Incremental Maintenance of Data Cubes

Dong Jin † Tatsuo Tsuji † and Ken Higuchi †

† [1]Graduate School of Engineering, University of Fukui 3-9-1 Bunkyo, Fukui-shi, 910-8507 Japan

E-mail: † {jindong, tsuji, higuchi}@pear.fuis.fukui-u.ac.jp

Abstract Data cube construction is a commonly used operation in data warehouses. Because of the volume of data stored and analyzed in a data warehouse and the amount of computation involved in data cube construction, *incremental maintenance of data cube* is really effective. To maintain a data cube incrementally, previous methods were mainly for relational databases. 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. Our data cube scheme and related maintenance methods are presented in this paper, and cost analysis on our approach is shown and compared with existing methods.

Keyword Data warehouse, Data cube, Incremental maintenance, Extendible array

1. Introduction

Analysis on large datasets is increasingly guiding business decisions. 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 only its changes. 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. Thus, several methods that allow the incremental maintenance of a data cube have been proposed in the past. The most recent one as we are aware of is [9]. But these methods are all for relational model, i.e. no papers for MOLAP (Multidimensional OLAP) until now 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 like in every week or month. At every dumping, a new empty fixed size array has to be prepared and the relational table will be dumped again from scratch. If the array dumped previously is intended to be used, all the elements in it should be relocated by using the corresponding address function of the new empty array, which also incurs huge cost.

In this paper, we use the extendible multidimensional array model proposed in [8] as a basis for incremental data cube maintenance in MOLAP. The array size can be extended dynamically in any directions during execution time [1][2][4]. While 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 wholly dumping a relational table.

To maintain a data cube incrementally, existing methods compute a *delta cube*, which represents the data cube consisting of the change of 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. We call the scheme as *single-array data cube scheme*. The main contributions of this paper can be summarized as follows.

(a) To avoid huge overhead in 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 *base cuboid* of delta cube, in propagate stage. Thus the cost of the propagate stage is significantly reduced in 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 refreshing the original data cube by scanning the base cuboid of the delta cube only once with limited working memory usage.

2. Employing Extendible Array

The extendible multidimensional array used in this paper is proposed in [8]. It is based upon the index array model presented in [4]. An n dimensional extendible array A has a history counter h and three kinds of auxiliary table for each extendible dimension i(i=1,...,n). See Fig. 1. 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 $[s_1, s_2,..., s_n]$ 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 $[s_1, s_2,...,s_{i-1}, s_{i+1},..., s_{n-1}, s_n]$ is dynamically 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 h 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 $(i_1, i_2, ..., i_{n-1})$ in an *n-1* dimensional fixed size array of size $[s_1, s_2, ..., s_{n-1}]$ is allocated on memory using addressing function like:

$$f(i_1,..., i_{n-1}) = s_2s_3 \ldots s_{n-1}i_1 + s_3s_4 \ldots s_{n-1}i_2 + \ldots + s_{n-1}i_{n-2} + i_{n-1}$$

We call $<s_2s_3...s_n, s_3s_4...s_n, ..., s_n>$ as a *coefficient vector*. Such a coefficient vector is computed at array extension and is held in a coefficient table. Using these three kinds of auxiliary tables, the address of array element $(i_1, i_2, ..., i_n)$ can be computed as follows.

(a) Compare $H_1[i_1]$, $H_2[i_2]$, ... , $H_{n-1}[i_n]$. 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 $(i_1, ..., i_{k-1}, i_{k+1}, ..., i_n)$ 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.

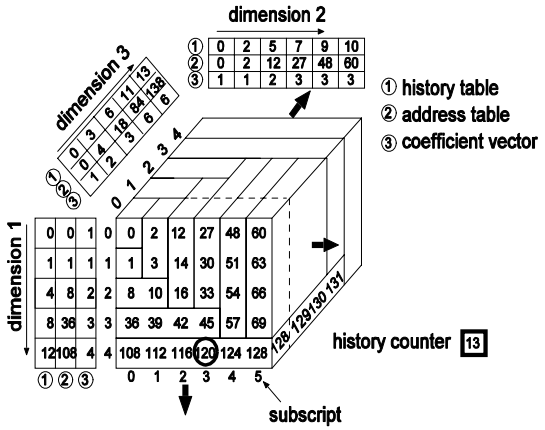


Fig. 1. A three dimensional extendible array

Consider the element (4,3,0) in Fig.1. Compare the history values $H_1[4]=12$, $H_2[3]=7$ and $H_3[0]=0$. Since $H_1[4]>H_2[3]>H_3[0]$, it can be proved that (4,3,0) is involved in the subarray *S* corresponding to the history value $H_1[4]$ in the first dimension and the first address of *S* is found out in $L_1[4]$ =108. From the corresponding coefficient vector $C_1[4]$ =<4>, the offset of (4,3,0) from the first address of *S* is computed as 4×3+0=12 according to expression (1), the address of the element is determined as 108+12=120. 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 [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" (fact data value) *sales* (*m*). To get the cube we will compute sales grouped by all subsets of these two dimensions. That is, we will have *sales* by *product* and *store*; *sales* by *product*; *sales* by *store*; and overall *sales*. We can denote these group-bys as cuboid *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. Fig. 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 got by retrieving array element set $\{(x_p,x_s)|\ x_p\neq0,x_s\neq0\}$; Cuboid *p* by $\{(x_p,0)|\ x_p\neq0\}$; Cuboid *s* by $\{(0,x_s)|\ x_s\neq0\}$; Cuboid Φ by (0,0). x_p denotes subscript value of dimension *p*, x_s denotes subscript value of dimension *s*.

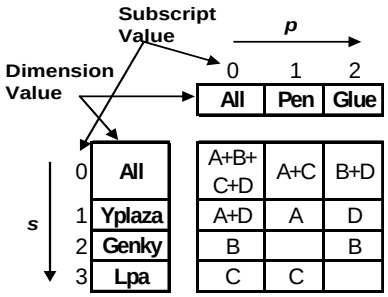
The data cube cells is one-to-one correspondence to the array elements. So we may also call a data cube cell as an element of the data cube in the following. For example in Fig. 2(b), cube cell <Yplaza, Pen> can be also referred to as cube element (1, 1).

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 <All, All> which represents overall *sales* with 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 base cuboid are computed from the base cuboid *ps* and added into the array. For example, we can compute the value of <Yplaza, All> as the sum of the values of <Yplaza, Pen> and <Yplaza, Glue> in the base cuboid. Refer to the result in Fig. 3. For simplicity of 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*.

We call the cells in the cuboids other than the base cuboid as *dependent cells* [17] because such cells can be computed from the cells of the base cuboid. For the same reason, we call the cuboids other than the base cuboid as *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 <Yplaza, Glue> and <Genky, Glue>, and one dependent cell <All, Glue>.

Store	Product	Sales
Yplaza	Pen	A
Genky	Glue	B
Lpa	Pen	C
Yplaza	Glue	D

(a) A fact table



(b) Data cube for (a)

Fig. 2. A data cube using a single array

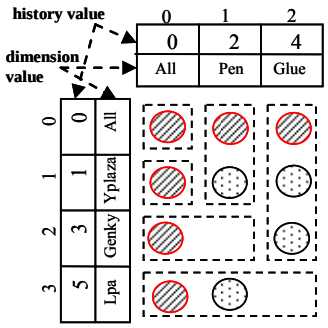


Fig. 3. Single-array data cube scheme

In the following we will 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 `SUM` function in this paper. Furthermore, 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 [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:

```
•Q : SELECT p, s, SUM(m)
     FROM •F
     CUBE BY p, s
```

We call ΔQ as 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.

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 array for ΔQ size as small as possible, but we need 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 different subscript value with the one in Q and Q': 3 in ΔQ, 4 in Q and Q'. Thus in the refresh stage, each dimension value table should be checked to get the corresponding array elements updated. It will lead to huge overhead for large datasets.

To avoid such huge overhead, our approach uses the same dimension table for original data cube Q and delta cube ΔQ. For the example in Fig. 4, it means 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 as *shared dimension method*.

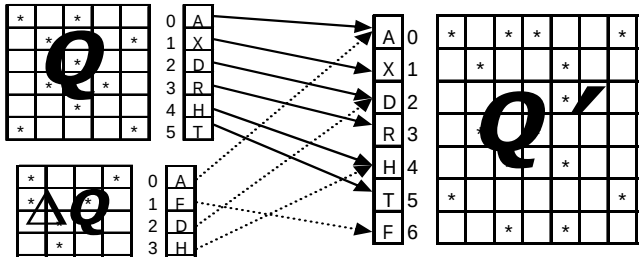


Fig. 4. Non-shared dimension method

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*, *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 of building delta base cuboid with shared dimension data updating is shown in Fig. 5.

3.2. Subarray-Based Method

Our approach materializes only one delta cuboid – base cuboid in the propagate stage by executing the algorithm in Fig. 5, so the cost of the propagate stage is significantly reduced. While in the refresh stage, the delta base cuboid is scanned by subarray to refresh the corresponding subarray in the original data cube in one pass. So we name such a method as *subarray-based method*.

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 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. We will further explain the scheme by showing the detail in our subarray-based refreshing algorithm in Fig. 6 and the example shown in Table 1.

In the algorithm shown in Fig. 6, $S[h]$ 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 *Td* which holds the intermediate result by aggregating with the subarrays extended on dimension d. *Td'* denotes all the intermediate result arrays except *Td*. 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.

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

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 1. The intermediate result array *Ts* is generated on history value 5 and *Tp* on history value 4. *Ts* consists of the intermediate result for <All, Pen> and <All, Glue>; *Tp* consists of the intermediate result for <Yplaza, All>, <Genky, All>, and <All, All>. As there are only two intermediate result arrays *Ts* and *Tp* in this example, *Ts* is equivalent to *Tp'* and *Tp* equivalent to *Ts'*.

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 in Fig. 3, it can be known that the total memory requirement for the intermediate result arrays is $M = \sum_{i=1}^{n} (\prod_{j=1, j \neq i}^{n} C_i)$, where C_i is the cardinality of the i-th dimension in 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.

Inputs:
The delta data cube ΔQ generated from the algorithm in Fig. 5;
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 *Td* does not exist **then**
Create n-1 dimensional array *Td* in memory;
End if
Update *Td* by aggregating with $S[h]$ in ΔQ along dimension d;
Let *Td'*[h] = all the elements in *Td'* whose corresponding history value is h;
Update *Td* by aggregating with *Td'*[h] along dimension d;
Refresh $S[h]$ in Q by aggregating with *Td'*[h] and $S[h]$ in ΔQ;
End for
Refresh $S[0]$ in Q by aggregating with intermediate result array for $S[0]$

Fig. 6. Refreshing algorithm for subarray-based method

It can be further proved that the total storage requirement in any array extension manner is bounded by M. In practical situation, the array extension may be not in round-robin manner and generally controllable. In the data cube maintenance for the 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

Table 1. 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>

which are not needed in later computation. All these are worth further study with experiments on real-world datasets.

4. Evaluation and Comparison

In this section, cost model for incremental data cube maintenance is developed. Based on the cost model, we make comparison among three methods: naïve method NV using all of 2^n delta cuboids, our subarray-based method SB and relational advanced method RA using ${}_nC_{n/2}$ delta cuboids [9]. Note that we only focus on the maintenance cost, but actually SB method also has the storage cost advantage over the other methods as it only materializes a single delta base cuboid in propagate stage.

4.1. Parameters

n: Number of dimensions of data cube *Q*

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

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

Nb: Total number of valid elements in the base cuboid.

Obviously, $Nb = \rho \prod_{i=1}^{n} C_i$.

Nd: Total number of valid elements in the dependent cuboids, so the total number of valid elements in *Q* is *Nb+Nd*.

4.2. Cost Model

We will use a 4-dimensional data cube for our cost evaluation and comparison. The 4 dimensions are denoted as a, b, c and d separately. Fig. 7 is the lattice diagram for the 4-dimensional data cube in analysis. We assume the values of some parameters of the data cube as follows:

n=4 ; C_i=100 (for all $1 \le i \le n$) ; ρ=0.1, 0.05, 0.01

We assume that the base cuboid is sparse which is common in practice (we can store only valid elements by some sparse array physical scheme like [8] [19]), while all the dependent cuboids are not sparse. It can be known that the total size of all the dependent cuboids *Nd* is $\prod_{i=1}^{n}(C_i+1) - \prod_{i=1}^{n} C_i$.

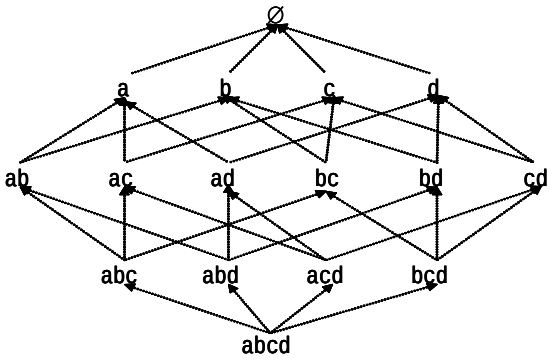


Fig. 7. Lattice diagram for a data cube

It is important for our evaluation that how many delta cuboids are materialized in these three methods. NV method materializes all the 2^n delta cuboids and get all the 2^n-*1* dependent delta cuboids using *parent method* in [14]. SB method materializes only the base delta cuboid and uses it to refresh all the 2^n cuboids of the original data cube.

RA method is a little complex since it materializes ${}_nC_{n/2}$ delta cuboids to refresh the original data cube: We can

assume the ${}_nC_{n/2}$ delta cuboids are *abcd*, *abc*, *abd*, *acd*, *bc*, *ad*, for more detail refer to [9]. Furthermore, we assume that any of the dependent cuboids are aggregated only from its parent cuboids in RA method. For example, cuboid *bc* can be only aggregated with cuboid *abc* or *bcd*.

We will denote the reading cost as *CR* and writing cost as *CW*. For simplicity, the total maintenance cost of the data cube is assumed to be the sum of *CR* and *CW*. We will use the simple linear cost model like [17] to measure *CR* and *CW*; the number of tuples read from or written to disk. The model was proved to be effective by experimental validation such as [17]. In Fig. 7 for example, if cuboid *ab* is computed from *abc*, *CR* is the size of cuboid *abc* and *CW* is the size of cuboid *ab*.

4.2.1. Propagate Cost.

In the propagate stage, reading from source relation is common to all the methods. The difference is how many delta cuboids are materialized in the delta cube. So we only compare the reading cost CR to materialize dependent cuboids and writing cost CW to materialize cuboids in this stage:

NV method: *CR*=total size of the parent cuboids of the 2^n*-1* dependent cuboids, *CW=Nb+Nd*

SB method: *CR=0, CW=Nb*

RA method: *CR*=total size of the parent cuboids of the ${}_nC_{n/2}$*-1* dependent cuboids, *CW*=total size of the ${}_nC_{n/2}$ cuboids

4.2.2. Refresh Cost.

In the refresh stage, writing cost to update the original data cube is common to all the methods. So we only compare the reading cost in this stage:

NV method: *CR= Nb+Nd*

SB method: *CR=Nb*

RA method: *CR*=total size of the ${}_nC_{n/2}$ cuboids

4.2.3. Total Cost.

We summarize all the cost in the propagate and the refresh stage except the reading cost from source relation and the writing cost to update the original data cube which are common to all the methods. We categorized the cost as reading cost *CR* and writing cost *CW*:

NV method: *CR*= total size of the parent cuboids of the 2^n*-1* dependent cuboids *+Nb+Nd*, *CW=Nb+Nd*

SB method: *CR= Nb, CW=Nb*

RA method: *CR*=total size of the parent cuboids of ${}_nC_{n/2}$*-1* dependent cuboids + total size of the ${}_nC_{n/2}$ cuboids, *CW*=total size of the ${}_nC_{n/2}$ cuboids

The total cost of the three methods are compared and shown in Fig. 8 and Fig. 9 respectively on reading and writing cost.

To summarize the evaluation and comparison, our subarray-based method shows significant advantage over other methods; as density of the data cube (base cuboid) becomes smaller, the advantage becomes even larger. Tuples in multidimensional model consist of only fact data with position information (refer to sparse array physical storage scheme such as [8] [19]), which are generally much smaller than tuples in relational model. Thus it saves much more storage cost in multidimensional model. In another word, if we abstract the cost model in terms of storage pages accessed, our approach which is based on multidimensional model will show much more advantage over the other methods in relational model.

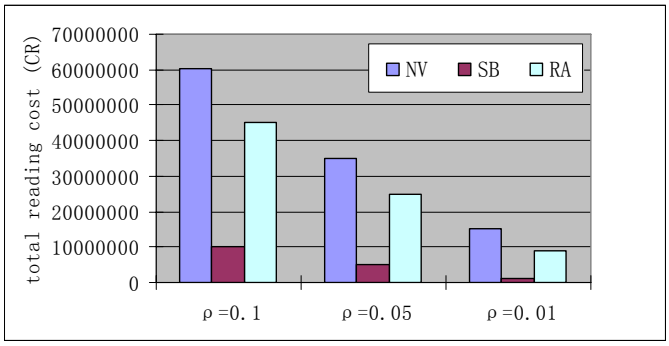


Fig. 8. Total reading cost comparison

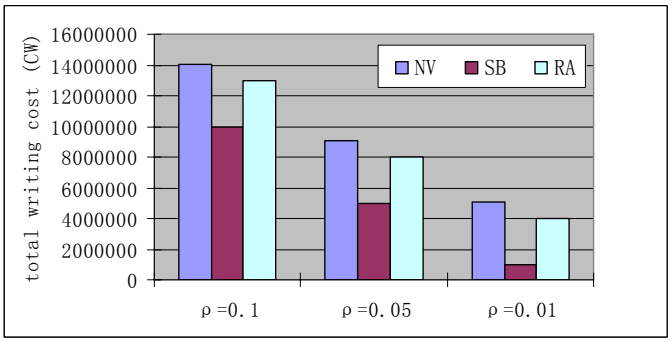


Fig. 9. Total writing cost comparison

5. Related Works

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 [9][13][14][17][19] 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, thus provide opportunities to simplify the data cube management.

[6] is the first paper that addressed the issue of efficiently maintaining a data cube in a data warehouse. [12] 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 [3]. [18] presented techniques for maintaining data cubes in the IBM DB2/UDB database system. [5] made some improvement based on [6]. All these methods build 2^n delta cuboids to maintain a full cube with 2^n cuboids. Recently, [9] propose an incremental maintenance method for data cubes that can maintain a full cube by building only ${}_nC_{n/2}$ delta cuboids. In comparison, our method only builds a single delta cuboid – base cuboid to maintain a full cube.

[3][5][6][9][12][18] are all for ROLAP, while our method is for MOLAP. In MOLAP papers [10][11][15][16], 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.

The work presented in [8] is also based on extendible arrays, but data cubing is not discussed in this work.

6. 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 cost analysis our approach shows the advantage over the state of the art.

References

[1] A.L.Rosenberg, Allocating Storage for Extendible Arrays, JACM, vol.21, pp.652-670, 1974.

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

[3] C. A. Hurtado, A. O. Mendelzon, and A. A. Vaisman, *Maintaining Data Cubes under Dimension Updates.* Proc. of the ICDE Conference, pp.346-355, 1999.

[4] E.J.Otoo and T.H.Merrett, *A Storage Scheme for Extendible Arrays,* Computing, vol.31, pp.1-9, 1983.

[5] H. Li, H. Huang, Y. Lin, *DSD: Maintain Data Cubes More Efficiently*. Fundam. Inform. 59(2-3): pp.173-190, 2004.

[6] I. S. Mumick, D. Quass, and B. S. Mumick, *Maintenance of Data Cubes and Summary Tables in a Warehouse*, Proc. of the ACM SIGMOD Conference, pp.100-111, 1997.

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

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

[9] K. Y. Lee, M. H. Kim, *Efficient Incremental Maintenance of Data Cubes,* Proc. of the VLDB Conference, pp.823-833, 2006.

[10] M. Riedewald, D. Agrawal, A. E. Abbadi, *Flexible Data Cubes for Online Aggregation*, Proc. of ICDT 2001: pp.159-173 , 2001

[11] M. Riedewald, D. Agrawal, A. E. Abbadi, R. Pajarola, *Space-Efficient Data Cubes for Dynamic Environments*, DaWaK 2000: pp.24-33, 2000.

[12] N. Roussopoulos, Y. Kotidis, and M. Roussopoulos, *Cubetree: Organization of and Bulk Incremental Updates on the Data Cube*, Proc. of the ACM SIGMOD Conference, pp.89-99, 1997.

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

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

[15] S. Geffner, D. Agrawal, A. E. Abbadi, *The Dynamic Data Cube,* EDBT 2000: pp.237-253, 2000.

[16] S. Geffner, M. Riedewald, D. Agrawal, A. E. Abbadi, *Data Cubes in Dynamic Environments*, IEEE Data Eng. Bull. 22(4): pp.31-40, 1999.

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

[18] W. Lehner, R. Sidle, H. Pirahesh, and R. Cochrane, *Maintenance of Cube Automatic Summary Tables*. Proc. of the ACM SIGMOD Conference, pp.512-513, 2000.

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

[20] D. Jin, T. Tsuji, M. Tsuchida, K. Higuchi, An Incremental Maintenance Scheme of Data Cubes, Proc. of DASFAA, 2008 (to appear).