An Efficient OLAP Cube Generation and Storage Scheme

Myung Kim, Yoonsun Lim, Ji Suk Song

Department of Computer Science and Engineering
Ewha Womans University
11-1 Daehyun-Dong, SeoDaeMun-Ku, Seoul 120-750, Korea
E-mail: mkim@ewha.ac.kr

Abstract: OLAP is a process and methodology for a multidimensional data analysis that is essential to extract desired data and to derive value-added information from an enterprise data warehouse. In order to speed up OLAP query processing, most OLAP systems pre-compute and store analysis results into arrays or tables, called "*cube*". In this paper, we present a fast and scalable cube generation algorithm and propose a cube storage scheme for fast query processing. Our cube generation algorithm has high memory utilization and our cube storage scheme uses the Z-indexing technique to cluster data for queries.

1. Introduction

OLAP(On-line Analytical Processing) is a process and methodology that analyzes and queries data stored in a data warehouse [7, 14]. With data mining [3] and XML/HTML document processing technologies, OLAP is one of the fundamental technologies for today's business intelligence infrastructure. Recently, IT industry has shown a great deal of interests in OLAP, and a lot of research activities are going on in this field. Various OLAP systems have been developed and are already on the market [2, 4, 5, 8, 9, 11].

Multidimensional data analysis frequently requires to scan the entire data set to answer queries. Thus, most OLAP systems generate a cube in advance in order to meet the performance requirement of the applications. For large data sets, cube generation is a very time consuming process, and tremendous amount of data is produced in this stage. Recently, there have been a lot of efforts [1, 15] to increase cube generation speed. [1]'s cube generation algorithms are for the data stored in relation tables (ROLAP cube generation). [15]'s algorithm is for the data stored in arrays (MOLAP cube generation). For relatively dense data sets, MOLAP style cube generation is much faster than that of ROLAP.

In this paper, we present a fast and scalable ROLAP cube generation algorithm. High performance and scalability is achieved by slicing the input fact table along one or more dimensions before generating the cube and by increasing memory reusability.

We also present a MOLAP cube storage scheme for fast query processing. MOLAP systems store their cubes in compressed arrays [15]. Depending on the mapping scheme of a multidimensional array onto disk, the speed of MOLAP operations, such as slice and dice, varies significantly. [12, 15] presented an efficient MOLAP cube storage scheme which divides a cube into small chunks with equal side length, compresses sparse chunks, and stores the chunks in row-major order of their chunk indexes. This gives a fair chance to all dimensions. We have developed a variant of this by placing chunks in a different order, which results in a significant reduction in disk I/O time. The purpose of rearranging the chunks is to align them to disk block boundaries and to cluster neighboring chunks so as to reduce disk I/O time for slice and dice operations.

This paper is organized as follows. Section 2 gives a ROLAP cube generation algorithm. Section 3 gives a cube storage scheme. In section 4, we draw conclusions.

2. A ROLAP cube generation algorithm

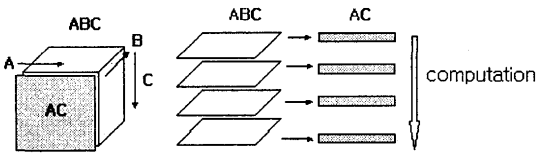
Suppose we are given a 3 dimensional fact table. Assume that the table has four attributes, product (P), store (S), time (T) and sales data (D). Sales data D in each tuple represents the dollar amount of the product P sold in store S at time T. For analysis, sales data can be grouped by P and S. It can also be grouped by S and T. The cube generation is to compute group-bys to all possible combinations of the three attributes, P, S and T. It is easily seen that the number of group-bys in a cube is 2^n for an n dimensional fact table.

We now present our cube generation scheme. Assume that we are given a 3 dimensional fact table ABC as in Figure 1. Let us focus on the computation of AC from ABC. Our objective is to compute AC by scanning ABC only once. In case ABC is not sorted, the entire 2 dimensional array AC should be in memory. However, suppose that ABC is partitioned along C, meaning that all the tuples (or cells) in a partition have the same value of C. In this case, we only

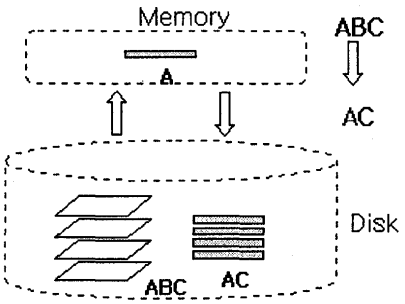
This work was supported by grant No. R04-2001-00191 from the Korea Science & Engineering Foundation.

need a 1 dimensional array A to compute AC. Each time a slice of ABC is read, a slice of AC is computed and moved to disk as in Figure 1(b). This is how we reduce the space requirement for the cube generation.

All three 2-dimensional group-bys can be computed simultaneously. Memory for computing AC and BC is A and B, respectively. Each time a slice of ABC is scanned, the corresponding slices of AC and BC are computed and moved to the disk. However, computation of AB requires space for the entire AB. Note that a group-by whose name does not have the dimension name along which the fact table is sliced needs space for the entire group-by.



(a) Computation of a slice of AC from a slice of ABC.



(b) Memory requirement for computing AC.

Figure 1. Computation of AC from ABC.

After a slice of AC is computed, it can be used to compute the corresponding slice of C. Thus, the group-bys in the dotted area marked with 'step 1' in Figure 2 can be computed with one scan of ABC. And the space needed for this step is AB, A, B, and *all*. Before saving AB to disk, it can be used to do step 2. Since AB is in memory, A, B, and *all* in step 2 can be computed together. Note that space needed for step 1 is the same as that for step 2.

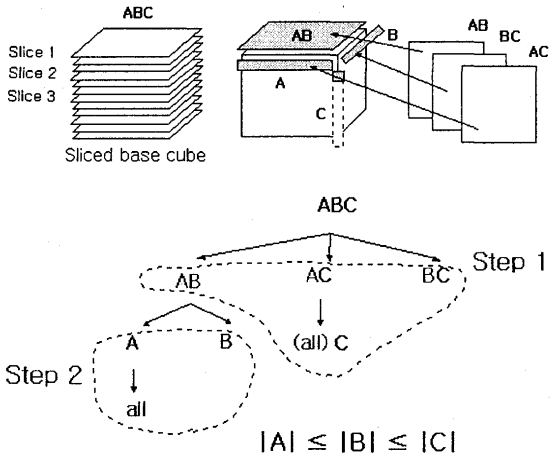


Figure 2. Cube generation from ABC.

Let us now consider a 4 dimensional fact table as in Figure 3. The cube can be computed similarly in two steps. Here the memory requirement for the cube generation is equal to the size of the cube for ABC. If $|A| \leq |B| \leq |C| \leq |D|$ is the case, the algorithm guarantees that all group-bys are computed from their smallest parents. Memory space needed for the cube computation is $(|A|+1)(|B|+1)(|C|+1)$.

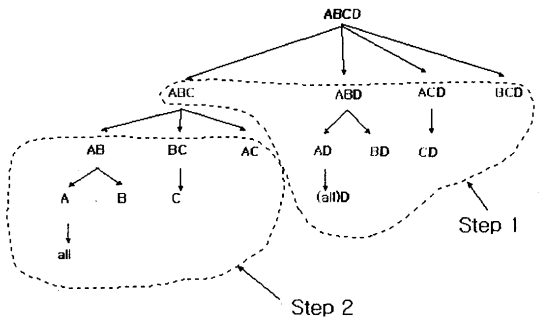
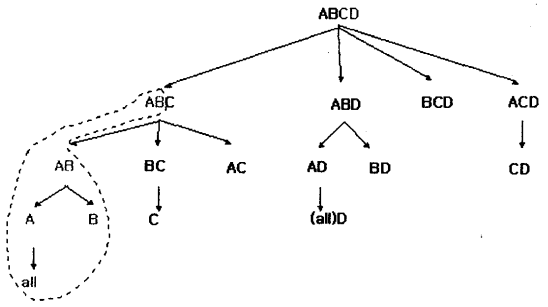


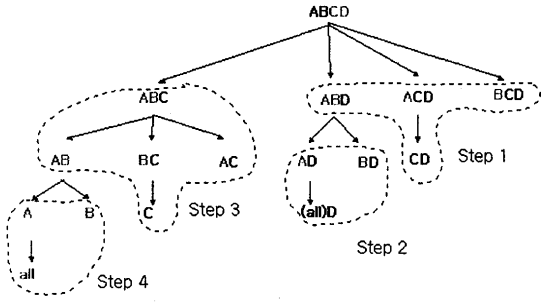
Figure 3. 4 dimensional cube generation

In case there is not enough memory space for the cube generation, we use more than one dimension for partitioning the fact table as in Figure 4. The above idea can be recursively applied. And the space requirement gets reduced significantly. For example, in case of Figure 4, fact table $ABCD$ is sliced along CD. And the memory requirement for the cube computation is what is needed for the group-bys in the dotted area of Figure 4(a). Here, they are AB, AB, A, B, and *all*.

When two dimensions are used for fact table slicing, the cube computation is carried out in four steps, as in Figure 4(b). We compared our scheme with the fastest known MOLAP cube generation algorithm in terms of the space and time complexities. It showed that our scheme is faster and more scalable.



(a) memory requirement for cube computation with the fact table sliced along CD.



(b) memory requirement for cube computation

Figure 4. 4 dimensional cube computation steps (fact table is sliced along CD).

3. A Z index based cube storage scheme

MOLAP with proper data compression is in general faster than ROLAP in the process of cube generation as well as OLAP operations [15]. MOLAP cube generation can be done so fast that a direct ROLAP cube generation is slower than converting a fact table to a MOLAP base cube, generating a MOLAP cube, and then transforming the resulting cube back to a ROLAP cube. Here, we analyze existing MOLAP cube structures, and design a new cube storage structure which is more efficient with respect to space and time complexities.

Let us first examine [15]'s MOLAP cube storage scheme. An example of a 3 dimensional base cube (or fact table) is shown in Figure 5. The base cube is divided into chunks. They are numbered in linear order and are stored in sequence in the data file. The chunk size is normally chosen as the disk block size so that each chunk can be read into memory independently [12].

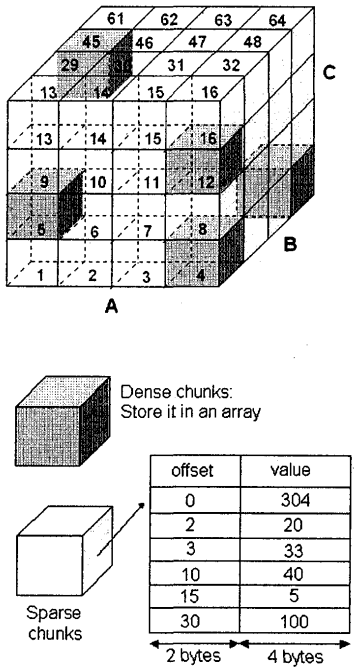


Figure 5. Chunk based MOLAP cube .

Dense chunks, whose density is above 40%, are stored as is. Sparse chunks are stored differently. Only the valid cells are taken and are stored with their offsets inside the chunk. The offset of a cell is the linear number of the cell inside the chunk. Due to such data compression, chunks became different in shapes and sizes. Thus, their sizes and positions in the file are stored separately as meta data.

We made the following observations. First, dense chunks are not aligned to disk block boundaries. This means that two disk block reads are needed to bring one dense block to memory in average. This can be avoided by reordering the chunks in the file.

Second, it is common that the base cube density is 0.1%~1%. In such cases, hundreds of very sparse chunks can be packed and stored into a disk block. These chunks are also along a particular dimension since they are stored in linear order. This would cause excessive disk I/O's when slice/dice operations are applied along other dimensions. This situation can also be avoided by reordering the chunks in the data file.

We now propose a new cube storage scheme that solves these problems. As with [15]'s method, the cube is divided into chunks. What makes our scheme different from [15]'s method is that chunks are stored in different order, called *the Z-index order*. We begin our discussion with describing the Z-index order. The details of the proposed scheme will follow.

(a) The Z indexing scheme

The Z indexing scheme is a popularly used method of

numbering pixels of a 2 dimensional image (or a 3 dimensional image) [13]. It numbers neighbor pixels before numbering far apart ones so that image component identifications can be easily made. Let us first explain how to apply the method to a 2-dimensional array. Consider an 8 × 8 array in Figure 6. The array cells in Figure 6(a) are numbered with the row-major order (or linear order), the array cells in Figure 6(b) are numbered with the shuffled-row-major order (or Z index order).

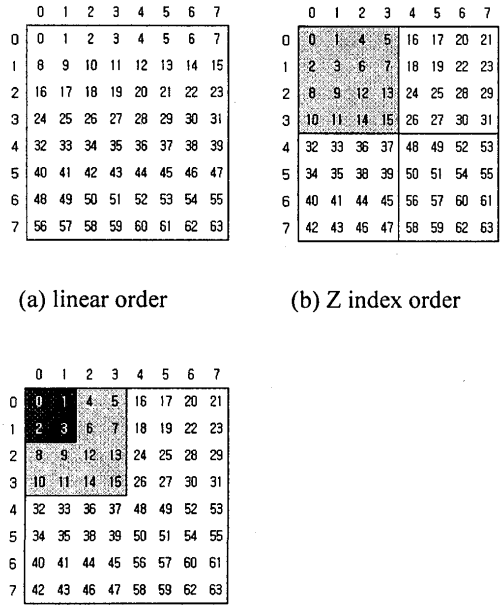


Figure 6. A Z indexed 2 dimensional array.

Consider a 2 dimensional array with $2^n \times 2^n$, $1 \le n$ cells. It is divided into 4 square blocks, each with $2^{n-1} \times 2^{n-1}$ cells. These blocks are called NW, NE, SW, and SE blocks. In order to assign Z-indices to array cells, we first number all the cells in the NW block, followed by all the cells in the NE block, followed by all the cells in the SW block, and followed by all the cells in the SE block. If the block has only one cell, the next available number is assigned to the cell, otherwise the block is recursively decomposed, and a similar method is applied. Figure 6(c) shows how to assign Z-indices to the cells in the NW block.

The Z indexing scheme can be extended to higher dimensional arrays. The first level decomposition of a d dimensional array produces 2^d blocks. For a 3 dimensional array, 8 cubic blocks are produced as in Figure 7. Each block is recursively decomposed in a similar fashion.

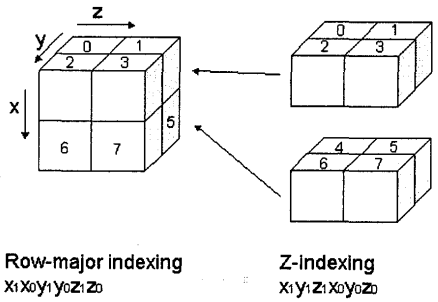


Figure 7. A Z indexed 3 dimensional array.

(b) A new MOLAP cube storage scheme

We propose a MOLAP cube storage structure that uses the Z-indexing scheme. We explain how to store a sub-cube (or a summary table). The entire cube can be stored similarly. A sub-cube is decomposed into chunks as [15]'s cube. Chunks are numbered with the Z indexing order. And they are classified into 2 groups (D group and S group) as in Figure 8. A chunk belongs to the D group if it is dense. A chunk belongs to the S group if it is sparse. D group chunks are stored at the beginning of the file. They are followed by the chunks in the S group. In each group, chunks are stored in increasing order of their Z indices.

The main purpose of classifying chunks of a sub-cube into two groups, and storing them separately is to solve the problems that were brought up in the previous section. By having the D group, we make all the dense blocks aligned with the disk block boundaries. It solves the problem of reading two disk blocks in order to read one dense chunk.

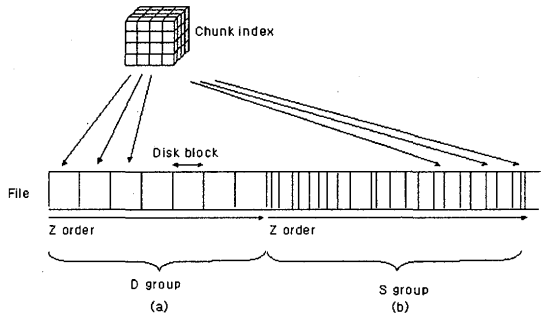


Figure 8. Storage structure of a sub-cube

By having the S group, we solved the second problem that is actually a suggestion to cluster nearby sparse chunks into a disk block as much as possible. Suppose that 8 3 dimensional chunks can be stored in a disk block. This means that what can be stored in a disk block is a "virtual" big chunk whose side length of each dimension is two times that of the original chunk. We call this *virtual big chunk effect*. Virtual big chunk effect can be used to reduce disk I/O's for slice or dice operations. We showed the new scheme

reduces disk I/O time through experiments.

4. Conclusions

In this paper, we presented a fast and scalable cube generation algorithm and a cube storage scheme that supports fast OLAP query processing. Our cube generation algorithm is fast and highly scalable. It works well especially for relatively dense data. The proposed cube storage scheme reduces disk I/O time significantly for slice and dice operations. However, the sizes of sparse chunks are very small so that the scheme needs relatively large indexes. This situation can be avoided by having one index for a group of sparse chunks. This can be done since sparse chunks are stored in Z index order. We also developed such an indexing scheme and the results will be published soon.

References

[1] Agarwal, S., R.Agrawal, P. M. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, S. Sarawagi, "On the Computation of Multidimensional Aggregates," *Proc. 22nd VLDB*, Mumbai (Bombay), India, 1996.

[2] Arbor Software Corporation, "Method and Apparatus for Storing and Retrieving Multi-dimensional Data in Computer Memory," *United States Patent 5,359,724*, Oct. 25, 1994.

[3] Berry, M. and G. Linoff, "Data Mining Techniques for Marketing, Sales and Customer Support," Wiley, New York, 1997.

[4] Mitsubishi Electric Corp, "DIAPRISM: An Accelerator for Data Warehouse/Data Mart, and Business Intelligence Solutions," *White Paper*, http://www.melco.co.jp/service/diaprism/index-e.htm.

[5] Hyperion Corp. "Large-Scale Data Warehousing Using Hyperion Essbase OLAP technology," http://www.hyperion.com/downloads/teraplex.pdf, Jan 2000.

[6] Information Advantage, "OLAP-Scaling to the Masses", *White Paper*, Information Advantage, http://www.infoadvan.com/

[7] Kim, W. and M. Kim, "Performance and Scalability in Knowledge Engineering: Issues and Solutions," *Journal of Object-Oriented Programming*, Vol. 12, No. 7, pp. 39-43, Nov/Dec. 1999.

[8] Microsoft Co. "Overview of Microsoft SQL Server 7.0 OLAP Services," http://msdn.microsoft.com/library/backgrnd/html/olapover.htm

[9] Micro Strategy Incorporated, "A Case for ROLAP,"http://www.microstrategy.com/files/whitepapers/wp_rolap.pdf .

[10] http://www.olapreport.com/DatabaseExplosion.htm.

[11] Oracle Corporation."Oracle Express Server: Delivering OLAP to the Enterprise," http://www.oracle.com/datawarehouse/products/servers/express/documents/oe_olap.pdf

[12] Sarawagi, S. and M. Stonebraker, "Efficient Organization of Large Multidimensional Arrays," *Proc. of 10th Data Engineering Conference*, Feb. 1994.

[13] Samet, H., "Application of Spatial Data Structures - Computer Graphics, Image Processing, and GIS," *Addison Wesley*, 1990.

[14] Thomsen, E. "OLAP Solutions: Building Multidimensional Information Systems," *John Wiley & Sons*, New York, 1997.

[15] Zhao, Y., P. Deshpande, J. Naughton, "An Array-Based Algorithm for Simultaneous Multidimensional Aggregates," *Proc. ACM SIGMOD '97*, pp. 159-170.