A Dynamic Load Balancing Strategy for Parallel Datacube Computation

Seigo Muto
Institute of Industrial Science,
University of Tokyo
7-22-1 Roppongi, Minato-ku,
Tokyo, 106-8558 Japan
+81-3-3402-6231 ext. 2357
muto@tkl.iis.u-tokyo.ac.jp

Masaru Kitsuregawa
Institute of Industrial Science,
University of Tokyo
7-22-1 Roppongi, Minato-ku,
Tokyo, 106-8558 Japan
+81-3-3402-6231 ext. 2357
kitsure@tkl.iis.u-tokyo.ac.jp

ABSTRACT

In recent years, OLAP technologies have become one of the important applications in the database industry. In particular, the datacube operation proposed in [5] receives strong attention among researchers as a fundamental research topic in the OLAP technologies. The datacube operation requires computation of aggregations on all possible combinations of each dimension attribute. As the number of dimensions increases, it becomes very expensive to compute datacubes, because the required computation cost grows exponentially with the increase of dimensions. Parallelization is very important factor for fast datacube computation. However, we cannot obtain sufficient performance gain in the presence of data skew even if the computation is parallelized. In this paper, we present a dynamic load balancing strategy, which enables us to extract the effectiveness of parallizing datacube computation sufficiently. We perform experiments based on simulations and show that our strategy performs well.

Keywords

OLAP, datacube, dynamic load balancing, parallel processing.

1. INTRODUCTION

In recent years, OLAP technologies have become one of the important applications in the database industry. OLAP systems provide multidimensional views for users who attempt to analyze data from various viewpoints for use in decision making. In particular, the datacube operation proposed in [5] receives strong attention among researchers as a fundamental research topic in the OLAP technologies. The datacube operation requires computation of aggregations on all possible combinations of each dimension attribute. As the number of dimensions increases, it becomes very expensive to compute datacubes, because the required computation cost grows exponentially with the increase of dimensions. To overcome this problem, several algorithms for computing datacubes as efficiently as possible have been developed[1][2][3][10][11][15]. They studied the optimization techniques considering overlaps of computations among aggregations, utilization of memory space and efficiency of disk I/O. However, in spite of these optimizations, the target of these techniques is only for uniprocessor systems, while multiprocessor systems are widely used today for database applications. Parallelization is very important factor for fast datacube computation and there are parallel algorithms developed for further performance improvement[6]. Simple parallelization of datacube computation is not so difficult, however, we cannot obtain sufficient performance gain in the presence of data skew. In this paper, we present a dynamic load balancing strategy, which enables us to extract the effectiveness of parallizing datacube computation sufficiently. In this strategy, we detect the load status of each processor and dynamically transfer the data between processors to balance the load during processing. We perform experiments based on simulations and show that our strategy performs well.

This paper is organized as follows. In section 2, we introduce the datacube operation and computation algorithms for this operation. In section 3, a parallelization method of datacube computation is described. Section 4 explains load balancing strategies and section 5 presents experimental results. Related work is presented in section 6. Section 7 concludes with future work.

2. DATACUBE COMPUTATION

2.1 Datacube Operation

[5] generalized the group-by of SQL language in relational databases and proposed the datacube operation, which performs aggregation operations on all possible combinations of each dimension attribute. For example, we assume tuples with 4 attributes named A, B, C and D. The first 3 attributes are assumed to be used for dimensions in a multidimensional space and the last one is assumed to be used as a value to be aggregated, which is usually called a *measure* value. In this case, we can compute a 3 dimensional datacube in which aggregations are performed using dimension combinations such as ABC, AB, AC, AD, A, B, C and all, which represents a value obtained by aggregating measure values of all tuples. Each view generated by these aggregations is often called *cuboid*.

As mentioned in [5], aggregate functions can be classified into three groups named *distributive* functions, *algebraic* functions and *holistic* functions. The feature of distributive and algebraic

functions is that we can use a divide and conquer strategy to compute these functions. Functions `count`, `sum`, `max`, `min` and `average` can be classified to this group. We assume that aggregate fucntions to be used are always distributive or algebraic because the most of frequently used aggregate functions belong to this group. Because of this property of these fucntions, there are dependencies among cuboids where a cuboid can be computed from another cuboid which is already computed. For example, AB, AC and BC can be computed from ABC. These dependencies are often represented by tree structures introduced in [8] as shown in Figure 1. [5] also mentioned the optimization technique called the *smallest parent*. Aggregation costs are minimized if we compute a cuboid from the smallest parent cuboid from which the child cuboid can be generated. This optimization technique is incorporated into the most of algorithms proposed for processing datacube operations.

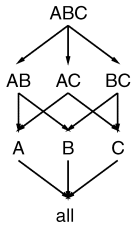

Figure 1: Dependency Tree

2.2 Computation Algorithms

Algorithms for computing datacubes can be categorized into two groups called ROLAP and MOLAP from the difference of data format to be handled directly by the systems. The ROLAP systems process tuples as data because the systems are based on relational databases, while the MOLAP systems use array structures that are more suitable for multidimensional data processing although a loading process is needed. Algorithms PipeSort, PipeHash[1][11], Overlap[1][3], PartitionedCube and MemoryCube[10] and BottomUpCube[2] are devised as the ROLAP approach. These algorithms make use of techniques sorting or hashing, which is commonly employed in implementations of normal aggregation operations in relational database systems. PipeSort, Overlap and MemoryCube are sort-based algorithms. The basic idea is to compute as many cuboids sharing the same dimension order as possible simultaneously after sorting tuples, because we can utilize the dependencies among the cuboids. Although BottomUpCube is also a sort-based algorithm, this algorithm does not use the smallest parent and computes cuboids in the opposite direction of dependencies using the results which is previously sorted for computing other cuboids. Different from other methods, PipeHash is an algorithm based on a hashing technique. For the detail of this algorithm, we describe in the next subsection. In [5], a simple algorithm using array structures is described. As the MOLAP approach, [15] developed more sophisticated array-based algorithm for computing datacubes.

2.3 PipeHash

Among the algorithms described in the previous subsection, we choose the PipeHash as an algorihtm to be parallelized. In the following, we present the detail description of this algorithm.

The PipeHash algorithm proceeds as follows. First we read raw data from disk into main memory, and apply a hash function to dimension attributes of tuples in main memory to insert the tuple into a hash table for the root cuboid. If the tuple with the same dimension attribute values is found in the hash table, the measure attribute value of the input tuple is aggregated into the measure attribute value of the tuple in the hash table. After all tuples are processed, the next child cuboids of the root cuboid are computed. Hash tables of these child cuboids are created and tuples from the hash table of the root cuboids are inserted into these hash tables by being applied a hash function to aggregate the measure attribute value. After these processings are completed, the next child cuboids are computed in the same way. In this way, processings continue until the computation of all cuboids is completed. The parent cuboid from which the cuboid is computed is determined based on the smallest parent. This is achieved by solving a minimum spanning tree problem where the cost of each node is the estimated cuboid size.

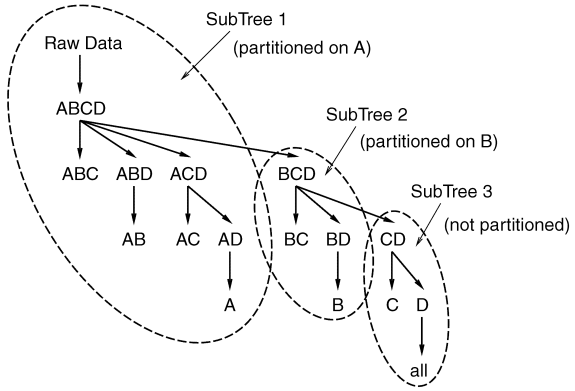

Figure 2: Subtrees in Dependency Tree

If hash tables cannot fit into main memory, cuboids have to be partitioned on some dimension attributes so that the memory space required for hash tables to be constructed from tuples in these partitions becomes smaller than the available memory space. In this case, however, only cuboids containing the partitioning dimension attributes can be computed simultaneously. For this reason, we have to divide the cuboids into some groups that can be computed at the same time. For example, we assume to compute a datacube which have 4 dimension attributes A, B, C and D. If we partition tuples on A and B, the cuboids are divided into 3 groups. Figure 2 depicts the dependency tree divided into 3 subtrees as a result of partitioning. First, since we partition raw data on A, only cuboids in the subtree 1 can be computed in this pass. In the next pass, partitioning on the attribute B is applied to ABCD and the computation of cuboids in the subtree 2 is done. Other remaining cuboids can be computed from BCD without partitioning as shown in subtree 3. When partitioning is performed, we have to determine how to divide the dependency tree into subtrees. A greedy algorithm is used in PipeHash for the decomposition of the tree.

3. PARALLELIZING DATACUBE COMPUTATION

In this section, we describe a simple parallelization algorithm for datacube computation. Although there are several kinds of architectures for multiprocessor systems, we assume that our algorithm is used on shared-nothing multiprocessors, which can provide higher scalability than the other architectures.

Figure 3 shows a parallel algorithm for PipeHash. Input to this algorithm is a dependency tree divided into subtrees based on the greedy algorithm used in the PipeHash. For the first subtree, we partition the root of this tree on some dimensions and allocate each partition to any one of processors based on the values of the partitioning dimensions. Once the partitions are sent to each processor, cuboids in this subtree are computed in a pipelined fashion using the PipeHash algorithm. This computation is performed independently among the processors on a partition by partition basis. If a cuboid corresponding to the root of the next subtree has been created from the partitions, the result is partitioned on some dimensions to distribute to each of the processors and cuboids in the next subtree are computed in the same way. Other remaining subtrees are also computed by repeating these processes.

Note that this algorithm is described in the abstract way. Although we chose the PipeHash algorithm, some other algorithms based on other techniques such as sort or array can be applied to this parallelization. This is true for load balancing strategies we will describe in the next section.

```
Algorithm ParallelPipeHash(subtrees S)
foreach subtree s in S do
   Partition the root of s on some dimensions;
   foreach partition p do
      Allocate p to the processor determined by the values of the
      partitioning dimensions;
   end
   foreach processor do
      foreach partition p do
         Compute cuboids in s from p using PipeHash;
      end
   end
end
```

Figure 3: Parallel PipeHash Algorithm

4. LOAD BALANCING STRATEGIES

If data is skewed, this simple parallelization method causes inefficient resource utilization, because the performance is bounded by the processor whose load is much higher than other processors due to data skew. To make effective use of the other processors which might be in the idle state, a load balancing mechanism should be incorporated into the parallel algorithm described above. We introduce load balancing strategies in this section. A Static approach and a dynamic approach can be considered as a load balancing method for parallel datacube computation. We describe each strategy in turn in the following subsections.

4.1 Definitions of Data Skew

Before describing each load balancing strategy, we first define the types of data skew that could arise in the parallel computation of datacubes. In [14], several kinds of data skew are classified for parallel join processing. [12] distinguishes the types of data skew in parallel aggregation. In the context of parallel datacube computation, two types of data skew are considered as follows.

Partition Skew: Skew of the number of tuples generated from the partitioning process in the parallel algorithm. This skew will occur when the number of each distinct value of partitioning dimensions is not uniformly distributed.

Cuboid Skew: Skew of the size of a cuboid created from each partition. This skew can occur due to the characteristics of data in each partition of the parent cuboid from which the cuboid can be derived. The number of tuples generated from the partition is determined by a data density of the partition and correlations of each of dimension attribute values in the partition.

4.2 A Static Load Balancing Strategy

```
Algorithm StaticLoadBalancing(subtrees S)
foreach subtree s in S do
   Partition the root of s on some dimensions;
   foreach partition p do
      Allocate p to the processor determined by the values of the
      partitioning dimensions;
   end
   let t_max = the largest number of tuples per processor;
   let t_min = the smallest number of tuples per processor;
   let p_max = a processor which has t_max;
   let p_min = a processor which has t_min;
   let d = difference between t_max and t_min;
   while d is reduced by reallocation of a partition do
      reallocate one of the partitions on p_max to p_min;
   end
   foreach processor do
      foreach partition p do
         Compute cuboids in s from p using PipeHash;
      end
   end
end
```

Figure 4: Static Load Balancing Algorithm

This subsection presents the description of a static load balancing strategy, which is provided especially for handling partition skew. The algorithm of this strategy is shown in Figure 4. The difference point from the simple parallel algorithm is in the partitioning phase. The simple partitioning method would cause load imbalances among processors in the presence of partition skew. In this algorithm, once a cuboid is partitioned on some dimensions, the part of these partitions are redistributed so that distribution of the number of tuples is as flat as possible among the processors. Since it is NP-hard to solve the optimal allocation of the partitions, we used a heuristic method, which allocates one of the partitions on the processor which has the largest number of tuples to the processor which has the smallest number of tuples.

This procedure is repeated while the difference in the number of tuples between the two processors selected for reallocation of a partition is reduced. However, if partition skew becomes higher, the load of datacube processing might concentrate on a small number of the processors even if the balancing mechanism of this algorithm is applied. To avoid this situation, we assume that the number of partitions to be created is sufficiently larger than the number of processors. This finer granularity of partitions contributes to reduce the concentrated load to few processors.

4.3 A Dynamic Load Balancing Strategy

```
Algorithm DynamicLoadBalancing(subtrees S)
foreach subtree s in S do
   Partition the root of s on some dimensions;
   foreach partition p do
      Allocate p to the processor determined by the values of the
      partitioning dimensions;
   end
   let t_max = the largest number of tuples per processor;
   let t_min = the smallest number of tuples per processor;
   let p_max = a processor which has t_max;
   let p_min = a processor which has t_min;
   let d = difference between t_max and t_min;
   while d is reduced by reallocation of a partition do
      reallocate one of the partitions on p_max to p_min;
   end
   foreach processor do
      foreach partition p do
         Start the computation of cuboids in s from p using
         PipeHash;
      end
   end
   while there are unprocessed partitions do
      if one of the processors p_dest complete the computation do
         choose the processor p_src which has the largest number
         of unprocessed partitions;
         reallocate one of the unprocessed partitions on p_src to
         p_dest;
      end
   end
end
```

Figure 5: Dynamic Load Balancing Algorithm

In the parallel algorithm described above, processing of tuples on each processor, disk I/O for reading or writing each partition and network I/O for transferring partitions can be overlapped completely. It is also possible to overlap computation of cuboids in a subtree and partitioning of a root cuboid of the next subtree. As the number of dimensions increases, the number of cuboids that should be created increases exponentially. Processor costs for executing aggregation operation for the cuboids are expected to dominate in computing datacubes. We assume that the performance is bounded by processor loads the most of which are occupied by aggregation costs. The aggregation costs depend on the number of tuples to be processed. Thus load imbalances among processors will be caused by the effect of skewed aggregation costs if there is cuboid skew. This type of skew effect cannot be avoided by the static load balancing because the sizes of cuboids to be produced cannot be predicted in advance unless we resort to any kind of estimation methods as developed in [13].

In this subsection, we introduce further enhancement to the static load balancing scheme to achieve more effective parallelization even in the presence of cuboid skew. We attempt to alleviate influence of this type of skew by dynamically transferring partitions between processors during computation. The algorithm is described in Figure 5. The partitioning method in the first phase is the same as the static approach. For the computation phase, a dynamic load balancing mechanism is incorporated.

After reallocation of partitions for evenly distributing tuples across processors, each processor starts the processing of its own allocated partitions. If one of the processors completes its processing of the partitions, the processor sends a complete signal to a master processor, which is responsible for the control of transfers of partitions between the processors. The master processor holds statistical information about how many partitions have been processed on each processor at that point. If the master processor receives the complete signal, a transfer signal with the destination information is sent to the processor on which the largest number of partitions is not processed. If the processor receives the transfer signal, the processor sends one of the unprocessed partitions to the processor which has already completed its processing. Likewise, every time one of the processors finishes the computation, a partition is dynamically migrated to the processor to prevent the waste of processor resources.

5. EXPERIMENTAL RESULTS

We made experiments based on simulations to examine the effectiveness of load balancing strategies in parallel datacube computation. For simplification, we assume that performance of the computation is bounded by processor costs except for the first partitioing process for raw data, which can be assumed to be disk I/O dominant. Aggregation costs are simply estimated by counting the number of tuples processed. Although simulations are under simplifying assumptions, it will be sufficient to compare relative performance of our strategies. Parameter values we used in these experiments are listed in Table 1. Disk I/O rate and time for aggregation are observed on IBM SP-2. When a value of a parameter is not changed in the experiment, the value in this table is used for the parameter.

Parameter	Value
number of processors	16
number of dimensions	10
number of tuples	10^8
cardinality	100
cuboid size factor	0.4165
disk read rate	6.45 (MB/s)
disk write rate	3.5 (MB/s)
aggregation time per tuple	7.89 (ms)
attribute size	4 (B)
partition size	64 (KB)

Table 1: Parameter Values

To study effects of partition skew, we varied a distribution of the number of tuples among processors in a partitioning step. We assume each partition is of the same size on each processor. For cuboid skew, we varied a distribution of aggregation costs among processors in a computation step. Likewise, each aggregation cost per partition is assumed to be equal on each processor. These distributions are varied based on a Zipf distribution, which is commonly used in experiments to investigate performance of skew handling techniques. The Zipf distribution is formulated as follows.

$$Zipf(i) = \frac{n}{i^{a} \sum_{i=1}^{p} (1/i^{a})}$$

. represents a zipf factor which indicates the degree of skew. As a value of this factor increases, the degree of skew becomes large. We assigned values from 0 to 0.5 to this factor in these experiments. p is used as the number of processors. When this distribution is used for partition skew, n represents the number of tuples of root cuboids to be partitioned in a partitioning step. For cuboid skew, n is assigned the number of tuples of child cuboids to be produced in a computation step.

We assume that all cardinalities of dimensions are equal. Suppose that the number of tuples and dimensions of raw data is t and n, and each cardinality is c, we determined the size in tuples of d dimensional cuboid $s(d)$ as follows.

$$s(d) = \min\{tf^{n-d+1}, c^{d}\}$$

In this equation, f denotes a cuboid size factor which indicates the difference in size between cuboids in different levels. We assume that the size of all cuboids in the same level are equal. The value of this factor should be determined such that the size of a child cuboid in a partition does not exceed the size of a parent cuboid of the child cuboid in that partition especially for the case of skewed data. We chose 0.4165 as a value of this factor because the largest partition size produced as a child cuboid in a computation step is equal to the partition size balanced in a partitioning step when this value is used under the condition where p is 16, . is 0.5.

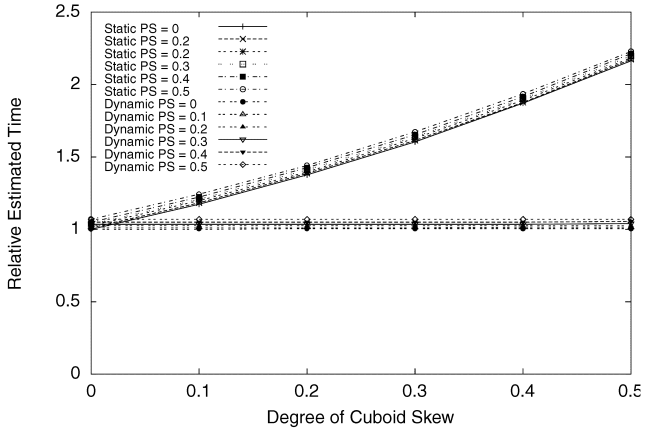

Figure 6: Effect of Data Skew

In the first experiment, we examine the impact of data skew. The results for various degrees of partition skew and cuboid skew are shown in Figure 6. The estimated execution time is normalized to observe the efficiency of processor resource utilization. Let p, t_e and t_s denote the number of processors, estimated execution time, and execution time calculated by summing the costs of all participating processors. The relative execution time t_r is formulated as follows.

$$t_r = \frac{t_e}{t_s / p}$$

If the value of t_r is closer to 1, this means that the load of processors is well balanced. For the change in the degree of partition skew, t_r is not so affected because of the balancing mechanism of both strategies in the partitioning phase, however, the performance becomes slightly worse as the degree of partition skew increases. This is because a large amount of transfers of partitions in the partitioning phase result in the increased disk I/O cost if partition skew becomes higher. For cuboid skew, the performance of the dynamic load balancing strategy is almost constant regardless of the degree of cuboid skew, while the performance of the static strategy degrades with increase in the degree of cuboid skew due to the load imbalances of the aggregation costs.

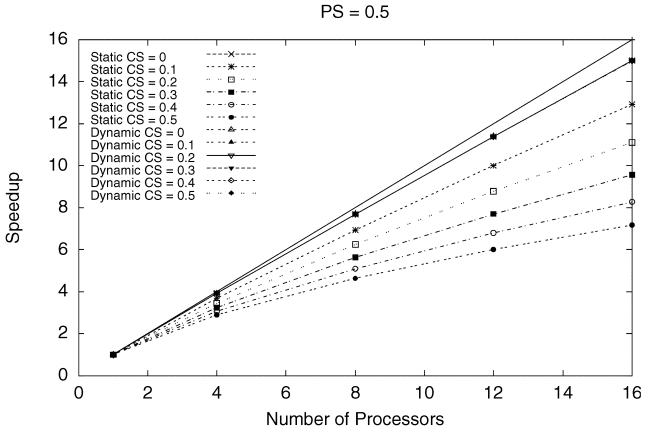

Figure 7: Speedup

In the following experiments, we fixed the degree of partition skew constantly to 1 to compare the static and dynamic load balancing algorithm for various degrees of cuboid skew. Figure 7 shows the speedup ratio to the number of processors. With the dynamic algorithm, the load balancing is effectively achieved as indicated in the results.

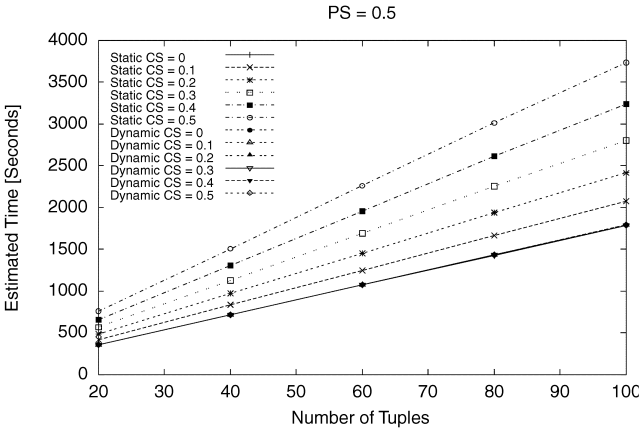

Figure 8: Performance for Various Number of Tuples

Finally, we performed experiments varying the number of tuples and dimensions as a parameter under the condition where the different degrees of cuboid skew exists. The results of these experiments are shown in Figure 8 and Figure 9. Clearly, the dynamic load balancing algorithm is not suffered from the effect of cuboid skew for the various values of these parameters, while the static algorihtm is very sensitive to cuboid skew.

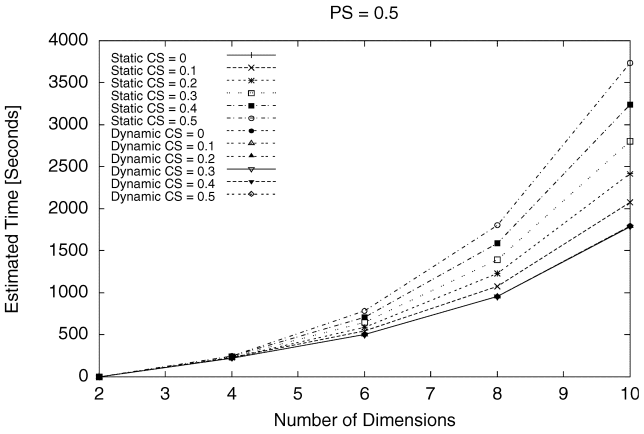

Figure 9: Performance for Various Number of Dimensions

6. RELATED WORK

There has been a considerable amont of work on dynamic load balancing on multiprocessor systems in database applications. In particular, dynamic load balancing in parallel join processing has been well studied before[4][7][9][14]. The basic idea of our method is based on these algorithms. However, to our knowledge, there has been no work on dynamic load balancing techniques for parallel datacube computation.

7. CONCLUSIONS

In this paper, we presented a dynamic load balacing strategy for parallel datacube computation in the presence of data skew. The computation of datacube can be easily parallelized, however, the effectiveness of the parallelization deteriorates in the case of skewed data. We presented the static strategy and the dynamic strategy for load balancing in the parallelized PipeHash algorithm, which makes it possible to avoid the concentration of processing load on some of the processors. We performed experiments based on simulations and it is shown that datacube computation was effectively parallelized by applying our strategy. As future work, we plan to actually implement our algorithms and evaluate the performance based on the implementation under various conditions.

8. REFERENCES

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

[2] K. S. Beyer and R. Ramakrishnan, "Bottom-Up Computation of Sparse and Iceberg CUBEs", In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 359-370, 1999.

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

[4] D. J. DeWitt, J. F. Naughton, D. A. Schneider and S. Seshadri, "Practical Skew Handling in Parallel Joins", In *Proceedings of the International Conference on Very Large Databases*, pages 27-40, 1992.

[5] 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.

[6] S. Goil and A. Choudhary, "High Performance OLAP and Data Mining on parallel computers", *Journal of Data Mining and Knowledge Discovery*, 1(4):391-417, 1997.

[7] K. A. Hua and C. Lee, "Handling Data Skew in Multiprocessor Database Computers Using Partition Tuning", In *Proceedings of the International Conference on Very Large Databases*, pages 525-535, 1991.

[8] V. Harinarayan, A. Rajaraman and J. D. Ullman, "Implementing Data Cubes Efficiently", In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 205-216, 1996.

[9] M. Kitsuregawa and Y. Ogawa, "Bucket Spreading Parallel Hash: A New, Robust, Parallel Hash Join Method for Data Skew in the Super Database Computer (SDC)", In *Proceedings of the International Conferen ce on Very Large Databases*, pages 210-221, 1990.

[10] 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.

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

[12] A. Shatdal and J. F. Naughton, “Adaptive Parallel Aggregation Algorithms”, In *Proceedings of the ACM SIGMOD Conference on Management of Data* , pages 104-114, 1995.

[13] A. Shukla, P. Deshpande, J. F. Naughton and K. Ramasamy, “Storage Estimation for Multidimensional Aggregates in the Presence of Hierarchies”, In *Proceedings of the International Conference on Very Large Databases* , pages 522-531, 1996

[14] C. B. Walton, A. G. Dale and R. M. Jenevein, “A Taxonomy and Performance Model of Data Skew Effects in Parallel Joins”, In *Proceedings of the International Conference on Very Large Databases* , pages 537-548, 1991

[15] 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.