that retain only the non-zero entries of the cube, thereby saving the space to store the zeros.) This can be easily achieved by adding a bitmap description of the cube that contains 0's for the zeros and 1's for the positive entries. This bitmap can be heavily compressed, and therefore will not take much space. In any given column, the linear regression would be performed over the non-zero entries 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.

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 retained entries.

- A bitmap indicating which entries of the cube are zeroes.
- The total value of the aggregates *total* (to reconstruct the aggregates matrix).

Our experiments have shown that the technique can produce very acceptable levels of errors by retaining a small fraction of the non-zero entries. This results in high storage savings with respect to the alternative of materializing the entire cube.

Also, by virtue of the implementation and the way we slice cubes, the overhead incurred to compute the Quasi-Cube is small when compared to computing the entire cube. (Of course, computing the Quasi-Cube requires more work than simply aggregating tuples to compute the cube.) Our preliminary results show that the extra time taken by the system to compute the Quasi-Cube is very reasonable. This can be attributed to the fact that we only use each tuple in the base relation once and also to the fact that the CPU time required to perform regressions is small. (Roughly 0.2 microseconds per entry in a SPARC 10.)

Another important point to address is the need to update the Quasi-Cubes as new tuples are added to the warehouse. New tuples cause one of two events: either an entry needs to be updated or a new entry in the cube needs to be added (one or more of the dimensions grow in the number of domain values). In both cases, the new point can be computed and tested using the corresponding model (e.g., the parameters of the column regression can be used to estimate the value and compare the result with the new value). If the error incurred is bigger than the level of error that one wants to support the new point needs to be retained (if it was not retained before). Otherwise, the point 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 points. So, an eventual recomputation of the affected models is advisable.

3 Multiresolution

The technique used to build Quasi-Cubes can be also seen as a method to classify the data in the cube. By computing the errors produced by the estimation procedure, it is easy to classify each entry into an "error bin," where a bin is defined by two error limits (i.e., more than 10% but less than or equal to 20 %). We explain here how this classification can be exploited to implement a multiresolution system using Relational OLAP (ROLAP) techniques.

A common implementation of ROLAP uses the star schema to support queries (see [1] for a detailed survey). In this schema, a *fact* table is the central piece that contains the data on which the aggregations are going to be based. For instance, a fact table could contain a row for each order by any product, any customer and any sales agent, indicating also the amount of the sales. The schema also contains *dimension* tables whose keys are foreign keys of the fact relation. The dimension tables are much smaller than the fact table. In our example there would be three dimension relations: one for product, one for customer and one for sale agents. Each one of these tables would contain information relevant to products, customers and sale agents respectively. A point in the data cube is formed by aggregating values of sales for a given combination of product, customer and sales agent. An efficient way of supporting queries is providing bitmap indexes that associate each value of the domain of the keys in the dimension tables with the rows in the fact table [4]. For instance, the product 'VCR' in the product relation would have a bitmap index that contains 1's in the positions of rows in the fact table that correspond to sales of VCRs. By intersecting bitmaps, the system can

efficiently find the row numbers of the tuples that need to be retrieved and aggregated to answer a query.

Multiresolution can be easily added to such a system in the following way. For a given level of error that the user is willing to tolerate, what is needed is an index that decides whether the entry in the cube needs to be materialized from the raw data (fact relation) or it can be estimated by the model. So, for each combination of key attributes in the dimension tables, a bit in this index will tell the system to proceed by materializing the data from the fact relation (using the dimension bitmaps to find which tuples need to be aggregated) or simply estimate the point. Therefore, the index is a bitmap mapping the cube to a decision of fetching data from the disk or estimating it. This bitmap (as the dimension bitmaps), can be heavily compressed using standard techniques. We use one bitmap per error level, so users interested in different error levels can be supported. (Hence, the multiresolution nature of the system.) The building of the bitmaps is simple: if the estimation of an entry is bound to cause an error bigger than the maximum error supported by the bitmap index, then the bitmap contains a 1 in the position corresponding to the entry. Otherwise, it contains a 0.

This scheme can be easily adapted to answer queries incrementally, presenting the user with a quick answer that gets subsequently polished as the user is looking at it. (In the fashion explained in [5].) By simply starting with the highest error level, and letting the system retrieve from disk the rows corresponding to entries that would cause the most errors (and estimating the rest), the user gets a first, quick, approximate answer. The answer is polished by going to the next level of error and retrieving more tuples (thereby replacing some of the previous estimates by real values) and increasing the accuracy. The process ends when the answer is 100 % accurate or when the user is satisfied and stops the query execution.

4 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. Further research is needed to experiment with other modeling techniques that may bring greater space savings. Since the issue of the model used is orthogonal to the method, it is easy to "plug in" other estimation methods in the system.

The multiresolution approach is a very efficient way to decrease the execution time for OLAP queries. Combining the approach with a method to increase the accuracy of the answers on-line is a very attractive proposition for OLAP analysts.

Finally, although we have not mention it in this paper so far, the techniques described here have a lot of potential for performing mining in the warehouse. For instance, the information about outliers (retained entries of the cube) is in itself valuable: these are points that exhibit a behavior that distinguishes them from the other entries and therefore are likely to be of interest to analysts. (For instance, an outlier could be a point that signals that some store sold a lot of certain type of items during a period of the year.) The search for outliers can be generalized by selecting other cube granules to be modeled. (For instance, one could model entire planes and find planes that are outliers with respect to the other planes.) Also, the parameters of the models are in themselves valuable information for

the analysts, since they explain how the "normal" entries behave, thereby uncovering patterns in the warehouse.

We are currently conducting experiments in each one of this areas and building prototypes to prove the practicality of the concepts presented here.

References

[1] S. Chaudhuri and U. Dayal. Decision Support, Data Warehousing and OLAP. VLDB tutorial, Bombay, India, 1996.

[2] 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.

[3] 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.

[4] H.Edelstein. Faster data warehouses. TechWeb, http://techweb.cmp.com/iwk/., December 1995.

[5] J.M. Hellerstein, P.J. Hass, and H.J. Wang. Online Aggregation. In *Proceedings of the ACM-SIGMOD Conference, Tucson*, 1997.

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

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