Improving Main Memory Utilization for Array-Based Datacube Computation

Seigo Muto Masaru Kitsuregawa
Institute of Industrial Science
University of Tokyo
{muto,kitsure}@tkl.iis.u-tokyo.ac.jp

Abstract

Computing datacubes requires multidimensional aggregations for all possible combinations of each dimension. In this paper, we present a method to improve main memory utilization efficiency for an array-based algorithm for datacube computation in a MOLAP context. The problem with the array-based algorithm is in its sparsity, where a large proportion of array cells are empty. The algorithm proposed in [ZDN97] reduces this space inefficiency by compressing arrays on disk. We improve on this algorithm by performing compression of arrays in main memory as well as on disk using a hashing method, which allocates main memory according to the number of non-empty array cells. We further improve the algorithm using a dynamic main memory allocation strategy. The algorithm by [ZDN97] computes the multiple aggregate views simultaneously, which consumes a lot of main memory space. We propose a main memory allocation method that minimizes the main memory requirement by dynamically allocating main memory only to necessary aggregate views at run time. These savings in main memory resources result in the reduction of disk I/O cost. We evaluate the performance of the proposed method by disk I/O analysis and demonstrate that the improved MOLAP algorithm compares well with a ROLAP algorithm.

1 Introduction

OLAP is one of the new applications emerging in database technologies in recent years and allows users to easily analyze large volumes of data in detail from various viewpoints for use in decision making. Multidimensional aggregation plays an important role in OLAP technology in providing such an environment for users. However, since computing these aggregations, even for a small number of dimensions, is very expensive, sophisticatied techniques are essential for obtaining good performance.

Several techniques have already been proposed in the past[AAD+96, DANR96, RS97, SAG96, ZDN97]. These techniques can be categorized into two groups called ROLAP and MOLAP. The ROLAP approach is based on relational databases, which are widely used today. In this approach, systems directly access and process tuples in the relations as data elements. On the other hand, MOLAP systems use a multidimensional data structure such as an array constructed from the original data, which are typically stored in relational databases. Although the MOLAP approach needs this loading process, it is possible to select data structures optimized for multidimensional data manipulations.

[ZDN97] proposes an algorithm using the MOLAP approach based on an array structure and shows that their method, even including the loading process, outperforms the ROLAP algorithm presented in [AAD+96, DANR96], where aggregations are performed after sorting tuples in contrast to directly aggregating values in the array cells as in the array approach. The problem with an array structure is its sparsity, which wastes main memory because many array cells are empty and thus are not used during the computation. In particular, the sparsity problem arises when the number of dimensions increases. This is because the number of all possible combinations of dimension values exponentially increases, whereas the number of actual data values would not increase at such a rate. In [RS97], attempts to solve this sparsity problem are made using the ROLAP approach, which employs a sort-based method developed in [AAD+96, SAG96], where multiple aggregations are overlapped in a pipeline fashion after sorting tuples.

In this paper, we propose another solution to handle this sparsity problem in the context of the MOLAP approach and show that this solution compares well with the sort-based ROLAP approach adapted to deal with high sparsity. In the array-based algorithm, the processing is done on a chunk by chunk basis. A chunk is a unit of processing used in this algorithm and compressed on disk when more than a certain number of cells are empty. For chunks in main memory, however, array data is not compressed in this algorithm. We overcome this main memory ineffeciency in its array-based algorithm by introducing compression for the array structure in main memory as well as on disk using a hashing method, which can determine the size of main memory to be allocated depending on the amount of actually existing data.

In addition, we provide further improvement for this algorithm using dynamic main memory allocation. While computing a datacube, main memory is allocated to multiple aggregate views in the datacube because these aggregations are performed simultaneously. In the original array-based algorithm[ZDN97], this main memory allocation is performed statically. The number of aggregate views that use main memory at the same time can be minimized by dynamically allocating main memory only to those aggregate views in use at that point in time.

As a result of these enhancements in main memory use, disk I/O can be reduced considerably. To examine the effectiveness of the proposed methods for the array-based algorithm compared to the ROLAP algorithm, we evaluate the performance by analyzing disk I/O cost since the performance of these algorithms is expected to be disk I/O bound.

The rest of the paper is organized as follows. In Section 2, we briefly introduce the array-based algorithm proposed in [ZDN97] and then describe the proposed hash-based compression method and the dynamic main memory allocation method. In Section 3, we present performance evaluation results to demonstrate the effect of these proposed improvements. In Section 4, we discuss our conclusions and future work.

2 Improving Main Memory Utilization of Array-Based Algorithm

Multidimensional aggregation is formalized as a datacube in the relational context in [GBLP96]. In a data cube, aggregates of values of a measure attribute in a relation are computed with respect to all possible combinations of each dimension attribute. For example, if a data cube has 3 dimensions A, B and C, aggregate views that should be computed are ABC, AB, AC, BC, A, B, C and a view obtained by aggregating all values denoted by $\{\}$. Aggregate functions such as count, sum, max, min, average have a distributive property defined in [GBPL96]. In these functions, there are dependencies where aggregate views in a data cube are computed from other aggregate views. For example, A can be computed from AB, AC or ABC. For this reason, the computations of these aggregate views can be overlapped. Like other methods, the array-based algorithm makes use of this fact to achieve efficient processing of datacubes.

2.1 Array-Based Algorithm

In this subsection, we briefly describe the array-based algorithm proposed in [ZDN97]. One of the features of this algorithm is "chunking" arrays. Multidimensional arrays are partitioned into chunks, based on the values of each dimension. Chunks are processed as they are read from disk one after another in a certain order. Consider the example given in Figure 1, which depicts dependencies amongst aggregate views in a data cube consisting of 4 dimensions A, B, C and D. Dependencies are determined such that an aggregate view is computed from the "smallest parent" aggregate view in order to minimize the aggregation cost pointed out in [GBLP96]. In this paper, we refer to the number of possible values for each dimension as the dimension size and the partitioned dimension size within a chunk as the chunk dimension size. We denote each dimension size by n_A, n_B, n_C and n_D. Figure 1 shows an optimized dependency tree in the case where $n_A \geq n_B \geq n_C \geq n_D$.

Chunks are read from disk in a left dimension major order. Suppose that dimension X is divided into n partitions named X_0, X_1, X_2, ..., X_n. We first read chunks such that only the chunk number of the dimension D increases as $A_0B_0C_0D_0$, $A_0B_0C_0D_1$, $A_0B_0C_0D_2$, ..., $A_0B_0C_0D_n$ in this example. When the chunk number of the dimension D equals n, we change the chunk number of the dimension C to the next chunk number and then increase the chunk number for the dimension D again from the first chunk in the same way as in $A_0B_0C_1D_0$, $A_0B_0C_1D_1$, $A_0B_0C_1D_2$, ..., $A_0B_0C_1D_n$. In this way, reading continues until chunk $A_nB_nC_nD_n$ is read.

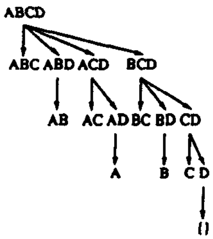

Figure 1: Dependency tree

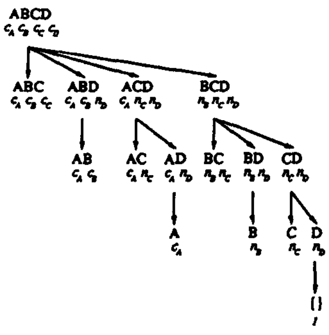

Figure 2: Main Memory cost

In main memory, array cells for each aggregate view are reserved to store aggregated values. Each time a chunk is read from disk, values in the chunk are aggregated into the corresponding array cells for each aggregate view. If we refer to the chunk dimension size of dimension X as c_X, the number of array cells needed for each aggregate view during processing will be as illustrated in Figure 2. For example, for an aggregate view ABD, array cells, size $c_Ac_Bn_D$, are required to reside in main memory. This is because for the dimension A and B, only values in the specified range are processed until all chunks containing the values in that range are exhausted, whereas for the dimension D, every possible value would be processed during that time. If $n_A \geq n_B \geq n_C \geq n_D$, the number of array cells required is minimized, since for this scheme, the larger dimensions use the chunk dimension size in determining the number of array cells required.

If the number of aggregate views becomes large, all the aggregate views may not fit in main memory. When this happens, the dependency tree must be divided into multiple subtrees. The problem is how to divide the tree into subtrees such that each subtree fits into the available main memory. This problem is likely to be NP-hard, hence the use of a heuristic algorithm in allocating main memory in a breadth-first manner. For each subtree that overflows out of main memory, only one chunk is allocated for the intermediate results to be materialized. Then, each subtree is processed in the same way after creating chunks from these intermediate results.

2.2 Hash-Based Compression

In the array-based algorithm, chunks are compressed on disk if many array cells in the chunks are empty. This is achieved by using a "chunk offset" that can be added to the address of the first element in the chunk to obtain the address of the corresponding chunk element. In a compressed chunk, only the array cells containing a value and their corresponding chunk offset are retained. Let s_m, s_i, t and n denote the size of an array cell, the size of the chunk offset, the number of distinct values and dimension size respectively. If $s_m n > (s_m + s_i)t$, the chunk should be compressed. This compression improves the disk I/O efficiency significantly.

Likewise, not all array cells in main memory can be used during computation. When the sparsity is very high, this could be very wasteful in terms of main memory utilization. Performance can be improved by using main memory more efficiently. Hence we propose a hash-based compression method to compress array cells in main memory as well as on disk. Like compression on disk, empty array cells do not have to be retained in main memory in this method. An index value representing a location in a multidimensional space, which corresponds to a chunk-offset for compression on disk, is calculated from the address of an element in a chunk being read from disk. For example, for the address (i, j, k) of a chunk element and dimension sizes n_i, n_j and n_k, the index value might be $(in_j + j)n_k + k$. Hashing is applied to this index value. Suppose that the size of measure values is s_m, the size of index values is s_i, the size of a pointer is s_p, dimension size is n and the number of distinct values is t. Assuming that we use a hashing method based on pointer chains, the main memory space required for this method is as follows.

$$(s_m + s_i + s_p)ft$$

The fudge factor f takes into account the collision of hash entries. If $s_m n > (s_m + s_i + s_p)ft$, compression should be applied to the corresponding array.

2.3 Dynamic Main Memory Allocation

In addition to compression in main memory, we propose another method for further improvement of the main memory utilization efficiency in the array-based algorithm. The idea is to minimize the main memory requirement by dynamically allocating main memory only to aggregate views necessary at that point in time. In the original array-based algorithm, main memory is allocated to as many aggregate views as possible before any chunks are read from disk. Once processing starts, the allocated main memory is not released until the computation of those aggregate views is completed. In our method, the unused main memory is released and reallocated to other aggregate views dynamically even during processing. Consider the computation of the aggregate views AC, AD and A in Figure 1 for example. While these aggregate views are being computed in main memory, no main memory is required to be allocated for the computation of views such as $ABCD$, ABC, ABD and AB. Since only main memory required by the current computation is allocated, significant savings in main memory can be achieved.

The main memory allocation process proceeds as follows. First, we allocate main memory to $ABCD$, ABC, ABD, ACD and BCD in that order as long as we have sufficient main memory. We assume that all of these five aggregate views fit into main memory. Chunks are read from disk one after another, from $A_0B_0C_0D_0$ to $A_nB_nC_nD_n$. Each time a new chunk is read, the corresponding aggregate views are computed. When the chunk $A_0B_0C_nD_n$ is read, we can, in addition to the computation of the above aggregate views, also compute AB from ABD. Instead of allocating new main memory to AB, we can release the main memory for $ABCD$ and ABC, which is not needed at this time, and reuse it for AB. Similarly, the arrival of $A_0B_nC_nD_n$ enables us to compute AC, AD and A from ACD. For this computation, the main memory for other aggregate views can be released with the exception of that for ACD and BCD, which must be reserved in main memory for later use.

However, it should be noted that reusable main memory may not be sufficient for the purpose to which it is to be reallocated. First we allocate main memory to a root aggregate view and its child aggregate views, which are $ABCD$, ABC, ABD, ACD and BCD in this example. In the original array-based algorithm, the main memory allocated first is always sufficient for computing other aggregate views. This is not the case, however, for our algorithm which includes compression in main memory. Depending on the data distribution, the main memory allocated first can be insufficient because the density at the lower dimension is usually higher than the density at the higher dimension. For this case, maximum main memory requirement is equal to the main memory required for aggregate views, which are obtained by recursively searching for the second child aggregate views from the right, starting from the rightmost child aggregate view of the root in the tree. This is because the right aggregate views are larger than the left aggregate views and the parent aggregate views of the second child aggregate views from the right must reside in memory for computing the rightmost child aggregate views. If there is only single child aggregate view, the search is done and the last one is added to these selected aggregate views. In this example, the main memory for BCD, BD and B can be maximum main memory requirement.

Another advantage of this method is that it is simple to determine how to divide a dependency tree into subtrees if main memory is not large enough to hold all aggregate views in a datacube. As stated above, the original algorithm uses a heuristic algorithm because this problem is likely to be NP-hard. In our method, however, we can always determine the division of the tree by only examining how many child aggregate views directly connected to the root of the tree can be placed in main memory. The process of dividing the tree into subtrees proceeds recursively. If a child aggregate view can not be placed in main memory, it in turn is considered a root and its child aggregate views are examined to see whether they can be placed in main memory. The process recursively continues until subtrees that fit into main memory are obtained. In the example in Figure 1, if only ABC and ABD fit into main memory, the tree is divided into subtrees ST1, ST2 and ST3 as in Figure 3. Moreover, in the subtree ST3, if only BC and BD can be staged into main memory, the subtree is further divided into subtrees ST4 and ST5 as shown in Figure 3. Unlike the original alogrithm, we create no intermediate results to avoid the extra disk I/O. Instead, we take as input an aggregate view that is already materialized on disk, from which the root of each subtree can be computed directly.

3 Cost Model

In this section, we describe an analytical cost model that calculates the amount of main memory required for the improved array-based algorithm described proposed in this paper.

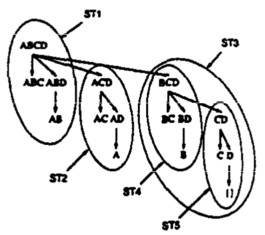

Figure 3: Subtrees

3.1 Preliminaries

We begin by making some assumptions for simplification. Assume that data is uniformly distributed and all dimensions has the same dimension size n. Given that the maximum number of dimensions in a datacube is k and the number of distinct values is t, the upper bound of the density, $d(i)$, in the case where the each number of dimensions in the datacube is i is as follows.

$$d(i) = \begin{cases} t/n^i & \text{if } n^i > t \\ 1 & \text{otherwise} \end{cases} \quad (i = 0, 1, 2, ..., k)$$

This is because the number of actually existing values in the lower dimension will not be larger than the higher dimension. However, for the dimension in which the number of possible values is smaller than the number of actually existing values in the higher dimension, the number of actual values is equivalent to the number of possible values. In a hashing method, we must estimate the hash table size in advance. This upper bound can be used for this purpose. At the same time, this function is considered to be approximately equal to the density in the case where the data is uniformly distributed, since the possibility that different measure values in a datacube will share the same dimension values is extremely low due to the randomness of the distribution. For this reason, we use this function as the density of uniformly distributed data.

3.2 Main Memory Cost

Figures 4 and 5 show the main memory requirement of each aggregate view in a 4 dimensional datacube when using an array structure and hashing respectively. c stands for the chunk dimension size, which is assumed to be the same in all dimensions. As we can see from these figures, the difference between them is the density. The main memory cost for an array is not affected by the density. On the other hand, for hashing the density must be taken into account. As shown in Figure 5, the density is simply multiplied by the number of array cells. This is possible because of the assumption that data is uniformly distributed.

We now derive the main memory cost for the improved array-based algorithm. First, we take an aggregate view $ABCD$ for example. As stated in the previous section, if $s_m c^4 > (s_m + s_i + s_p) f c^4 d(4)$, we should use hashing rather than an array for the aggregate view. Consequently, the main memory requirement for $ABCD$ is as follows.

$$\min\{s_m c^4, (s_m + s_i + s_p) f c^4 d(4)\}$$

This can be applied to other aggregate views in the same way. Using these main memory costs, we derive the total main memory requirement for datacube computation. As we described earlier, main memory, allocated dynamically,

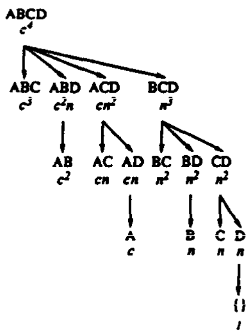

Figure 4: Main Memory cost for array

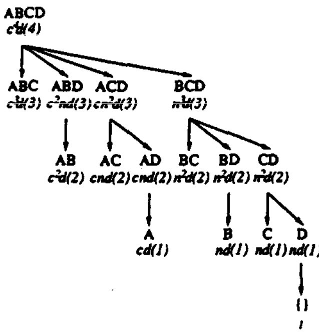
ABCD
$c^4d(4)$
ABC ABD ACD BCD
$c^3d(3)$ $c^2nd(3)$ $cn^2d(3)$ $n^3d(3)$
AB AC AD BC BD CD
$c^2d(2)$ $cnd(2)$ $cnd(2)$ $n^2d(2)$ $n^2d(2)$ $n^2d(2)$
A B C D
$cd(1)$ $nd(1)$ $nd(1)$ $nd(1)$
{}
1

Figure 5: Main Memory cost for hash

can be maximum in two cases. We use j to denote the jth child aggregate view. j increases from the left child aggregate view to the right aggregate view in a tree. In the first case, main memory cost is the main memory required for the root aggregate view and its child aggregate views in the tree, which is formulated as follows.

$$\begin{aligned} M_1(j) &= \min\{s_m c^k, (s_i + s_m + s_p) f c^k d(k)\} \\ &+ \sum_{i=0}^{j} \min\{s_m c^{k-1-i} n^i, \\ &\quad (s_i + s_m + s_p) f c^{k-1-i} n^i d(k-1)\} \end{aligned}$$

In the other case, main memory cost is the main memory required for the aggregate views obtained by recursively choosing a child aggregate view next to the rightmost child aggregate view that has the same parent, which is is formulated as follows.

$$\begin{aligned} M_2(j) &= \sum_{i=0}^{\lceil j/2 \rceil} \min\{s_m c^{k-1-i} n^{j-i}, \\ &\quad (s_i + s_m + s_p) f c^{k-1-i} n^{j-i} d(j-i)\} \end{aligned}$$

Consequently, the total main memory cost required for datacube computation is as follows.

$$M(j) = \max\{M_1(j), M_2(j)\}$$

4 Experimental Results

We performed analytical experiments based on the cost model formulated in the previous section. We used parameter values as shown in the Table 1 in these experiments. Chunk

dimension size	100
page size	4KB
size of measure values	4B
size of index values	4B
size of pointer	4B
fudge factor	1.1

Table 1: Parameters

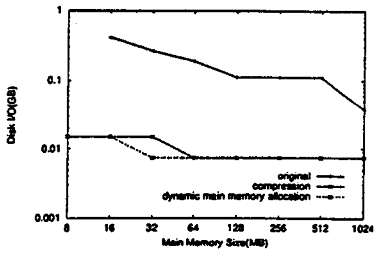

Figure 6: Disk I/O vs. Main Memory Size(5 dimensions, 10^6 tuples)

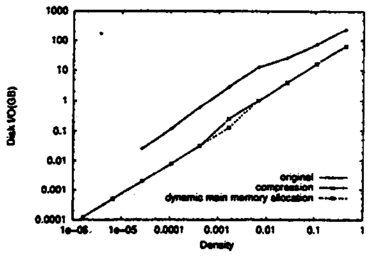

Figure 7: Disk I/O vs. Density(5 dimensions, main memory 256MB)

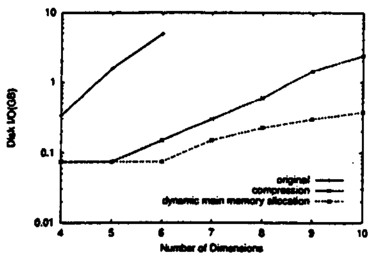

Figure 8: Disk I/O vs. Number of Dimensions(5 dimensions, 10^7 tuples, main memory 256MB)

dimension size is determined such that the chunk size for the root aggregate view is equal to the page size. Performance evaluation is based on a comparison of disk I/O. We assume that the performance is bounded by the disk I/O. The input data is assumed to be in the form of tuples in relational databases. Note that results in these experiments do not include disk I/O for the output results.

4.1 Array-Based algorithm vs. Improved Array-Based algorithm

We evaluate the performance of the improved array-based algorithm compared with the original array-based algorithm in our first experiment. The disk I/O of the loading process is not included in these results. For comparison, we assume that no intermediate results are created for the original

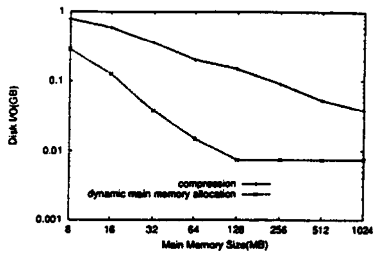

Figure 9: Disk I/O vs. Main Memory Size (10 dimensions, 10^6 tuples)

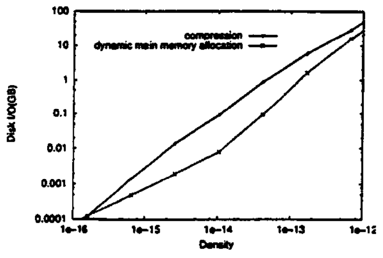

Figure 10: Disk I/O vs. Density(10 dimensions, main memory 256MB)

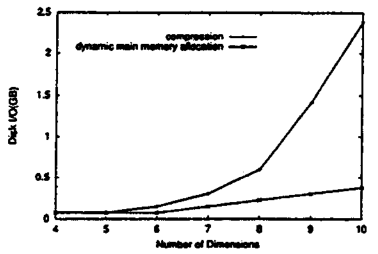

Figure 11: Disk I/O vs. Number of Dimensions(10 dimensions, 10^7 tuples, main memory 256MB)

array-based algorithm as well as the improved array-based algorithm.

Figure 6, 7 and 8 show the amount of disk I/O, varying main memory size, density and the number of dimensions respectively. Since the original array-based algorithm assumes low dimensional datacubes, we used 5 as the number of dimensions of a datacube for these experiments. However, since this number of dimensions is too low to observe the effect of dynamic main memory allocation, we used 10 dimensions for comparison between hash-based compression and dynamic main memory allocation, which are shown in Figure 9, 10 and 11. We can see from these figures that the improved array-based algorithm shows better performance when we apply hash-based compression to arrays in main memory. Furthermore, we can see that the performance of the array-based algorithm is further improved by dynamic main memory allocation. There are some cases where the results of the original array-based algorithm are not shown in these graphs, because in these cases, even the root aggregate view does not fit into main memory unless the number of elements in a chunk is made smaller, which is less efficient in disk I/O.

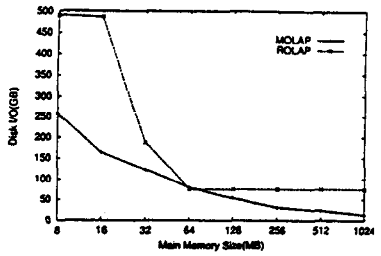

Figure 12: Disk I/O vs. Main Memory Size(10 dimensions, 10^8 tuples)

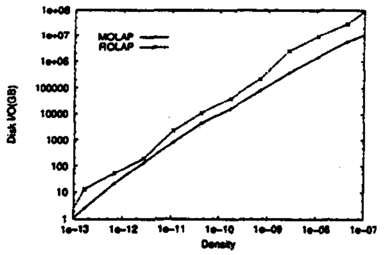

Figure 13: Disk I/O vs. Density(10 dimensions, main memory 256MB)

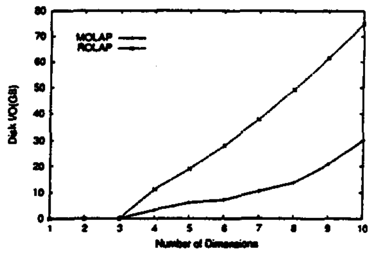

Figure 14: Disk I/O vs. Number of Dimensions(10 dimensions, 10^8 tuples, main memory 256MB)

4.2 MOLAP vs. ROLAP

In the following experiments, we compared the MOLAP algorithm and the ROLAP algorithm. We used the improved array-based algorithm, including the loading process as the MOLAP approach. For the ROLAP approach, the sort-based algorithm proposed in [RS97] was used. Since the sort-based algorithm assumes the higher dimensions than the original array-based algorithm, we computed a 10 dimensional datacube in these experiments.

Figure 12, 13 and 14 show the amount of disk I/O as a function of main memory size, density and the number of dimensions respectively. We can observe that the performance of the MOLAP algorithm dominates the ROLAP algorithm for various cases.

The primary feature of the ROLAP algorithm proposed by [RS97] is that the disk I/O cost is a linear function of the number of dimensions. On the other hand, the disk I/O cost of the array-based algorithm, even with the improved main memory efficiency, is not likely to be linear with respect to the number of dimensions. However, for the dimension size range and other parameters in this experiment, the performance of the MOLAP algorithm is much better than that of the ROLAP algorithm.

5 Related Work

Several researches on datacube computation have already been discussed. In this section, we discuss work not already cited earlier. In [AAD+96, SAG96], a hash-based algorithm as well as a sort-based algorithm is proposed for the ROLAP approach. Hashing as well as sorting is a commonly used technique for the aggregate operation in relational databases. An attempt is made in the research to apply these methods to the computation of datacubes. On the other hand, our algorithm uses a hashing method as a compression technique for an array structure in the MOLAP approach. The most significant difference between these two methods is that when data is partitioned using an appropriate number of dimensions, the hash-based algorithm does not compute aggregate views not containing the partitioning attributes. We believe that this difference affects the performance significantly.

6 Conclusions and Future Work

We have presented an algorithm for computing datacubes from the MOLAP viewpoint, which allows systems to choose data structures well suited for multidimensional aggregation. We proposed efficient main memory utilization methods for the array-based algorithm. Experimental results based on our cost model show that the MOLAP algorithm using our methods performs better than the ROLAP algorithm in many cases.

However, our work is still in the preliminary stages. Our performance evaluation is merely based on an analytical cost model, in which we made simplifying assumptions that data is uniformly distributed and all dimensions have the same size. In the near future, we plan to actually implement our algorithm and evaluate its performance using various data distributions and different dimension sizes.

References

[AAD+96] S. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishman and S. Sarawagi, "On the Computation of Multidimentional Aggregates", In *Proceedings of the International Conference on Very Large Databases*, pages 506–521, 1996.

[DANR96] P. M. Deshpande, S. Agarwal, J. F. Naughton and R. Ramakrishman, "Computation of Multidimensional Aggregates", Technical Report 1314, University of Wisconsin, Madison, 1996.

[GBLP96] J. Gray, A. Bosworth, A. Layman and H. Pirahesh, "A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals", In *Proceedings of the IEEE International Conference on Data Engineering*, pages 152–159, 1996.

[RS97] K. A. Ross and D. Srivastava, "Fast Computation of Sparse Datacubes", In *Proceedings of the International Conference on Very Large Databases*, pages 116–125, 1997.

[SAG96] S. Sarawagi, R. Agrawal and A. Gupta, "On Computing the Data Cube", Research Report RJ10026, IBM Almaden Research Center, San Jose, CA, 1996.

[ZDN97] Y. Zhao, P. M. Deshpande and J. F. Naughton, "An Array-Based Algorithm for Simultaneous Multidimensional Aggregates", In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 159–170, 1997.