The Role of Approximations in Maintaining and Using Aggregate Views

Daniel Barbará and Xintao Wu
George Mason University *
Information and Software Engineering Department
Fairfax, VA 22030 Email: {dbarbara,xwu}@gmu.com

Abstract

Techniques for approximating the data cube have been the object of much research lately. In this paper, we present a summary of Quasi-Cubes, a technique that allows to compress data cubes while keeping guaranteed bounds for the errors made when querying the approximate versions, regardless of the distribution of the underlying data. We also present a suite of applications that can benefit from using Quasi-Cubes, among them some in the area of data mining.

1 Introduction

Data cubes [9] are a widely used data abstraction for aggregates of multidimensional data. A cube is simply a multidimensional structure that contains in each cell an aggregate value, i.e., the result of applying an aggregate function to an underlying relation that consists of a series of attributes known as dimensions and one or more attributes known as measures. For instance, for a retail sales dataset with dimensions *day*, *store* and *product* and measure *sales*, each cell of the data cube would contain the total sales for a combination of the values of the dimensions. At the finest level of aggregation (known as the core cuboid), cells would contain sales for specific days, stores and product values, e.g., $January\ 1st$, $NewYork$, and *shoes*. At other levels of aggregation cells would contain combinations of proper subsets of the dimensions, e.g., total sales of *shoes* in the $NewYork$ store (for all the "days"). The different levels of aggregation, or cuboid, form a lattice, such as the one presented in Figure 1, with the core cuboid being the one at the bottom of the lattice and the cuboid with the coarser level of aggregation (i.e., the one that contains a single cell with the total value of sales) at the top (ALL). Tools that implement data cubes allow users to ask a variety of queries whose answers are summaries of the data for a subset of the dimensions of the dataset (e.g., "what were the total sales of *shoes* for the first quarter of 1999?").

As it is the case in most database systems, the tradeoff between space and time plays a crucial role in data cube implementations. If one materializes all the possible cuboids, in order to be ready for any

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*This work has been supported by NSF grant IIS-9732113

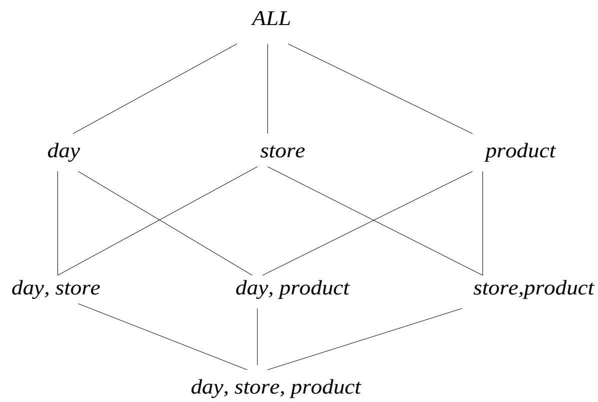

Figure 1: Lattice for the data cube with dimensions $day, store, product$.

query posed by the users, the space needed to store the materializations grows enormously, but the queries are answered very fast. If one does not materialize anything, but takes the "lazy" approach of computing the aggregates on demand, the demand for space is reduced considerably at the expense of long delays to get the answers. Data cubes can be prohibitively large: consider the previous example with 1,000 stores, 10 years (3,650 days) and 20,000 products. Materializing just the core cuboid would require storage for 73 billion aggregate values. Even if the data is sparse (not all aggregates are present) and only 1 % of the cells have values, we would still need space for 0.73 billion cells. If we add the cells for other cuboid and the fact that in many cases users want to see aggregates for all the elements on the hierarchy of dimensions (e.g., the values on the dimension *day* might be viewed by $month$, $quarter$, $year$ and so on), the space demands grow considerably. On the other hand, users of data cube technology demand fast answers to their queries, so they can quickly find areas of interest within the cube space and investigate trends and patterns in them. This tradeoff forces the designer to choose carefully which cuboids provide the most efficient choices for materialization. Strategies to select a subset of cuboids are presented in [11]. These techniques aim to control the space used in materialized cuboids while keeping the response time within acceptable limits.

Alternatively, researchers have begun to investigate ways to compress the data cube in such a way that only a fraction of the space taken by the whole cube is needed [1, 3, 4, 7, 20, 22]. Since the compression techniques are lossy, one can only provide approximate answers to the queries posed to the data cube. On the other hand, the queries can be answered without incurring into much disk I/O, so the response time is considerably smaller than the one experienced in uncompressed data cubes. Of all these techniques, ours [3, 4, 7] is the only one that allows the designer to establish guarantees for the errors incurred in the answers in such a way that the errors are kept below a predefined threshold, *regardless of the distribution of the data.* (All the other techniques can guarantee error levels, but they vary with the underlying data distributions and cannot be fixed by the designer.) We achieve that by insisting in retaining cells of the cube whose values, when recreated by the decompression algorithms differ from the real values in more than the tolerated threshold. That way, our compressed cubes, or *Quasi-Cubes* are composed by models used to estimate cells and a set of "outlier" values, or retained cells, that if estimated would incur in errors whose magnitude exceeds the predefined threshold.

In this paper we present a brief summary of our Quasi-Cube technique along with a list of possible applications that go beyond simply providing approximate answers to aggregate queries. Information about the project in which this paper is based can be found in [5]. The paper is organized as follows.

In Section 2 we briefly summarize how Quasi-Cubes are implemented (more details can be found in [3, 4, 7]). Section 3 lists the applications enabled by this technology. Finally, Section 4 offers some conclusions.

2 Quasi-Cubes

We model regions of the core cuboid and employ these models to estimate the values of the individual cells. The reason to focus on the core cuboid is simple: the error guarantees for queries to the core cuboid hold for any other cuboid in the lattice. (In practice, the errors incurred when aggregating estimated cells of the core cuboid decrease dramatically because they tend to cancel each other.) As we stated in the introduction, to avoid incurring in large errors by the estimation, we retain all the cell values whose estimations are farther away from the real value by more than a pre-established threshold. This threshold becomes the guarantee of the approximate answer has. (As we will show later, many answers are, in reality closer to the real answer than what the threshold predicts.) We store the model parameters (for each modeled region of the cuboid) along with the retained cells to process the queries. The choice of models, although important because some models produce a better fit than others, is orthogonal to the Quasi-Cubes idea. We have experimented with two types of models: linear regression [4] and loglinear models [7]. (For a review of data reduction models see [2].)

The issues involved in compressing the cube, giving approximate answers to the queries and evaluating our technique can be summarized as follows:

- Selecting chunks of the core cuboid that will be described by models (regions of the core cuboid that are sufficiently dense). We do this by dividing the core cuboid space, usually starting with a regular grid and classifying the chunks into three types:
    1. Dense chunks, which are candidates for modeling.
    2. Empty chunks, i.e., sub-spaces that do not contain any data cells and can be discarded.
    3. Sparse chunks, which contain very few cells. Cells in these types of chunks are retained without attempting to model them.
- For each chunk to be modeled, computing the model parameters based on the data contained in the chunk and then, based on the estimated values compute for each non-zero cell in the chunk the estimation error and determine if the cell needs to be retained.
- Organizing the model parameters and retained cells to efficiently access them when processing queries.

Dividing the core cuboid in dense chunks is of outmost importance. Dense chunks render better-fitting models which in turn results in a higher compression rate. Moreover, finding enough dense chunks is the key to a good compression rate. Fortunately, most datasets in practice are skewed, i.e., the non-empty cells are clustered in regions of the core cuboid. However, finding these sub-spaces is far from trivial. So far, we have taken the approach of dividing the core cuboid in a regular grid, identifying potentially dense chunks, and subdividing those as needed until a sufficiently dense region is found. We are currently experimenting with techniques to "re-shuffle" the order of the rows in some (or all) dimensions to increase the density of regions. (For some type of dimensions, e.g., categorical attributes such as *product* in our example, it is irrelevant in which order the values are kept, while for some others, such as *time*, the original order must be preserved.)

Computing the models is a step whose implementation depends, of course, on the choice of models. (Details of how to efficiently model chunks of data using linear regression can be found in [4]; for loglinear models, see [7]; for kernel estimations see [20], and for wavelets [22].)

The dense chunk descriptions are organized as blocks of data containing the model description, parameter values, outliers and and index (we index either non-empty cells or empty cells, depending on which set is smaller). These descriptions are smaller than the data they characterize and thus, many of them can be brought to main memory at once. Indexing of the chunks can help in quickly finding which chunks are needed to answer a query.

3 Applications

In this section we list the applications that are enabled by Quasi-Cubes.

3.1 Approximate query answering

Of course, this is the idea that motivated cube compression in the first place. A range query over the cuboid can be decomposed as the union of several disjoint queries, each spanning a chunk in the cuboid. That being the case, for each one of the disjoint sub-queries there are two possibilities:

- The sub-query completely includes its respective chunk. In this case, the answer of the sub-query can be immediately obtained from the chunk description, which also contains the aggregated value of all the cells in the chunk. Notice that this value is free of error and that the answer to this sub-query does not require retrieving any of the values in the retained list of cells for the chunk or estimating any of the cells values.

- The sub-query covers the respective chunk only partially. In this case, we need to estimate or retrieve (from the list of retained cells) the individual cell values and aggregate them.

In larger queries, i.e., aggregates over large portions of the core cuboid, many sub-queries span complete chunks and therefore, the running time is sped up considerably. Our experiments [7] show a considerable gain in execution time for approximate queries when compared with the same queries posed over the uncompressed cube.

3.2 On-line aggregation

On-line aggregation [10], is the process by which one can streamline the computation of more refined answers to the queries as the user is looking at the current estimate. The aim of this method is to reduce the latency of the query, by offering answers long before the final, correct answer is computed. With Quasi-Cubes, there is a relatively easy way to implement this: using different error thresholds, classify the data into *error bins*, each bin corresponding to an error level. This leads to a series of chunk descriptions, corresponding to each error lever. To produce the first estimate, only the data in the highest-error chunk description needs to be brought from disk; to refine the answer, successive bins are brought to memory.

Hellerstein et al. in [10], proposed to implement on-line aggregation by incrementally sampling the base relations that underlie the data cube. Other researchers [1] have shown that taking samples on the base relations leads to problems such as the lack of uniformity in the likelihood of cells appearing in the answer. The work in Join Synopses [1] attempts to remedy this by sampling the actual joins. However, a common problem with techniques that use sampling is that the samples must be refreshed periodically to avoid showing bias in the query answers. Our approach, on the other hand, does not exhibit any of these problems.

3.3 Supporting query answering in update periods.

While the data warehouse is being updated, the system is usually made unavailable for query answering. (Doing otherwise may create consistency problems in the answers or slow the updating considerably.) For that reasons, researchers [14, 17] have been investigating ways of reducing the window of time needed for performing updates in the warehouse. However, as organizations become more global, shutting down the querying system at any time of the day becomes a difficult proposition.

Quasi-Cubes can help in enabling approximate query answering while the actual data cube is being updated. The answers will, of course, be approximations of the state of the cube before the update, but for many cases this will be an acceptable tradeoff. Notice that since the Quasi-Cube description (chunk descriptions) can be maintained decoupled from the actual data in the cube, the users will be accessing a separate database while the warehouse is being updated. (Of course, this application would not support data cube compression and it assumes that space is not a problem.) The Quasi-Cube descriptions need, of course, to be updated (we have shown inexpensive ways to do this in [7]), but this can be done after the warehouse update has finished (then the query processing can be redirected back to the real cube).

3.4 Lighter Data Marts

The chunk descriptions can be made to map areas of the core cuboid that correspond to data which is of interest to a particular division in the organization. These sub-cubes are commonly known as Data Marts, and many commercial tools made provisions to export the Data Mart to remote clients. Using the Quasi-Cube technology, these Data Marts can be made extremely "light," with respect to the space they occupy in the remote client at the expense of providing only approximate views of the relevant data. (More refined views can always be obtained by going to the data warehouse.) This idea is very much in line with the opinion of some data mining experts who claim that the trend should be in exporting models and not data to the clients [8].

3.5 Approximate Data Mining

The ultimate goal of building a data cube is to extract knowledge from the data stored in it. To this end, we are currently investigating data mining techniques that can benefit from the usage of Quasi-Cubes. The idea is to perform mining in the compressed form of the data, thereby getting the mining task to complete quickly at the expense of having approximate results. We present here two examples of techniques that we are actively investigating.

3.5.1 Exploratory Data Analysis

Exploratory Data Analysis (EDA) [12, 13, 21] is a suite of widely used techniques that aim to determine which factors have the most influence on data values in a multi-way table, or which cells in the table can be considered anomalous with respect to the other cells. EDA is performed without any a-priori hypothesis in mind: rather it searches for "exceptions" of the data values relative to what those values would have been if anticipated by an statistical model. This statistical model fits the rest of the data values rather well and can be accepted as a description of the dataset. When applied to multidimensional tables, EDA also uncovers the attribute values that have the greatest effect on the data points. As pointed out by [19], these exceptions can be used by an analyst as starting points in the search for anomalies, guiding the analyst work across a search space that can be a very large.

At any level of aggregation, data cubes can be viewed as *multi-way tables*, that can be subjected to EDA. Two traditional ways of performing EDA in tables are *median polish* and *mean polish* [12].

Both methods try to fit a model (additive or multiplicative) by operating on the data table, finding and subtracting medians (means) along each dimension of the table. Starting with one dimension, the method calculates the median (mean) of each "row" [1] and subtracts this value from every observation in the row. Then, the method moves to the next dimension and uses the table resulting from the previous operation to find medians (means) in each row and subtract them from the entries in this row. Of course, doing that changes the medians (means) on the previous dimension rows. The process continues with each dimension and iteratively cycles through the set of dimensions. In principle, the process continues until all the rows in each dimension have zero median (mean), hence the number median (mean) polish. (In practice, the process stops when it gets sufficiently close to that goal.) One gets two kinds of information from this process. First, one gets the *effect* that each row in each dimension has on the model, given by the algebraic sum of the medians (means) that have been subtracted in that row at every step. Secondly, one gets *residuals* in each cell of the table, which tell us how far apart that particular cell is from the value that would have been predicted by the model being fit.

It has been pointed out that the main drawback of the mean polish method is its lack of resistance to outliers (i.e., cells that do not fit the model well). This lack of resistance manifests itself specially when "holes," e.g., missing cells, are present [12]. This is particularly troublesome for data cubes, which are usually sparse (i.e., not every cell has a value). On the other hand, median polish, being more resistant to outliers and holes, is very adversely affected by holes which increase substantially the number of iterations needed by the process [12]. Increasing the number of iterations can drastically impose enormous I/O demands (by requesting many passes over a large dataset) and therefore render the process impractical for large data cubes. Previous work in using EDA for data cubes [19] has chosen to use mean polish, precisely for being less demanding on the number of iterations needed to finish.

We have implemented a method of performing median polish in a Quasi-Cube which uses the chunk descriptions instead of the cube data to perform the polishing of the medians. The results, reported in [6] show that the quality of the results obtained by this method is extremely good, while at the same time the method outperforms the running time of doing median polish on the data itself, allowing the user to perform EDA in cases where the cost of using the standard procedure (over the real data) would be prohibitive.

We are currently investigating extensions of this work such as the obtaining joint influences of more than one attribute value and performing non-parametric tests in the data [23].

3.5.2 Decision Trees

Decision trees [15] are a classification technique, used to forecast membership for a multidimensional data point. For instance, a decision tree could be trained to identify "risky" and "good" customers from multidimensional descriptions of people who apply for loans. Nodes in a decision tree involve testing a particular attribute, usually comparing the attribute value with a constant. The decision tree needs to be trained by using a data set in which each multivariate record has a label indicating the class it belongs to. The classical implementation of decision trees (C4.5 [16]) is meant to be used with memory-resident data sets. An implementation, named SPRINT, designed for large data sets is reported in [18]. SPRINT uses a combination of clever data structures that make it possible to manage disk-resident data sets. These data structures have to be recreated at every step of the training phase, using the disk-resident data set. SPRINT can still experience a large I/O activity for large data sets, due to the need of swapping the tables in and out of main memory. We are currently investigating the

[1] we use the term "row" to refer to the set of cells in the hypercube which share the same attribute value for one of the dimensions

use of our compression technique to make the decision tree training step scale with large sets. The idea is to drive algorithms such as SPRINT not with the data set itself, but with compressed versions of it. The class or label attribute becomes a dimension of the Quasi-Cube. Once that is done, the decision of what is the question to be asked in the next node of the tree can be answered (approximately) by using the chunk descriptions, instead of going to the data itself (like SPRINT does). A caveat, though, is that as one goes deeper in the tree, the queries that are needed to fill the tables become more and more fragmented, using a larger number of chunk descriptions (i.e., they partially intersect a large number of chunks in the core cuboid). When the time to bring lots of chunk descriptions (I/O time) and estimate cells (CPU time) exceeds that of using the data in the cube (mostly I/O), we need to switch to use the cube data. So, at some level in the creation of the tree, one needs to switch from driving SPRINT from the Quasi-Cubes to driving it from the actual data, and the decision must be done solely on the basis of the increase in chunk processing time. Nevertheless, we expect a significant improvement on the running time of the algorithm due to the effect of the upper levels of the tree (which actually need the bulk of the data to be fetched, since as we go down the tree, the queries become more and more restrictive).

4 Conclusions

We have presented in this paper an overview of Quasi-Cubes, a technique that allows the designer of data cubes to compress the data and still maintain guaranteed bounds for the errors made in querying the cube. We have also presented a list of applications that can substantially benefit from the use of approximations.

References

[1] S. Acharya, P.B. Gibbons, V. Poosala, and S. Ramaswamy. Join Synopses for Approximate Query Answering. In *Proceedings of the ACM SIGMOD Conference, Philadelphia, PA*, June 1999.

[2] 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. *Data Engineering Bulletin*, 20(4):3–45, December 1997.

[3] D. Barbará and M. Sullivan. Quasi-Cubes: A space-efficient way to support approximate multidimensional databases. Tech. Report, Dept. of Information and Software Systems Engineering, George Mason University, 1997.

[4] D. Barbará and M. Sullivan. Quasi-cubes: Exploiting approximations in multidimensional databases. *SIGMOD Record*, 26(3), September 1997.

[5] D. Barbará. Quasi-Cubes: Exploiting Approximations in Multidimensional Data Sets. http://www.ise.gmu.edu/ dbarbara/quasi.html.

[6] D. Barbará and X. Wu. Using Approximations to Scale Exploratory Data Analysis in Datacubes. In *Proceedings of the 1999 ACM SIGKDD Conference, San Diego, CA*, August 1999.

[7] D. Barbará and X. Wu. Using loglinear models to compress data cubes. Tech. Report, Dept. of Information and Software Systems Engineering, George Mason University, 1999.

[8] U. Fayyad. Data mining techniques. Tutorial at Very Large Databases Conference, New York, 1998.

[9] 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 Intl. Conference on Data Engineering, New Orleans*, 1996.

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

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

[12] D.C. Hoaglin, F. Mosteller, and J.W. Tukey. *Exploring Data Tables, Trends and Shapes.* Wiley, 1985.

[13] D.C. Hoaglin, F. Mosteller, and J.W. Tukey. *Understanding Robust and Exploratory Data Analysis.* Wiley, 1986.

[14] W. Labio, R. Yerneni, and H. Garcia-Molina. Shrinking the Warehouse Update Window. Tech. Report, Computer Science Dept., Stanford University.

[15] J.R. Quinlan. Introduction of decision trees. *Machine Learning 1(1): 81-106*, 1986.

[16] J.R. Quinlan. *C4.5: Programs for machine learning.* Morgan Kaufmann, San Francisco, 1993.

[17] N. Roussopoulos, Y. Kotidis, and M. Roussopoulos. Cubetree: Organization of and Bulk Incremental Updates on the Data Cube. In *Proceedings of the ACM SIGMOD Conference, Tucson, Arizona*, May 1997.

[18] J.C. Shafer, R. Agrawal, and M. Mehta, SPRINT: A Scalable Parallel Classifier for Data Mining. In *Proceedings of the Very Large Databases Conference, Mumbai (Bombay), India*, Sept. 1996.

[19] S. Sarawagi, R. Agrawal, and N. Meggido. Discovery-driven Exploration of OLAP Data Cubes. In *Proceedings of Conference on Extending Data Base Technology*, pages 168–182, 1998.

[20] J. Shanmugasundaram, U. Fayyad, and P.S. Bradley. Compressed Data Cubes for OLAP Aggregate Query Approximation on Continuous Dimensions. In *Proceedings of the ACM-SIGKDD Conference, San Diego, CA*, August 1999.

[21] J.W. Tukey. *Exploratory Data Analysis.* Addison Wesley, 1977.

[22] J.S. Vitter and M. Wang. Approximate Computation of Multidimensional Aggregates of Sparse Data Using Wavelets. In *Proceedings of the 1999 ACM-SIGMOD Conference, Philadelphia, PA*, June 1999.

[23] T.H. Wonnacott and R.J. Wonnacott. *Introductory Statistics.* Wiley, 1990.