MOLAP データキューブの並列構築

金　　東† 都司　達夫† 樋口　健†

† 福井大学大学院工学研究科 〒 910-8507 福井県福井市文京 3-9-1
E-mail: †{jindong,tsuji}@pear.fuis.fukui-u.ac.jp, ††higuchi@u-fukui.ac.jp

あらまし　データキューブの事前計算は多次元 OLAP システムの応答性を確保するために重要である。データ量の増大に対処するための性能向上の手段のひとつとして，データキューブの並列構築の方式は重要である。本論文では拡張可能配列の概念に基づいた 2 つの並列構築の方式を共有メモリ並列マシンアーキテクチャに基づいて提案する。拡張可能配列は既存要素の再配置をすることなく任意の方向に動的に拡張可能である。性能限界について定量的に評価し，これらの方式の台数効果を示す。

キーワード　並列データベース，MOLAP，データキューブ，拡張可能配列，メモリ非共有マルチプロセッサ

A New Parallel MOLAP Data Cube Construction Scheme

Dong JIN†, Tatsuo TSUJI†, and Ken HIGUCHI†

† Graduate School of Engineering, University of Fukui　Bunkyo 3–9–1, Fukui-shi, Fukui, 910–8507 Japan
E-mail: †{jindong,tsuji}@pear.fuis.fukui-u.ac.jp, ††higuchi@u-fukui.ac.jp

Abstract　The pre-computation of data cubes is critical for improving the response time of multidimensional OLAP systems. In order to meet the need for improved performance created by growing data sizes, parallel solutions for data cube construction are becoming increasingly important. This paper presents two parallel methods for data cube construction based on an extendible multidimensional array, which is dynamically extendible along any dimension without relocating any existing data. Load balancing is achieved by simple solutions on shared-memory multiprocessors. Quantitative analysis on the performance limit and parallel scalability of the methods are also given in this paper.

Key words　Parallel Database, MOLAP, Data Cube, Extendible Array, Shared-memory Multiprocessors

1. Introduction

The pre-computation of the various views (group-bys) of a data cube, i.e. the forming of aggregates for every combination of GROUP-BY attributes, is critical for improving the response time of On-Line Analytical Processing (OLAP) queries in decision support systems [4]. When the number of dimension attributes is n, the data cube computes 2^n group-bys, each of which is called a cuboid. A lattice can be used to express dependencies among cuboids [7]. Figure 1 shows a lattice for a 4-dimensional data cube with dimension a, b, c and d. An edge between two cuboids indicates that the target cuboid can be computed from the source cuboid by aggregation along one dimension. We call the cuboid *abcd* at the bottom of the lattice *base cuboid*. The others are called *dependent cuboids* because they can be computed directly or indirectly from the base cuboid.

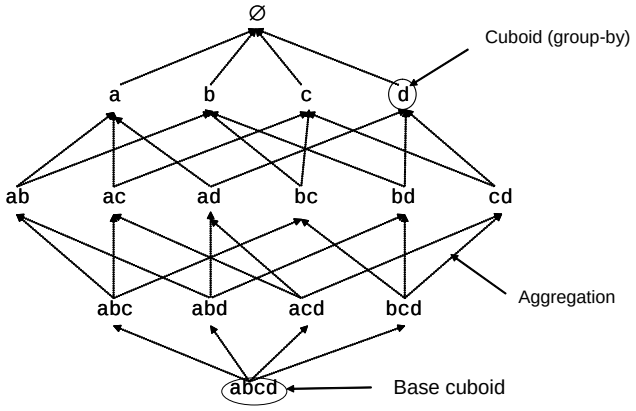


Fig. 1　A 4-dimensional data cube lattice

As the number of dimensions increases, data cube computation cost grows exponentially. Besides many methods have been proposed for data cube construction on sequential sys-

tems [6–10], parallel solutions on multiprocessor systems are becoming very popular for fast data cube computation [11–21]. They are all based on two kinds of cluster architecture: shared-storage or shared-nothing architecture depending on the nature of disks access. In this paper, we choose to implement on shared-memory multiprocessors. The processors can use shared memory to exchange data between each other to avoid the large data communication cost which may be caused by parallel data cube computation on shared-nothing multiprocessors systems.

In previous works on parallel systems [11–21], data management is one challenge because they all organize a data cube on cuboid level as far as we know. Data management becomes very complex for high dimensional data cubes because a full data cube is usually stored into $O(2^n)$ files corresponding to the 2^n cuboids. All of the parallel methods are also challenged by load imbalance which is caused by different cuboid sizes or data skew[12,19]. Due to dependency among cuboids, there must be parallel data cube construction performance limit in the previous works[11–21]. However, the performance limit was not explicitly discussed in those papers.

In this paper we overcome the challenge of complex data management by implementing a single array based data cube. The parallel data cube construction algorithms proposed in this paper achieve load balancing by simple solutions on shared-memory multiprocessors. We will also discuss the parallel performance limit quantitatively together with the related construct, parallel scalability.

There are two basic data cube representations: ROLAP representations where cuboids are represented as relational tables and MOLAP representations where cuboids are represented as multi-dimensional arrays. Multi-dimensional arrays are natural to express the multi-dimensionality of OLAP data, which makes MOLAP more suitable for data analysis. Among the parallel data cube construction papers [11–21], papers [11–13, 18–21] are for ROLAP; papers [14–17] are for MOLAP. Fixed-size multidimensional arrays are used in MOLAP papers [14–17]. In this paper, we use the extendible multidimensional array model proposed in [5] as a basis for data cube construction in MOLAP. Unlike a fixed size multidimensional array usually employed for MOLAP, the array size can be extended dynamically along any dimension without any relocation of existing data [1–3], so that the fact data from a front-end OLTP database can be dumped into the array in real time. This paper is, to our knowledge, the first using extendible arrays on parallel data cube construction.

The remainder of this paper is organized as follows. Section 2. shows how to use a single extendible array to store a full data cube and how to dump fact data into the extendible array online with data partitioning. In section 3. we present our parallel MOLAP data cube construction algorithms with discussion on load balancing, performance limit and scalability. Section 4. concludes the paper.

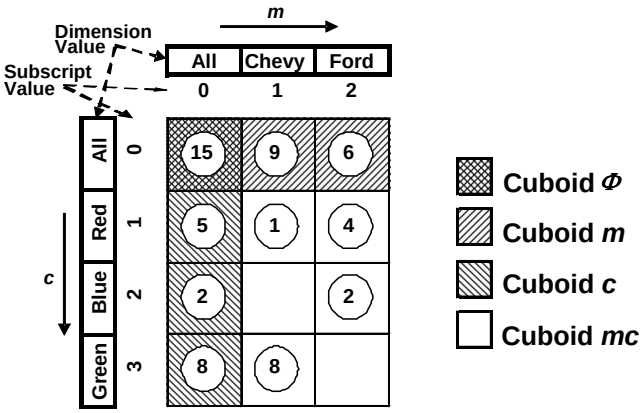
2. Extendible Array-based Data Cubing

Model	Color	Sales
Chevy	Red	1
Ford	Blue	2
Chevy	Green	8
Ford	Red	4

(a) Fact Data

Model	Color	Sales	
Chevy	Red	1	Cuboid *mc*
Ford	Blue	2	
Chevy	Green	8	
Ford	Red	4	
Chevy	ALL	9	Cuboid *m*
Ford	ALL	6	
ALL	Red	5	Cuboid *c*
ALL	Blue	2	
ALL	Green	8	
ALL	ALL	15	Cuboid *Φ*

(b) ROLAP data cube



(c) MOLAP data cube

Fig. 2 Fact data, ROLAP data cube and MOLAP data cube

Consider a 2-dimensional data cube with model(m), color(c) as dimensions and sales as a measure. $2^2 = 4$ cuboids are computed for the data cube: mc, m, c, and ϕ, where ϕ denotes the empty group-by. Obviously cuboid mc is a base cuboid, while m, c, and ϕ are dependent cuboids. An aggregate of a dimension is represented by introducing a new value ALL. Given fact data in Fig.2(a), (b) and (c) illustrate the 2-dimensional data cube represented by a relational table and a fixed size array respectively.

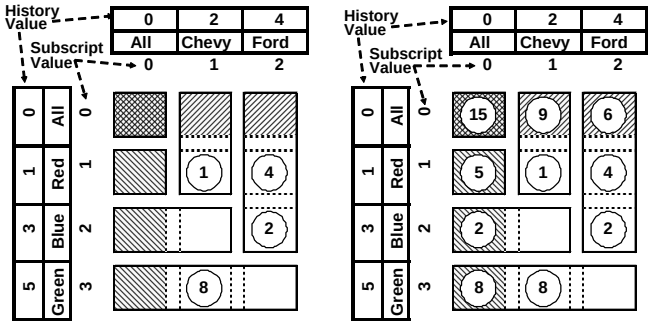
2.1 Extendible Array

In this paper we store data cubes by the extendible multidimensional array model proposed in [5]. It is based upon the index array model presented in [3]. An extendible array is dynamically extendible along any dimension without relocating any existing data. Such an extendible array consists of a set of subarrays; each subarray is allocated along some dimension d as a new distinct dimension value appears on dimension d. Figure 3 shows an extendible array based data cube given the fact data in Fig.2(a). Refer to [5] for detail on how to address an array element in an extendible array.

Element address can be computed very efficiently owing to the random access capability of multidimensional array.

As shown in Fig.3, the total number of subarrays in an extendible array based data cube is $h_{max}+1$ where h_{max} is the total of all the dimension cardinality. Each subarray is uniquely identified by a history value from 0 to h_{max} in sequence of the extension history, so we will denote a subarray which is identified by history value h simply as subarray h. As a full data cube is stored into a single extendible array, data management becomes simpler than that in the previous works as mentioned in Section 1.. Named as single-array data cube scheme, [22] describes the extendible array based data cube scheme in detail.

Note that a subarray of an extendible array based data cube generally consists of the base cuboid part and the dependent cuboid part. We call the cells in the base cuboid part as base cells, the cells in the dependent cuboid part as dependent cells. For example in Fig.3(b), subarray 4 consists of two base cells $\langle Red, Ford\rangle$ and $\langle Blue, Ford\rangle$, and one dependent cell $\langle All, Ford\rangle$.



(a) Base cuboid (b) Full data cube

Fig. 3 An extendible array based data cube

2.2 Data Partitioning On-line

As mentioned in Section 1., the fact data from a front-end OLTP database can be dumped into an extendible array in real time. For each new fact data, its dimensional values are inspected and the fact data are stored in the corresponding extendible array element. If a new dimensional value is found, the corresponding dimension of the extendible array is extended by one, and the dimensional value is mapped to the new subscript of the dimension. Consider dumping the fact data in Fig.2(a). First, the array is empty, and cell $\langle All, All\rangle$ which represents overall sales with the initial value of 0 is added into the array. Then the fact data are loaded into the array one after another to build the base cuboid mc into the array; this causes extensions of the array. Figure 3(a) shows the dumping result. We can see that the fact data dumped are naturally partitioned into corresponding subarrays.

In the previous works [14–17] in which parallel MOLAP data cube construction is based on fixed size arrays, loading and partitioning fact data into the base cuboid are necessary for data cube construction. In contrast, these steps can be saved in parallel data cube construction based on the extendible array as the base cuboid is ready in the extendible array after dumping on line. Parallel data cube construction discussed in our paper means computing the other 2^n-1 dependent cuboids from the base cuboid as input. We use the data cube in Fig.3 as the running example in the next section.

3. Parallel Data Cube Construction

In [23], subarray-based method is proposed for data cube maintenance. In this paper we apply the method on data cube construction. For detailed explanation on subarray-based method, refer to [23]. By this method, data cube construction becomes a repeated process of computing dependent cells in the 2^n-1 dependent cuboids by subarray. The method bases on a fact that base cells in a subarray usually can not determine the dependent cells in the subarray by themselves. For example, subarray 2 in Fig.3(b). The dependent cell $\langle All, Chevy\rangle$ is not only determined by internal cell $\langle Red, Chevy\rangle$, but also determined by external cell $\langle Green, Chevy\rangle$ which is out of subarray 2. Through subarray-based method, first the intermediate result for $\langle All, Chevy\rangle$ in subarray 2 is kept by aggregation with the external cell $\langle Green, Chevy\rangle$ in subarray 5; then $\langle All, Chevy\rangle$ can be computed from the intermediate result together with the internal cell $\langle Red, Chevy\rangle$. By using the intermediate result as a bridge between the subarrays, a building block of subarray processing is setup so that a data cube can be constructed in parallel grained by subarray as will be described in this section.

3.1 Subarray Processing

We describe the building block of subarray processing in Procedure 1. For a subarray h, d denotes the extended dimension of subarray h; BC represents for base cuboid part and DC for dependent cuboid part in subarray h. In the algorithm, we divide the intermediate result involved in subarray processing into two parts: IR and FIR. For a subarray h, IR denotes the intermediate result for DC; FIR denotes the intermediate result further updated in subarray h processing. IR is located in subarray h; FIR is distributed in some subarrays whose history values are less than h. See the running process of data cube construction from the base cuboid in Fig. 3(a) by repeated subarray processing from subarray $h_{max}=5$ to 0 in Table 1.

Subarray processing consists of two subroutines: *updating FIR* and *computing DC*. Note that subroutine *updating FIR* is unnecessary to be executed in subarray 0 processing.

Procedure 1: Subarray-processing

Input: BC, IR and FIR
Output: FIR and DC

(1) *Updating FIR* subroutine
Update FIR by aggregating with BC and IR along dimension d.
(2) *Computing DC* subroutine
Aggregate BC and IR to get DC, and combine DC with BC.

Table 1 Data cube construction by subarray-based method

h	*d*	*BC*	*IR*	*FIR*	*DC*
5	*c*	<Green,Chevy>:8		<All,Chevy>:0→8	<Green,All>:8
4	*m*	<Red,Ford>:4		<Red,All>:0→4	<All,Ford>:6
		<Blue,Ford>:2		<Blue,All>:0→2	
3	*c*		<Blue,All>:2	<All,All>:0→2	<Blue,All>:2
2	*m*	<Red,Chevy>:1		<Red,All>:4→5	<All,Chevy>:9
			<All,Chevy>:8	<All,All>:2→10	
1	*c*		<Red,All>:5	<All,All>:10→15	<Red,All>:5
0			<All,All>:15		<All,All>:15

Figure 4 shows the data flow diagram of subarray 2 processing in Table 1. The input includes BC $\{\langle Red, Chevy\rangle{:}1\}$, IR $\{\langle All, Chevy\rangle{:}8\}$ and FIR $\{\langle Red, All\rangle{:}4, \langle All, All\rangle{:}2\}$; the output includes DC $\{\langle All, Chevy\rangle{:}9\}$ and FIR $\{\langle Red, All\rangle{:}5, \langle All, All\rangle{:}10\}$. FIR is both in the input and output because it is updated in the subarray processing. Since the input and output of computing DC are limited in current processing subarray, computing DC can be executed in parallel by subarray based manner without confliction. Subroutine *updating FIR* is responsible for data exchange among subarrays through intermediate result. The output of updating FIR in a subarray h must be input of subarray processing for some subarrays whose history values are smaller than h. For example, the output FIR $\{\langle Red, All\rangle{:}5, \langle All, All\rangle{:}10\}$ of subarray 2 updating FIR are input IR $\{\langle Red, All\rangle{:}5\}$ and FIR $\{\langle All, All\rangle{:}10\}$ of subarray 1 processing. In other words, subarray 2 updating FIR must be prior to subarray 1 processing.

Updating FIR in various subarrays processing may access the same intermediate result. Consider FIR for subarray 2 and 4 in Table 1. They all include the intermediate result for $\langle Red, All\rangle$. It means if subarray 2 and 4 are in parallel processing, FIR accessing should be mutually exclusive.

Due to above properties in updating FIR, it should be carefully dealt with to ensure intermediate result to be updated exclusively in the reverse order of history values of subarrays. In the following, we will show two methods of parallel subarray processing. The methods are based on shared-memory multiprocessors system and all the inputs of subarray processing are shared and can be accessed by all processors.

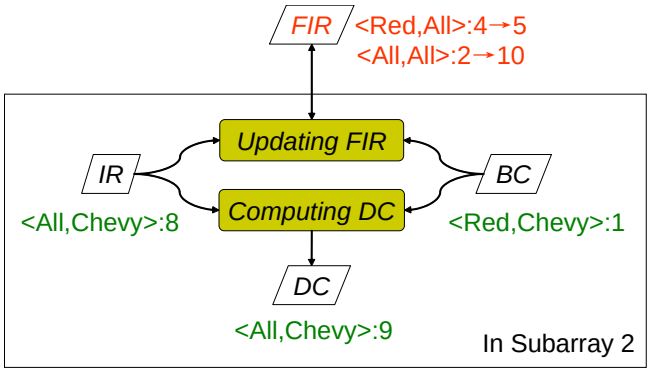


Fig. 4 Data flow diagram of subarray processing for subarray 2 in Table 1

3.2 Simple Parallel Method

In this method, all the two subroutines in the subarray processing are performed by each processor. See the algorithm executed by each processor in Procedure 2. The subarrays are allocated to the processors by a shared subarray pointer *sp*. The initial value of the pointer is h_{max}; i.e., the maximum history value. The pointer *sp* also controls *updating FIR* to be executed exclusively by subarray in sequence.

Procedure 2: Simple Parallel Algorithm

Repeat the following until no unprocessed subarrays exist.
(1) Take control of the subarray pointer *sp*. Otherwise wait.
(2) Get the current value h of *sp*.
(3) If $h > 0$, execute *updating FIR* for subarray h.
(4) Decrease *sp* by 1 and release control of *sp*.
/* From this point, other processors can take control of the subarray pointer *sp* and execute *updating FIR* */
(5) Execute *computing DC* for subarray h.

As all of the subarrays are not allocated to the processors in advance, there is no processor overload because each processor applies for a subarray to process only when it is idle. In other words, the method is free of load balancing. See the illustration of simple parallel method in Fig.5.

3.3 Layered Parallel Method

In this method, subarray processing is divided into two "layers": updating FIR layer and computing DC layer. In updating FIR layer, subroutine *updating FIR* are exclusively executed by one processor. See the algorithm in Procedure 3. In computing DC layer, the other processors execute *computing DC* in parallel. See the algorithm in Procedure 4. A shared subarray counter *sc* is used between two layers to memorize the current number of subarrays for which updating FIR is finished and computing DC is still not started. The subarray counter *sc* is initialized to zero. If the subarray counter *sc* is 0, computing DC processors wait until a new

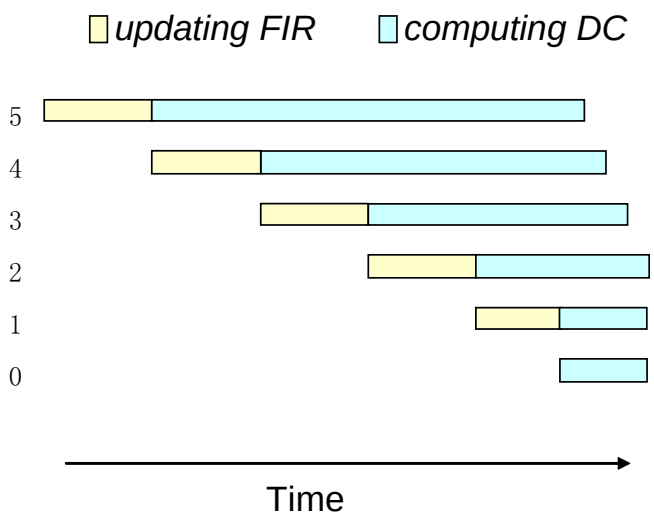


Fig. 5 Illustration of Simple Parallel Method for a data cube having 6 subarrays.

subarray is available for processing. Such an arrangement ensures the intermediate result ready to start *computing DC* for any subarray. A shared subarray pointer *sp* is still used in computing DC layer to allocate subarrays to the processors. Its initial value is h_{max}.

Procedure 3: Updating FIR layer algorithm

/* The subarray counter *sc* should be accessed exclusively */
For $h = h_{max}$ to 1 do
(1) Execute *updating FIR* for subarray h.
(2) Increase the subarray counter *sc* by 1 for subarray h.
(3) Wakeup slept processors in computing DC layer.
End For
(4) Increase *sc* by 1 for subarray 0.

Procedure 4: Computing DC layer Algorithm

/* The subarray counter *sc* and pointer *sp* should be accessed exclusively */
Repeat the following until no unprocessed subarrays exist.
(1) If the subarray counter *sc* is zero, then sleep else decrease *sc* by 1.
(2) Get the current value h of the subarray pointer *sp*.
(3) Decrease *sp* by 1.
(4) Execute *computing DC* for subarray h.

Layered Parallel Method is also free of load balancing inside each layer, but it has two kinds of load imbalance between the two layers. In the following we will analyze them and provide load balancing solutions.

1. In the case of overload in updating FIR layer and idle in computing DC layer.

In fact, this is not a load imbalance issue because updating FIR can not be speeded up by multiprocessors. In this condition, parallel processing must approach to the performance limit. There are unnecessary processors in computing DC layer. By reducing those unnecessary ones, parallel processing should still approach to the performance limit with load balance. We will discuss the performance limit later.

2. In the case of overload in computing DC layer and idle in updating FIR layer.

This means that there are many subarrays waiting to be processed in computing DC layer when updating FIR for all the subarrays are finished. We can simply solve the issue by assigning the updating FIR layer processor to execute computing DC with other processors after it finished updating FIR for all the subarrays. See the illustration of Layered Parallel Method in Fig.6. The updating FIR layer processor can execute *computing DC* for subarray 0 because updating FIR for all the subarrays are already finished.

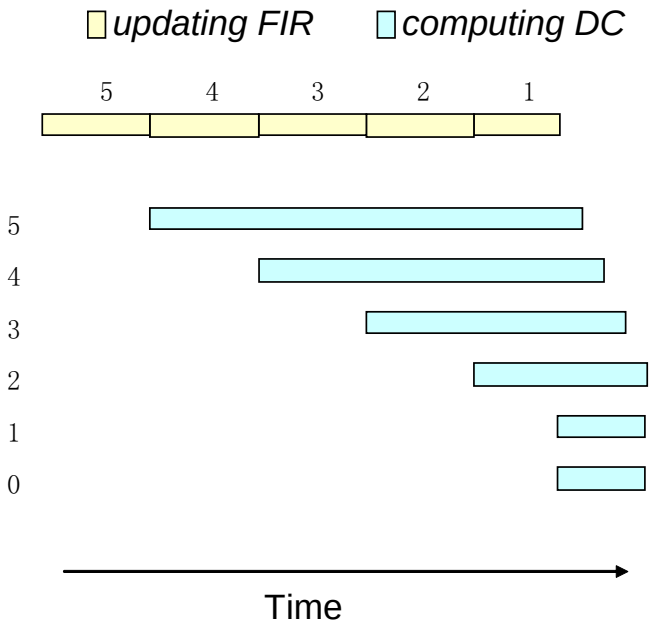


Fig. 6 Illustration of Layered Parallel Method for a data cube having 6 subarrays.

3.4 Performance Limit and Scalability

It can be proved that there must be a minimum parallel processing time in both of the above methods. Figure 5 and 6 illustrate the minimum parallel processing time of Simple Parallel Method and Layered Parallel Method respectively. We can conclude that the two methods' minimum parallel processing time should be same regardless of other performance loss. Assume the following denotations. Let T, T_{FIR} and T_{DC} denote total data cube construction time, total updating FIR time and total computing DC time in sequential

mode respectively, where $T = T_{FIR}+T_{DC}$; T_h denotes subarray h processing time and T_{min} denotes the minimum parallel processing time. Obviously if any $T_h \ll T$, T_{min} is approximate to T_{FIR}. In other words, T_{FIR} is the performance limit on parallel processing time. Correspondingly, $\psi = T/T_{FIR}$ is the performance limit represented by speedup ratio.

We can also use the ratio ψ as a scalability index of our parallel methods because data cube construction can be effectively parallelized by more processors if T is much larger relative to T_{FIR}. Obviously larger ψ indicates higher scalability. The ratio ψ also means the minimum processor resource utilization needed to reach the performance limit if there is no performance loss.

4. Conclusion

In this paper, we present and analyze subarray-based parallel MOLAP data cube construction methods based on an extendible multidimensional array. The extendible array can be dynamically extended along any dimension without relocating any existing data and enables on-line fact data loading and partitioning. The data management are simplified by storing a full data cube into an extendible array. Load balancing is achieved by simple solutions on shared-memory multiprocessors. Quantitative analysis on the performance limit and parallel scalability of the methods are also given in this paper. We have implemented and evaluated our parallel data cube construction methods on shared-memory multiprocessors. Given the performance limit, the methods achieved close to linear speedup with load balance. We also proved by the experiments that our parallel methods can be more scalable on higher dimensional data cube construction. Due to the paper length limit, the detailed experiment result can not be presented in this paper.

References

[1] Rosenberg, A.L: Allocating Storage for Extendible Arrays, *JACM, Vol.21*, pp.652–670, 1974.

[2] Rosenberg, A.L. and Stockmeyer, L.J.: , Hashing Schemes for Extendible Arrays, *JACM, Vol.24*, pp.199–221, 1977.

[3] Otoo, E.J. and Merrett,T.H.: A Storage Scheme for Extendible Arrays, *Computing, Vol.31*, pp.1–9, 1983.

[4] Gray, J., Bosworth, A., Layman, A., and Pirahesh, H.: Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals, *Proc. of the ICDE Conference*, pp.152–159, 1996.

[5] Hasan, K. M. A., Kuroda M., Azuma N., Tsuji T.: An Extendible Array Based Implementation of Relational Tables for Multidimensional Databases, *Proc. of DaWaK*, pp.233–242, 2005.

[6] Agrawal, S., Agrawal, R., Deshpande, P. M., Gupta, A., Naughton, J. F., Ramakrishnan, R. and Sarawagi, S.: On the Computation of Multidimensional Aggregates, *Proc. of the VLDB Conference*, pp.506–521, 1996.

[7] Harinarayan, V., Rajaraman, A. and Ullman J. D.: Implementing Data Cubes Efficiently, *Proc. of the ACM SIGMOD Conference*, pp.205–216, 1996.

[8] Zhao, Y., Deshpande, P. M. and Naughton, J. F. : An array based algorithm for simultaneous multidimensional aggregate, *Proc of the ACM SIGMOD Conference*, pp.159–170, 1997.

[9] K. Ross and D. Srivastava, Fast computation of sparse datacubes, *Proc. of 23rd VLDB Conference*, pp. 116–125, 1997.

[10] J. Yu and H. Lu, Multi-cube computation, *Proc. of 7th International Symposium on Database Systems for Advanced Applications*, Hong Kong, pp. 126–133, 2001.

[11] F. Dehne, T. Eavis, S. Hambrusch, and A. Rau-Chaplin, Parallelizing the data cube, *Distributed and Parallel Databases, vol. 11*, no. 2, pp. 181–201, 2002.

[12] Y. Chen, F. Dehne, T. Eavis, and A. Rau-Chaplin, Parallel ROLAP data cube construction on shared-nothing multiprocessors, *Distributed and Parallel Databases, vol. 15*, no. 3, pp. 219–236, 2004.

[13] Frank K. H. A. Dehne, Todd Eavis, and Andrew Rau-Chaplin: The cgmCUBE project: Optimizing parallel data cube generation for ROLAP, *Distributed and Parallel Databases vol. 19*, no. 1, pp.29–62, 2006.

[14] S. Goil and A. Choudhary, High performance OLAP and data mining on parallel computers, *Journal of Data Mining and Knowledge Discovery, vol. 1*, no. 4, pp. 391–417, 1997.

[15] Sanjay Goil, Alok N. Choudhary: PARSIMONY: An Infrastructure for Parallel Multidimensional Analysis and Data Mining, *J. Parallel Distrib. Comput., vol. 61*, no. 3, pp.285–321, 2001.

[16] G. Yang, R. Jin, and G. Agrawal, Implementing data cube construction using a cluster middleware: Algorithms, implementation experience, and performance evaluation, *Future Generation Computer Systems*, vol. 19, pp. 533–550, 2003.

[17] R. Jin, G. Yang, K. Vaidyanathan, and G. Agrawal, Communication and Memory Optimal Parallel Data Cube Construction, *IEEE Transactions On Parallel and Distributed Systems, Vol.16*, No. 12,pp.1105–1119, 2005.

[18] H. Lu, X. Huang, and Z. Li, Computing data cubes using massively parallel processors, *Proc. of 7th Parallel Computing Workshop (PCW¨ 97)*, Canberra, Australia, 1997

[19] S. Muto and M. Kitsuregawa, A dynamic load balancing strategy for parallel datacube computation, *Proc. of ACM Second International Workshop on Data Warehousing and OLAP (DOLAP 1999)*, pp.67–72, 1999.

[20] H. Lu, J.X. Yu, L. Feng, and X. Li, Fully dynamic partitioning: Handling data skew in parallel data cube computation, *Distributed and Parallel Databases, vol. 13*, pp.181–202, 2003.

[21] R. Ng, A. Wagner, and Y. Yin, Iceberg-cube computation with pc clusters, *Proc. of ACM SIGMOD Conference on Management of Data*, pp. 25–36, 2001.

[22] Jin, D., Tsuji, T., Tsuchida, T., Higuchi, K.: An Incremental Maintenance Scheme of Data Cubes, *Proc. of DASFAA*, pp.172–187, 2008.

[23] Dong Jin, Tatsuo Tsuji and Ken Higuchi, An Incremental Maintenance Scheme of Data Cubes and Its Evaluation, *IPSJ Online Transactions, Vol. 2*, pp.36–48, 2009.