IPSJ Transactions on Databases Vol. 1 No. 3 36–48 (Dec. 2008)

Regular Paper

An Incremental Maintenance Scheme of Data Cubes and Its Evaluation

Dong Jin,[†1] Tatsuo Tsuji[†1] and Ken Higuchi[†1]

Data cube construction is a commonly used operation in data warehouses. Since both the volume of data stored and analyzed in a data warehouse and the amount of computation involved in data cube construction are very large, incremental maintenance of data cube is really effective. In this paper, we employ an extendible multidimensional array model to maintain data cubes. Such an array enables incremental cube maintenance without relocating any data dumped at an earlier time, while computing the data cube efficiently by utilizing the fast random accessing capability of arrays. In this paper, we first present our data cube scheme and related maintenance methods, and then present the corresponding physical implementation scheme. We developed a prototype system based on the physical implementation scheme and performed evaluation experiments based on the prototype system.

1. Introduction

Analysis on large datasets is increasingly guiding business decisions. Retail chains, insurance companies, and telecommunication companies are some of the examples of organizations that have created very large datasets for their decision support systems. A system storing and managing such datasets is typically referred to as a data warehouse and the analysis performed is referred to as On Line Analytical Processing (OLAP). At the heart of all OLAP applications is the ability to simultaneously aggregate across many sets of dimensions. Jim Gray has proposed the cube operator for data cube [7]. Data cube provides users with aggregated results that are group-bys for all possible combinations of dimension attributes. When the number of dimension attributes is n, the data cube computes 2^n group-bys, each of which is called a cuboid.

As the computation of a data cube typically incurs a considerable query processing cost, it is usually precomputed and stored as materialized views in data warehouses. A data cube needs updating when the corresponding source relation changes. We can reflect changes in the source relation to the data cube by either recomputation or incremental maintenance. Here, the incremental maintenance of a data cube means the propagation of changes to the data cube. When the amount of changes during the specified time period are much smaller than the size of the source relation, computing only the changes of the source relation and reflecting to the original data cube is usually much cheaper than recomputing from scratch. Consequently, several methods that allow the incremental maintenance of a data cube have been proposed in the past. The most recent one we are aware of is Ref. 8). However, until now these methods are all for relational model, i.e., there seems no satisfactory papers for MOLAP (Multidimensional OLAP) as far as we know.

In MOLAP systems, a snapshot of a relational table in a front-end OLTP database is taken and dumped into a fixed size multidimensional array periodically, for example, every week or month. At every dumping, a new empty fixed size array has to be prepared and the relational table is dumped again from scratch. If the array dumped previously is intended to be used, all of the elements in it must be relocated by using the corresponding address function of the new empty array, incurring a huge cost.

In this paper, we use the extendible multidimensional array model described in Ref. 20) as a basis for incremental data cube maintenance in MOLAP. The array size can be extended dynamically in any direction during execution time [1),2),4)]. When a dynamic array is newly allocated when required at the execution time, all the existing elements of an extendible array are used as they are without any relocation; only the extended part is dynamically allocated. For each record inserted after the latest dumping, its column values are inspected and the fact data are stored in the corresponding extendible array element. If a new column value is found, the corresponding dimension of the extendible array is extended by one, and the column value is mapped to the new subscript of the dimension. Thus incremental dumping is sufficient instead of entirely dumping a relational table.

To maintain a data cube incrementally, existing methods compute a *delta cube*,

†1 Graduate School of Engineering, University of Fukui

representing the changes to the original data cube. The incremental maintenance of a data cube is divided into two stages: *propagate* and *refresh*[6]. The *propagate stage* computes the change of a data cube from the changes of the source relation, i.e., constructing delta cube. Then, the *refresh stage* refreshes the original data cube by applying the computed change (delta cube) to it. In this paper, we address a number of data structure and algorithm issues for efficient incremental data cube maintenance using the extendible multidimensional array. We use a single extendible array to store a full data cube, called *single-array data cube scheme*. The main contributions of this paper can be summarized as follows:

(a) To avoid huge overhead in the refresh stage, we propose *shared dimension method* for incremental data cube maintenance using the single-array data cube scheme.
(b) We propose to materialize only the *base cuboid* of delta cube, in propagate stage. Therefore the cost of the propagate stage is significantly reduced by using our method compared with the previous methods.
(c) By partitioning the data cube based on the single-array data cube scheme, we present a *subarray-based algorithm* that refreshes the original data cube by scanning the base cuboid of the delta cube only once with limited working memory usage.
(d) We implement our approach by a prototype system. Through experiment evaluation on the prototype system, we proved the effectiveness of our approach on incremental maintenance of data cubes.

2. Employing Extendible Array

The extendible multidimensional array used in this paper is presented in Ref. 20). It is based upon the index array model presented in Ref. 4). An n dimensional extendible array A has a *history counter* and three kinds of auxiliary table for each extendible dimension $i\,(i = 1, \ldots, n)$. **Figure 1** is an example of two dimensional extendible array. These tables are *history table* H_i, *address table* L_i, and *coefficient table* C_i. The history tables memorize extension history. If the size of A is $[s_1, s_2, \ldots, s_n]$ and the extended dimension is i, for an extension of A along dimension i, contiguous memory area that forms an $n-1$ dimensional subarray S of size $[s_1, s_2, \ldots, s_{i-1}, s_{i+1}, \ldots, s_{n-1}, s_n]$ is dynamically

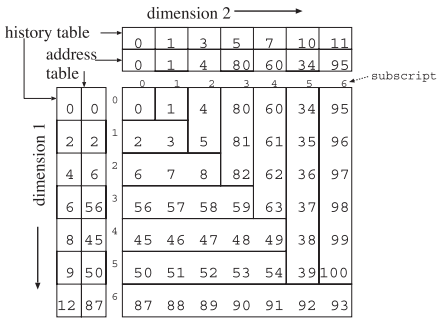

Fig. 1 A two dimensional extendible array.

allocated. Then the current history counter value is incremented by one, and it is memorized on H_i, also the first address of S is held on L_i. Since history value increases monotonously, H_i is an ordered set of history values. Note that an extended subarray is one-to-one corresponding with its history value, so the subarray is uniquely identified by its history value.

As is well known, element $(i_1, i_2, \ldots, i_{n-1})$ in an $n-1$ dimensional fixed size array of size $[s_1, s_2, \ldots, s_{n-1}]$ is allocated on memory using addressing function like:

$$f(i_1, \ldots, i_{n-1}) = s_2 s_3 \ldots s_{n-1} i_1 + s_3 s_4 \ldots s_{n-1} i_2 + \ldots + s_{n-1} i_{n-2} + i_{n-1} \quad (1)$$

We call $\langle s_2 s_3 \ldots s_{n-1}, s_3 s_4 \ldots s_{n-1}, \ldots, s_{n-1} \rangle$ a coefficient vector. If n is greater than 2, such a coefficient vector is computed at array extension and is held in a coefficient table of the corresponding dimension. Note that n is 2 in the array in Fig. 1, so the coefficient table is void. Using these three kinds of auxiliary tables, the address of array element $(i_1, i_2, \ldots, i_n)$ can be computed as follows.

(a) Compare $H_1[i_1], H_2[i_2], \ldots, H_n[i_n]$. If the largest value is $H_k[i_k]$, the subarray corresponding to the history value $H_k[i_k]$, which was extended along dimension k, is known to include the element.
(b) Using the coefficient vector memorized at $C_k[i_k]$, the offset of the element $(i_1, \ldots, i_{k-1}, i_{k+1}, \ldots, i_n)$ in the subarray is computed according to its addressing function in (1).
(c) $L_k[i_k]$+(the offset in (b)) is the address of the element.

For example, consider the element $\langle 3, 4 \rangle$ in Fig. 1. Since, $H_1[3] < H_2[4]$, it can

be known that the element is involved in the extended subarray S of history value $H_2[4] = 7$. So the first address of S is known to be $L_i[4] = 60$. Since the offset of the element $\langle 3, 4\rangle$ from the first address of S is 3, the address of the element is determined as 63. Note that we can use such a simple computational scheme to access an extendible array element only at the cost of small auxiliary tables. The superiority of this scheme is shown in Ref. 4) compared with other schemes such as hashing [2].

3. Our Approach

In our approach, we use a single multidimensional array to store a full data cube [7]. Each dimension of the data cube corresponds to a dimension of the array with the same dimensionality as the data cube. Each dimension value of a cell of the data cube is uniquely mapped to a subscript value of the array. Note that special value *All* in each dimension of the data cube is always mapped to the first subscript value 0 in each dimension of the array. For concreteness, consider a 2-dimensional data cube, in which we have the dimensions *product* (p), *store* (s) and the "measure value" (or fact data) *sales* (m). To get the cube we will compute sales grouped by all subsets of these two dimensions. That is to say, we will have *sales* by *product* and *store*; *sales* by *product*; *sales* by *store*; and overall *sales*. We can denote these *group-bys* as cuboids *ps*, *p*, *s*, and Φ, where Φ denotes the empty *group-by*. We call cuboid *ps* as *base cuboid* because other cuboids can be aggregated from it. Let **Fig. 2** (a) be the fact table of the data cube. Figure 2 (b) shows the realization of the 2-dimensional data cube using a single 2-dimensional array. Note that the dimension value tables are necessary to map the dimension values of the data cube to the corresponding array subscript values.

Obviously, we can retrieve any cuboid as needed by simply specifying corresponding array subscript values. For the above 2-dimensional data cube, see Fig. 2 (b). Cuboid *ps* can be obtained by retrieving array element set $\{(x_p, x_s)|x_p \neq 0, x_s \neq 0\}$; Cuboid p by $\{(x_p, 0)|x_p \neq 0\}$. Cuboid s by $\{(0, x_s)|x_s \neq 0\}$; Cuboid Φ by (0,0). x_p and x_s denote subscript values of dimension p, and dimension s respectively.

The data cube cells are a one-to-one correspondence to the array elements. So we may also call a data cube cell an *element of the data cube* thereafter. For example in Fig. 2 (b), cube cell $\langle$*Yplaza, Pen*$\rangle$ can be also referred to as cube element (1, 1).

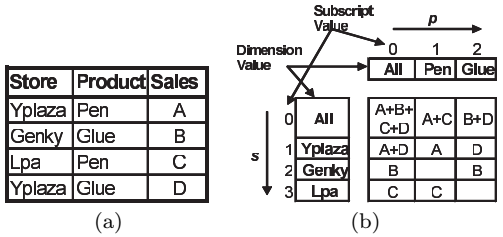

Fig. 2 A data cube using a single array.

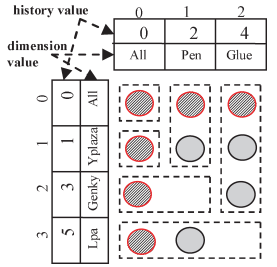

Fig. 3 Single array data cube scheme.

Now we implement the data cube with the extendible multidimensional array model presented in Section 2. Consider the example in Fig. 2 (b). First, the array is empty, and cell $\langle All, All\rangle$ which represents overall sales with the initial value 0 is added into the array. Then the fact data are loaded into the array one after another to build the base cuboid *ps* into the array; this causes extensions of the array. Then the cells in the cuboids other than the base cuboid are computed from the base cuboid *ps* and added into the array. For example, we can compute the value of $\langle$*Yplaza, All*$\rangle$ as the sum of the values of $\langle$*Yplaza, Pen*$\rangle$ and $\langle$*Yplaza, Glue*$\rangle$ in the base cuboid. Refer to the result in **Fig. 3**. To simplify the figure, the address tables and the coefficient tables of the extendible array explained in Section 2 are omitted. We call such a data cube scheme as *single-array data cube scheme*.

The cells in the cuboids that are other than the base cuboid are called *dependent*

cells[16] because these cells can be computed from the cells of the base cuboid. For the same reason, we call the cuboids other than the base cuboid *dependent cuboids*. Obviously, any dependent cell has at least one dimension value "*All*". Therefore in our single-array data cube scheme any array element having at least one zero subscript value is a dependent cell. Note that a subarray generally consists of base cuboid cell (s) and dependent cell (s). For example in Fig. 3, the subarray with history value 4 consists of two base cuboid cells $\langle$*Yplaza, Glue*$\rangle$ and $\langle$*Genky, Glue*$\rangle$, and one dependent cell $\langle$*All, Glue*$\rangle$.

In the following we use the single-array data cube scheme to maintain a data cube incrementally. The aggregate functions used in the data cube maintenance need to be distributive[7]. For simplicity, we only focus on the `SUM` function in this paper. In addition, we assume that the change of the corresponding source relation involves only insertion. However, our approach can be easily extended to handle deletions and updates using the techniques provided in Ref. 6).

3.1 Shared Dimension Method

As we described in Section 1, the incremental maintenance of a data cube consists of the *propagate stage* and the *refresh stage*. The propagate stage computes the change of a data cube from the change of the source relation. Then, the refresh stage refreshes the data cube by applying the computed change to it. Let ΔF denote a set of newly inserted tuples into a fact table F. The propagate stage computes ΔQ which denotes the change of a data cube Q from ΔF. Take the 2-dimensional data cube Q in the above as an example, ΔQ can be computed using the following query:

```
SELECT p, s, SUM(m)
FROM ΔF
CUBE BY p, s
```

We call ΔQ a delta cube. A delta cube represents the change of a data cube. The definition of ΔQ is almost the same as Q except that it is defined over ΔF instead of F. In this example, ΔQ computes four cuboids as Q. We call a cuboid in a delta cube as a *delta cuboid*, and denote delta cuboids in ΔQ as Δps, Δp, Δs and $\Delta\Phi$ which represent the change of cuboid ps, p, s and Φ in the original data cube Q respectively.

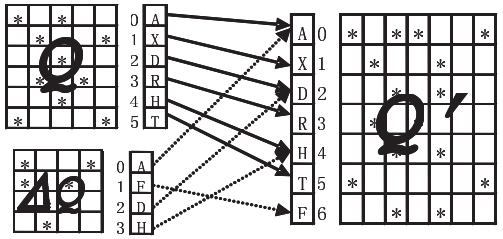

Fig. 4 Non-shared dimension method.

We can implement original data cube Q and delta data cube ΔQ as distinct extendible arrays. As ΔF is usually much smaller than F, the dimension sizes of the extendible array for ΔQ are supposed to be smaller than that for the original data cube Q. For example, the original data cube Q has six distinct values in a dimension, while the delta cube ΔQ has four distinct values in the dimension. See **Fig. 4**. They all have fewer distinct values than the dimension of the updated data cube Q' which has seven distinct values (a new dimension value 'F' is appended from ΔQ). In such a method, we can keep the size of the array for ΔQ as small as possible, but we need to keep another dimension value table for ΔQ. Thus the same dimension value may have different subscript values between the arrays. Assume the first subscript value is 0. The dimension value 'H' in ΔQ has a different subscript value with the one in Q and Q': 3 in ΔQ, 4 in Q and Q'. Therefore during the refresh stage, each dimension value table should be checked to get the corresponding array elements updated. This will lead to huge overhead for large datasets.

To avoid such a huge overhead, our approach uses the same dimension table for the original data cube Q and delta cube ΔQ. For example in Fig. 4, only the dimension value table for Q' will be used. So in the refresh stage, the dimension value tables need not to be checked because the corresponding array elements have the same subscript values in every array. We call such a method *shared dimension method*.

To apply the shared dimension method into the extendible array model, the original and delta data cubes physically share one set of dimension value tables,

Inputs:
- The changes of source relation (fact table) ΔF;
- Shared dimension data (*dimension value tables*, *history tables,* and *coefficient tables*);

Outputs:
- The delta data cube ΔQ with only delta base cuboid materialized;
- Updated shared dimension data;

Method:
- Initialize delta data cube ΔQ with the current shared dimension data;

For each tuple *tp* in ΔF **do**
  For each dimension value *v* in *tp* **do**
    If *v* is not found in the corresponding shared dimension value table **then**
      - update shared dimension data accordingly;
      - allocate new subarray by extending along the dimension by one;
    End if
  End for
  - update the related base cuboid cell of ΔQ by aggregating fact data of *tp*;
End for

Fig. 5 Delta base cuboid building with shared dimension data updating.

history tables, and coefficient tables, while each data cube has independent set of subarrays to store fact data. The shared dimension data (dimension value tables, history tables, and coefficient tables) are updated together with building the *delta base cuboid* caused by source relation change ΔF. The algorithm for building a delta base cuboid with shared dimension data updating is shown in **Fig. 5**.

For example, let the original fact table F consist of the first two tuples in Fig. 2 (a), and the change of the source relation ΔF consist of the rest two tuples. **Figure 6** (a) shows the original data cube Q, Fig. 6 (b) shows the resulted delta base cuboid in delta cube ΔQ along with the updated shared dimension data. Note that the dimension data in Fig. 6 (a) and (b) are physically in the same storage and are shared between Q and ΔQ. In other words, updating of the shared dimension data in the delta base cuboid (Fig. 6 (b)) is reflected in the shared dimension data in the original data cube (Fig. 6 (a)).

3.2 Subarray-Based Method

Our approach materializes only one delta cuboid, namely base cuboid in the

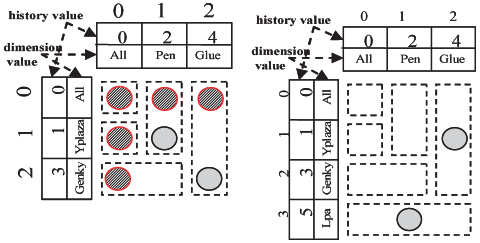

Fig. 6 Original data cube and delta data cube with shared dimension data.

propagate stage by executing the algorithm in Fig. 5. Therefore the cost of the propagate stage is significantly reduced. During the refresh stage, the delta base cuboid is scanned by subarray to refresh the corresponding subarray in the original data cube in one pass. This is why we name such a method as *subarray-based method*.

3.2.1 Partitioning of a Data Cube

In the single-array data cube scheme, a subarray is allocated for each distinct value of a dimension. Since a subarray is one-to-one corresponding with its history value as we noted in Section 2, a distinct dimension value is also one-to-one corresponding with the history value of its subarray. So, we can call the history value corresponding to a distinct dimension value v as the history value of v. Let $e = (v_1, v_2, v_3)$ be any base cuboid element in a 3-dimensional data cube, so $v_1, v_2, v_3 \neq 0$. We denote h_i as the history value of v_i $(i = 1, 2, 3)$. Without loss of generality, we assume the history value h_i satisfies $0 < h_1 < h_2 < h_3$. Note that hereafter history value 0 will be often denoted as h_0 for clarity. According to the semantics of the `CUBE` operator, there are $2^3 - 1 = 7$ dependent cells of e in the data cube. **Table 1** shows the list of the base cuboid element e and its 7 dependent elements implemented by our data cube scheme.

In Table 1, eight elements are partitioned into four groups according to the subarrays to which they belong. Each group is one-to-one corresponding with the history value of its corresponding subarray. Therefore we call the group corresponding to history value h as group h. So $(v_1, v_2, v_3), (0, v_2, v_3), (v_1, 0, v_3)$, and $(0, 0, v_3)$ is in group h_3, $(v_1, v_2, 0)$ and $(0, v_2, 0)$ is in group h_2, $(v_1, 0, 0)$ is in

Table 1 A base cuboid element and its dependent elements in a 3-dimensional data cube.

related element	subarray history (group)	can be aggregated with
$(\mathbf{v_1}, \mathbf{v_2}, \mathbf{v_3})$	h_3	-
$(0, v_2, v_3)$	h_3	$(\mathbf{v_1}, \mathbf{v_2}, \mathbf{v_3})$
$(v_1, 0, v_3)$	h_3	$(\mathbf{v_1}, \mathbf{v_2}, \mathbf{v_3})$
$(0, 0, v_3)$	h_3	$(\mathbf{v_1}, \mathbf{v_2}, \mathbf{v_3})$
$(\mathbf{v_1}, \mathbf{v_2}, \mathbf{0})$	h_2	$(\mathbf{v_1}, \mathbf{v_2}, \mathbf{v_3})$
$(0, v_2, 0)$	h_2	$(\mathbf{v_1}, \mathbf{v_2}, \mathbf{0})$
$(\mathbf{v_1}, \mathbf{0}, \mathbf{0})$	h_1	$(\mathbf{v_1}, \mathbf{v_2}, \mathbf{0})$
$(\mathbf{0}, \mathbf{0}, \mathbf{0})$	h_0	$(\mathbf{v_1}, \mathbf{0}, \mathbf{0})$

group h_1, and (0, 0, 0) is in group h_0. In each group there is one and only one element with which the rest elements can be aggregated. Such element will be called the base element of the group. Obviously $(v_1, v_2, v_3), (v_1, v_2, 0), (v_1, 0, 0)$, and (0, 0, 0) are the base elements of the group h_3, h_2, h_1, and h_0 respectively. They are in bold typeface in Table 1. Furthermore, the base element of group h_{i-1} can be aggregated with the base element of group h_i along the extended dimension $i\,(i = 1, 2, 3)$. That is to say, $(v_1, v_2, 0)$ is the base element of group h_2 and can be aggregated with (v_1, v_2, v_3), the *base element* of group h_3 along the extended dimension 3; $(v_1, 0, 0)$ is the base element of group h_1 and can be aggregated with $(v_1, v_2, 0)$; (0, 0, 0) is the base element of group h_0 and can be aggregated with $(v_1, 0, 0)$.

From the above 3-dimensional data cube, we can generalize our data cube scheme for an n-dimensional data cube as follows. For any base cuboid element e in an n-dimensional data cube, there are $2^n - 1$ dependent cells of e. We can get these cells by substituting 0 for the n coordinate terms of e. The obtained 2^n elements (including e itself) can be partitioned into $n + 1$ groups according to their history values of the subarrays to which they belong. So group h_i is the group of cells corresponding to history value $h_i\,(i = 0, \ldots, n)$, where $0 = h_0 < h_1 < h_2 < \ldots < h_n$. We can arrange the dimension values of e as $(v_1, v_2, \ldots, v_n)$, where the corresponding history value of v_i is $h_i\,(i = 1, \ldots, n)$. Then, all the elements in group h_i can be presented as n dimensional coordinate $(*, \ldots, *, v_i, 0, \ldots, 0)$ where * represents either $v_j\,(j = 1, \ldots, i-1)$ or 0. Element $(v_1, v_2, \ldots, v_i, 0, \ldots, 0)$ is the base element of group h_i, and the rest elements in the group can be aggregated with it. There are total of 2^{i-1} elements in group $h_i\,(i > 0)$. In group h_0 there is always only one element, $(0, 0, \ldots, 0)$. Furthermore, the base element of group h_{i-1} can be aggregated with the base element of group h_i along the extended dimension $i\,(i = 1, \ldots, n)$. We will show later that we need to keep the intermediate result for the base element of group h_{i-1} by aggregating it with the base element of group h_i along the extended dimension i.

3.2.2 Refreshing Scheme

As our subarray-based method only materializes the delta base cuboid in the propagate stage, for each element in the delta base cuboid we need to update the corresponding 2^n elements of the original cube during the refresh stage. We can separate the updating of the 2^n elements into $n+1$ groups. The elements in group h_n are refreshed together with the base cuboid element as they are in the same subarray. For the elements in the other n groups, we keep the intermediate results for the base elements of the groups until we refresh the corresponding subarrays whose history values are $h_i\,(i = 0, \ldots, n-1)$. As we mentioned, the intermediate result for the base element of group h_{i-1} can be aggregated with the base element of group h_i along the extended dimension of $i\,(i = 1, \ldots, n)$. See the example in Table 1; for any element $e = (v_1, v_2, v_3)$ in the subarray of the delta data cube, its $2^3 = 8$ corresponding elements are updated in the subarrays corresponding to h_3, h_2, h_1, and h_0 of the original data cube. For the refreshment of the subarray corresponding to h_3, update $(v_1, v_2, v_3), (0, v_2, v_3), (v_1, 0, v_3)$, and $(0, 0, v_3)$ and keep the intermediate result for $(v_1, v_2, 0)$ by aggregating (v_1, v_2, v_3) along dimension 3; for the refreshment of the subarray corresponding to h_2, update $(v_1, v_2, 0)$ and $(0, v_2, 0)$ and keep the intermediate result for $(v_1, 0, 0)$ by aggregating $(v_1, v_2, 0)$ along dimension 2; for the refreshment of the subarray corresponding to h_1, update $(v_1, 0, 0)$ and keep the intermediate result for (0, 0, 0) by aggregating $(v_1, 0, 0)$ along dimension 1; finally for the refreshment of the subarray corresponding to h_0 only update (0, 0, 0) without keeping further intermediate result.

To describe generally, for all the delta base cuboid elements of a subarray ΔS in the delta cube and all the intermediate result T for ΔS, our refreshing scheme based on the subarray-based method performs two things to refresh the corresponding subarray S in the original data cube; one is to update S with T and

Inputs:
- The delta data cube ΔQ generated from the algorithm in Fig. 6;
- The original data cube Q;

Output:
- The updated data cube Q;

Method:

For each history value h of a subarray in ΔQ from the max history value to 1 **do**
- let d = the extended dimension on history value h;
- **If** the intermediate result array T_d does not exist **then**
  - create n-1 dimensional array T_d in memory;
- **End if**
- update T_d by aggregating with $S[h]$ in ΔQ along dimension d;
- Let T_d'$[h]$ = all the elements in T_d' whose corresponding history value is h;
- update T_d by aggregating with T_d'$[h]$ along dimension d;
- refresh $S[h]$ in Q by aggregating with T_d'$[h]$ and $S[h]$ in ΔQ;

End for
- Refresh $S[0]$ in Q by aggregating with intermediate result array for $S[0]$

Fig. 7 Refreshing algorithm for subarray-based method.

all the base cuboid elements in ΔS; the other is to keep the further intermediate results by aggregating with T and all the base cuboid elements in ΔS along the extended dimension of ΔS. See the detail of our subarray-based refreshing algorithm in **Fig. 7** and the example running result in **Table 2**.

In the algorithm shown in Fig. 7, $S[h]$ denotes the subarray whose corresponding history value is h. As the intermediate result data are always aggregated along the subarray extended dimension, we can hold the intermediate result data for dependent cells in n arrays of $n-1$ dimensionality. We denote such data as T_d which holds the intermediate result by aggregating it with the subarrays extended on dimension d. T'_d denotes all the intermediate result arrays except T_d. Note that the intermediate results for the dependent cells in a subarray with history value h always come from the subarrays whose corresponding history values are larger than h. Thus, to get all the intermediate results, we must start refreshing from the subarray with the maximum history value.

Table 2 Result of the refresh algorithm against ΔF in Fig. 2 (a).

history value	extended dimension	aggregated elements in ΔQ & intermediate result	updated intermediate result	subarray elements refreshed
5	***s***	<Lpa,Pen,C>	<*All*,Pen,C> in *Ts*	<Lpa,Pen,C>, <Lpa,*All*,C>
			<*All*,Glue,0> in *Ts*	
4	***p***	<Yplaza,Glue,D>	<Yplaza,*All*,D> in *Tp*	<Yplaza,Glue,D>, <Genky,Glue,B>, <*All*,Glue,B+D>
		<Genky,Glue,B>	<Genky,*All*,B> in *Tp*	
		<*All*,Glue,0> in *Ts*	<*All*,*All*,0> in *Tp*	
3	***s***	<Genky,All,B> in *Tp*	<*All*,*All*,B>	<Genky,*All*,B>
2	***p***	<Yplaza,Pen,A>	<Yplaza,*All*,A+D>	<Yplaza,Pen,A>, <*All*,Pen,A+C>
		<*All*,Pen,C> in *Ts*	<*All*,*All*,B+C>	
1	***s***	<Yplaza,*All*,A+D> in *Tp*	<*All*,*All*,A+B+C+D>	<Yplaza,*All*,A+D>
0		<*All*,*All*,A+B+C+D> in *Tp*		<*All*,*All*,A+B+C+D>

Assume the example in Fig. 2 (a) as ΔF. For simplicity, we assume the original data cube Q is empty. See the running result in Table 2. The intermediate result array T_s is generated on history value 5 and T_p on history value 4. T_s consists of the intermediate result for $\langle All, Pen\rangle$ and $\langle All, Glue\rangle$; T_p consists of the intermediate result for $\langle Yplaza, All\rangle$, $\langle Genky, All\rangle$, and $\langle All, All\rangle$. As there are only two intermediate result arrays T_s and T_p in this example, T_s is equivalent to T'_p and T_p equivalent to T'_s.

In order to avoid frequent accesses to the disks, the intermediate result arrays must be kept in main memory. If array extension is in round-robin manner for all dimensions just like the one shown in Fig. 3, it can be known that the total memory requirement for the intermediate result arrays is

$$M = \sum_{i=1}^{n} \left(\prod_{j=1, j\neq i}^{n} C_i \right),$$

where C_i is the cardinality of the i-th dimension in the base cuboid ($1 \leq i \leq n$). Obviously M is much smaller than the size of the base cuboid if the dimension cardinalities are large enough.

It can be further proved that the total storage requirement in any array extension manner is bounded by M. In the data cube maintenance for a real-world dataset, it is common that the valid elements of the delta cube are not uniformly

distributed in the base cuboid. So the actual memory requirement can be much smaller than M. Furthermore, we can refine our subarray-based algorithm to deallocate the memory for those intermediate results which are not needed in later computation.

4. Physical Implementation

In practice, it is common that most multidimensional arrays for data cube are large but sparse. Multidimensional arrays are good containers to store dense data, but for sparse data cubes huge memory will be wasted because a large number of array cells are empty and thus are very hard to use in actual implementation. In particular, the sparseness problem becomes serious for delta data cube whose logical array size can be the same as that of the original data cube, but usually much more sparse than it.

HOMD (History Offset implementation scheme for Multidimensional Datasets) model presented in Ref. 20) seems to be one of the efficient storage schemes to store such sparse data. It employs extendible multidimensional arrays as its underlying logical data structure. In this section, we will extend the HOMD model for physically implementing our single-array data cube scheme and shared dimension method.

4.1 HOMD Implementation Model

The HOMD model is based on the extendible array explained in Section 2. Each dimension of a data cube corresponds to a dimension of the extendible array and each dimension value of the data cube is uniquely mapped to a subscript value of the array dimension. A subarray is constructed for each distinct dimension value. **Figure 8** shows the HOMD implementation of the two dimensional data cube in Fig. 3. For an n-dimensional data cube Q, the corresponding logical structure of HOMD is the pair (M, A). A is an n dimensional extendible array created for Q and M is a set of mappings which were mentioned in Section 3 as *dimension value tables*. Each m_i in M maps i-th dimension values of Q to subscript values of the dimension i of A. A will be often called as a logical extendible array.

Each element of an n dimensional extendible array can be specified by its n dimensional coordinate. In HOMD model, we have directed our attention to that each element can be specified by using the pair of history value and

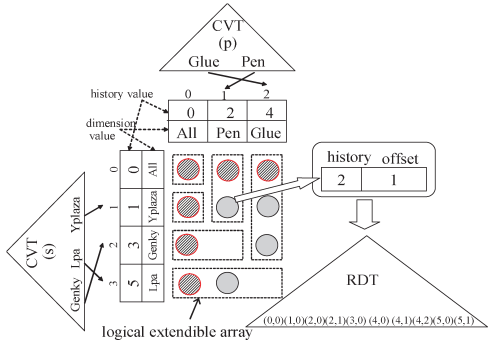

Fig. 8 Physical structure for a data cube implemented by HOMD.

offset value. Note that since each history value h is unique and has a one-to-one correspondence with its corresponding subarray S, S is specified uniquely by h. Moreover, the offset value of each element in S can be computed as in Section 2 and this is also unique in the subarray. Therefore each element of an n dimensional extendible array can be referenced by the pair (*history value, offsetvalue*). In the HOMD logical structure (M, A), each mapping m_i in M is implemented using a single B^+ tree called CVT (key subscript ConVersion Tree), and the logical extendible array A is implemented using a single B^+ tree called RDT (Real Data Tree) and n HOMD tables.

Definition 1 CVT: CVT_k for the k-th dimension of an n dimensional data cube is defined as a structure of B^+ tree with each distinct dimension value v as a key value and its associated data value is subscript i of the k-th dimension of the logical extendible array A. So the entry in the sequence set of the B^+ tree is the pair (v, i). i references to the corresponding entry of the HOMD table in the next definition.

Note that the special value *All* is unnecessary to be mapped in CVT as it is always the first subscript value 0 in each array dimension.

Definition 2 HT: HT (HOMD Table) includes three kinds of sub-table for each dimension; the history table and the coefficient table corresponding to the ones in an extendible array described in Section 2, and the column value table. Note that the address table of an extendible array in Section 2 can be void in

our HOMD physical implementation.

Elements in HT are arranged according to the insertion order. For example, the column value "Genky" is mapped to the subscript 2 in the insertion order, though in the sequence set of CVT, the key "Genky" is in position 1 due to the property of B^+ tree. At insertion of a record, each column value in it is searched in the corresponding CVT. If a key value in the record does not exist, the logical extendible array A is extended by one along the dimension, and a new slot in HT is assigned and initialized.

Definition 3 RDT: The set of the pairs (*history value, offset value*) for all of the effective elements in the extendible array are housed as the keys in a B^+ tree called RDT. Therefore, the entry of the sequence set of the RDT is the pair ((*history value, offset value*), m), here, m denotes the *measure value* for fact data in a data cube.

Note that the RDT together with the HTs implements the logical extendible array on the physical storage. We assume that a key (history value, offset value) occupies fixed size storage and the history value is arranged in front of the offset value. Hence the keys are arranged in the order of their history values and keys that have the same history value are arranged consecutively in the sequence set of RDT. Note also that since the RDT stores only the keys corresponding to the existing multidimensional data, it is highly compressed and does not contain empty array elements.

We implement our data cube scheme by HOMD aiming compression of data cubes while preserving the random accessing capability of multidimensional arrays. For an n dimensional data cube in our data cube scheme, its HOMD implementation is the set of n CVTs, n HTs and RDT. For our shared dimension method, HOMD implementation of original data cube Q and delta cube ΔQ share one set of n CVTs and n HTs, while each data cube has independent RDT to store fact data; RDT for Q and ΔRDT for ΔQ.

4.2 Refreshing of Data Cubes

We implement subarray-based method described in Section 3.2 by HOMD. We will use a B^+ tree called IRT (Intermediate Result Tree) in main memory to contain the intermediate result for the dependent cells instead of n intermediate result arrays. The data structure of IRT is the same as RDT. Note that we can easily delete those intermediate results which are not needed in later computation by IRT. In **Fig. 9**, we describe the physical refreshing algorithm for subarray-based method corresponding to the logical one described in Fig. 7. As we extend HOMD with our shared dimension methond, note that the algorithm shown in Fig. 9 is not CVT related.

```
Inputs:
  - ΔRDT for the delta cube ΔQ
        (only include base cuboid);
  - RDT for the original data cube Q ;
Output:
  - The updated RDT for data cube Q ;
Method:
  - Initialize IRT on main memory to be empty;
    For each history value h from max history value
      in ΔRDT to 1 do
     - let d = the extended dimension on history
           value h;
      For each element e with history value h in IRT
       - insert or update corresponding element e_d
           in IRT by aggregating with value of e
           along dimension d;
      End for
      - load all the elements with history value h in
           ΔRDT into main memory;
      For each element e with history value h in
           ΔRDT  do
        - insert or update corresponding element e_d
             in IRT by aggregating with value of e
             along dimension d;
      End for
       - get all the dependent cells of S[h] in ΔQ by
             aggregating with all the elements with
             history h  In ΔRDT & IRT;
       - delete all the elements with history value h
             in IRT;
       - Refresh RDT with the result calculated for
             S[h] in  ΔQ in main memory;
    End for
  - Refresh (0,0) in RDT by aggregating with (0,0)
      in IRT
```

Fig. 9 Physical refreshing algorithm for subarray-based method.

5. Performance Evaluation

In this section, we present the results of performance experiments. In Ref. 22), analytical performance evaluation is presented based on the number of tuple accesses in constructing a data cube. Here, in order to do the evaluation using an

actual system, we developed a prototype system based on HOMD implementation scheme described in the previous section.

In the experiments, we compared our subarray-based method with an incremental cube maintenance method that uses all of 2^n delta cuboids. The performance of a maintenance method is measured by the time taken for maintaining a data cube by that method. The prototype system runs on a Sun Blade 1000 with 750 MHz UltraSparc III CPU and 512 MB RAM. We implemented the two maintenance methods used in the experiments in our prototype system. We also constructed another prototype system for fixed-size sparse array data cube recomputation based on the same B^+ tree library and compared the results of the two incremental maintenance methods.

5.1 Datasets

The synthetic datasets used in the experiments were generated mainly by the following parameters:

n: Number of dimensions of a data cube

C_i: Number of dimension values (cardinality) of the i-th dimension in a base cuboid ($1 \leq i \leq n$).

ρ: Density of valid elements in an original base cuboid, it reflects the sparseness of the cuboid

Table 3 shows the parameters of three original data cubes, i.e., Q_1, Q_2 and Q_3, used in the experiments. Each cube has a different number of dimension attributes. **Table 4** shows the parameters of three datasets for new data cubes Q'_1, Q'_2 and Q'_3 corresponding to the original data cubes in Table 3. Note that each dimension in the delta data cubes has 10% new dimension values than corresponding dimension in the original data cubes, and it will cause the array to be extended.

Table 3 Original cubes used in the experiments.

cube	n	cardinality of dimensions
Q_1	3	1000*500*200
Q_2	4	100*100*100*100
Q_3	5	100*50*50*20*20

Table 4 New cubes generated in the experiments.

cube	n	cardinality of dimensions
Q'_1	3	1100*550*220
Q'_2	4	110*110*110*110
Q'_3	5	110*55*55*22*22

The records in the synthetic fact tables are uniformly distributed in random. The fact tables have no index attached. In the experiments, we varied the size of the changes to the fact table from 2% to 20% of its original size. We also varied the size of the fact table from about 600,000 tuples to 3,000,000 tuples. We made changes to the fact table by inserting new tuples to the fact table.

5.2 Experimental Evaluation

Figure 10 shows the result of performance experiment when we varied the size of changes, for a fixed size (about 600,000 tuples) of the fact tables. Note that ρ is also fixed at about 0.6% in this case. We compared our subarray-based method SB with the conventional method CV that uses all of 2^n delta cuboids. The same experiment was performed on Q_1, Q_2 and Q_3. In Fig. 10, SB (Total) and CV (Total) represent the time taken for the whole process including *propagate* and *refresh stage* in our SB method and CV method, respectively. Note that reading cost from the source relation and updating cost to refresh the original cube are included in the total cost shown in Fig. 10. In the analytical evaluation part of Ref. 22), they are mentioned but excluded from the cost comparison because they are same both for CV and SB method. As shown, the total maintenance time is reduced in our SB method because it computes only a delta base cuboid. Also, SB (Refresh) and CV (Refresh) represent the time taken for the refresh stage in our SB method and CV method respectively. Note that the refresh time is also slightly reduced in our SB method though it takes much more CPU computing cost and IRT processing cost than that of CV method. This is because only the number of tuples in the delta base cuboid need to be read from disk in our SB method. Thus, we can confirm that our SB method does not increase the refresh cost. Especially, we note that the benefit of our SB method increases as the number of dimension attributes increases. This is because the number of delta cuboids computed in CV method increases as the number of dimension attributes increases.

Figure 11 shows the two methods' storage costs by the total number of tuples in delta cuboids that are actually generated in the experiment in Fig. 10. As

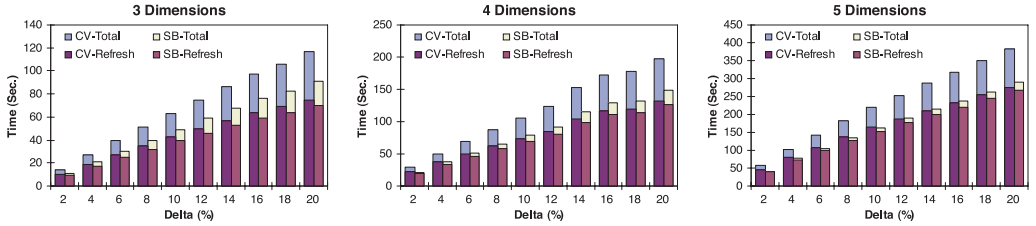

Fig. 10 Performance by varying the size of delta fact tables.

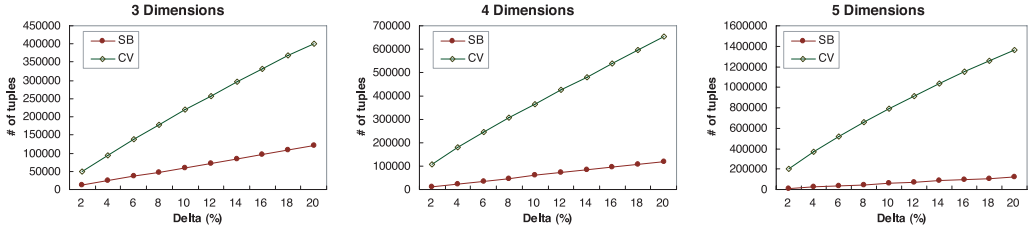

Fig. 11 Number of tuples in the delta cubes generated in the experiment in Fig. 10.

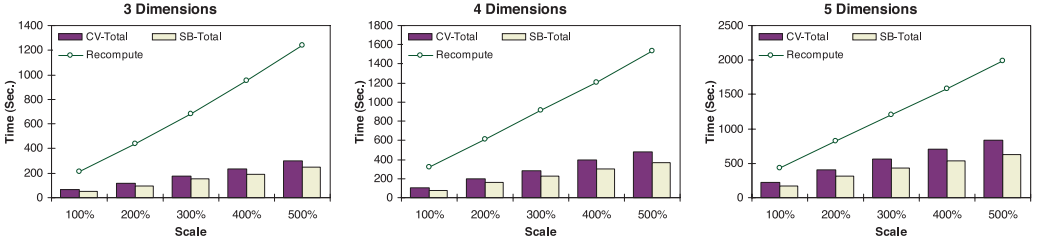

Fig. 12 Performance by varying the size of original fact tables.

shown, SB method has the storage cost advantage over CV method as the former only materializes a single delta base cuboid in propagate stage. The number of tuples generated also affects the performance and disk I/O overhead of a maintenance method. As shown in Fig. 11, the number of tuples generated by our SB method is always less than that by CV method. From Fig. 10 and Fig. 11, we can see that the performance of a maintenance method improves as the number of tuples generated decreases.

Figure 12 shows the result of the performance evaluation when we scaled the size of the fact table from 100% (about 600,000 tuples) to 500% (about 3,000,000 tuples) for a fixed rate (10%) of the change. Note that ρ is also varied accordingly from about 0.6% to 3.0%. Again in this case, our SB method outperforms the CV method for Q_1, Q_2, and Q_3. Furthermore, we can see that both CV and SB methods are much more effective than cube recomputation using fixed-size array. Hence, we can confirm that both SB and CV method can be effectively used for fast incremental maintenance of data cubes.

6. Related Work

Since Jim Gray proposed the data cube operator, techniques for data cube construction and maintenance have been extensively studied. As far as we know, these papers include Refs. 8), 12), 13), 16), 18) mainly optimize the computation on cuboid level. So they usually implement an n-dimensional data cube into 2^n nested relations or arrays corresponding to the 2^n cuboids on data organization for relational and multidimensional databases. In this paper, we propose to organize all the 2^n cuboids of a data cube into only a single extendible array. Doing so provides opportunities to simplify the data cube management.

Reference 6) is the first paper that addressed the issue of efficiently maintaining a data cube in a data warehouse. Reference 11) proposed the cubetree as a storage abstraction of a data cube for efficient bulk increment updates. The problem of maintaining data cubes under dimension updates was discussed in Ref. 3). Reference 17) presented techniques for maintaining data cubes in the IBM DB2/UDB database system. Reference 5) made some improvement based on Ref. 6). All these methods build 2^n delta cuboids to maintain a full cube with 2^n cuboids. Recently, Ref. 8) proposes an incremental maintenance method for data cubes that can maintain a full cube by building only ${}_nC_{n/2}$ delta cuboids. In comparison, our subarray-based method only builds a single delta cuboid - base cuboid to maintain a full cube.

References 3), 5), 6), 8), 11), 17) are all for ROLAP. There seems no satisfactory papers for MOLAP (Multidimensional OLAP) as far as we know. But, the method shown in Ref. 6) can also be effectively implemented in MOLAP as *CV* method we showed in Section 5. In MOLAP papers [9),10),14),15)], the notion of a data cube is different from the terminology in our paper. In fact, data cubes defined in these papers are the cuboids generated by CUBE operator in our paper. Therefore, they actually addressed cuboid maintenance to improve range query performance instead of the data cube maintenance in our context. So, they are completely different from our work.

This paper is an extended version of the one presented in Ref. 22). Reference 22) does not provide evaluation based on acutually implemented system, instead gives an analytical evaluation based on the number of tuples to be accessed during propagate stage and refresh stage. The works presented in Refs. 20), 21) are based on HOMD, on which our implementation is also based on, but data cubing is not discussed in these works. Reference 19) presents another storage scheme of multi-dimensional database by employing extendible array of high density, but it doesn't discuss anything about data cube operations, either.

7. Conclusion

In this paper we presented data structure and algorithm for data cube incremental maintenance based on the notion of an extendible array. By using the single-array data cube scheme, we developed shared dimension method and subarray-based algorithm to implement data cube incremental maintenance efficiently. Through performance experiment on a prototype system our approach shows effectiveness on fast incremental maintenance of data cubes.

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) Hurtado, C.A., Mendelzon, A.O. and Vaisman, A.A.: Maintaining Data Cubes under Dimension Updates, *Proc. ICDE Conference*, pp.346–355 (1999).
4) Otoo, E.J. and Merrett, T.H.: A Storage Scheme for Extendible Arrays, *Computing*, Vol.31, pp.1–9 (1983).
5) Li, H., Huang, H. and Lin, Y.: DSD: Maintain Data Cubes More Efficiently, *Fundam. Inform*, Vol.59, No.2-3, pp.173–190 (2004).
6) Mumick, I.S., Quass, D. and Mumick, B.S.: Maintenance of Data Cubes and Summary Tables in a Warehouse, *Proc. ACM SIGMOD Conference*, pp.100–111 (1997).
7) Gray, J., Bosworth, A., Layman, A. and Pirahesh, H.: Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals, *Proc. ICDE Conference*, pp.152–159 (1996).
8) Lee, K.Y. and Kim, M.H.: Efficient Incremental Maintenance of Data Cubes, *Proc. VLDB Conference*, pp.823–833 (2006).
9) Riedewald, M., Agrawal, D. and Abbadi, A.E.: Flexible Data Cubes for Online Aggregation, *Proc. ICDT*, pp.159–173 (2001).
10) Riedewald, M., Agrawal, D., Abbadi, A.E. and Pajarola, R.: Space-Efficient Data Cubes for Dynamic Environments, *Proc. DaWaK*, pp.24–33 (2000).
11) Roussopoulos, N., Kotidis, Y. and Roussopoulos, M.: Cubetree: Organization of

and Bulk In-cremental Updates on the Data Cube, *Proc. ACM SIGMOD Conference*, pp.89–99 (1997).
12) Jin, R. Yang, G., Vaidyanathan G. and Agrawal, K.: Communication and Memory Optimal Paral-lel Data Cube Construction, *IEEE Transactions On Parallel and Distributed Systems*, Vol.16, No.12, pp.1105–1119 (2005).
13) Agrawal, S., Agrawal, R., Deshpande, P.M., Gupta, A., Naughton, J.F., Ramakrishnan, R. and Sarawagi, S.: On the Computation of Multidimensional Aggregates, *Proc. VLDB Conference*, pp.506–521 (1996).
14) Geffner, S., Agrawal, D. and Abbadi, A.E.: The Dynamic Data Cube, *Proc. EDBT 2000*, pp.237–253 (2000).
15) Geffner, S., Riedewald, M., Agrawal, D. and Abbadi, A.E.: Data Cubes in Dynamic Environments, *IEEE Data Eng. Bull.*, Vol.22, No.4, pp.31–40 (1999).
16) Harinarayan, V., Rajaraman, A. and Ullman J.D.: Implementing Data Cubes Efficiently, *Proc. ACM SIGMOD Conference*, pp.205–216 (1996).
17) Lehner, W., Sidle, R., Pirahesh, H. and Cochrane, R.: Maintenance of Cube Automatic Sum-mary Tables, *Proc. ACM SIGMOD Conference*, pp.512–513 (2000).
18) Zhao, Y., Deshpande, P.M. and Naughton, J.F.: An array based algorithm for simultaneous multidimensional aggregate, *Proc. ACM SIGMOD Conference*, pp.159–170 (1997).
19) Otto, E.J. and Rotem, D.: A storage scheme for multi-dimensional databases using extendible array files, *Proc. STDBM*, pp.67–76 (2006).
20) Hasan, K.M.A., Kuroda M., Azuma N. and Tsuji T.: An Extendible Array Based Implementation of Relational Tables for Multidimensional Databases, *Proc. DaWaK*, pp.233–242 (2005).
21) Hasan, K.M.A., Tsuji, T. and Higuchi, K.: An Efficient Implementation for MOLAP Basic Data Structure and Its Evaluation, *Proc. DASFAA*, pp.288–299 (2007).
22) Jin, D., Tsuji, T., Tsuchida, T. and Higuchi, K.: An Incremental Maintenance Scheme of Data Cubes, *Proc. DASFAA*, pp.172–187 (2008).

(Received June 20, 2008)
(Accepted August 10, 2008)

(Editor in Charge: *Makoto Onizuka*)

Dong Jin is now a candidate of Ph.D. at Graduate School of Engineering, University of Fukui. He received his B.E. degree from Qingdao Institute of Chemical Technology, China in 1992, and MBA degree from Xi'an Jiaotong University, China in 2004. He was mainly engaged in MIS related work since graduation in 1992. His current research interests include data warehousing and management information systems.

Tatsuo Tsuji received his Ph.D. degree in Information and Computer Science from Osaka University in 1978. In the same year, he joined the Faculty of Engineering at University of Fukui in Japan. Since 1992, he has been a professor in the Information Science Department of the faculty. His current research interests include database implementation schemes and data warehousing systems. He is the author of the book, "Optimizing Schemes for Structured Programming Language Processors" published by Ellis Horwood (1990).

Ken Higuchi received his B.E., M.E. and D.E. degrees in Communications and Systems Engineering in 1992, 1994 and 1997, respectively, from the University of Electro-Communications. He is now an associate professor of the Graduate School of Engineering, University of Fukui. His research interests include database management system, parallel processing, and XML document management system.