Available online at www.sciencedirect.com
SCIENCE DIRECT®
Information Processing Letters 92 (2004) 99–105
Information Processing Letters
www.elsevier.com/locate/ipl

A compression method for prefix-sum cubes

Heum-Geun Kang [a,*], Jun-Ki Min [a], Seok-Ju Chun [b], Chin-Wan Chung [a]

[a] *Division of Computer Science, Department of Electrical Engineering and Computer Science, Korea Advanced Institute of Science and Technology, 373-1 Kusong-dong, Yusong-gu, Taejon 305-701, Republic of Korea*
[b] *Department of Information and Communication Engineering, Korea Advanced Institute of Science and Technology, 373-1 Kusong-dong, Yusong-gu, Taejon 305-701, Republic of Korea*

Received 21 October 2003; received in revised form 1 March 2004

Available online 18 August 2004

Communicated by S.E. Hambrusch

Keywords: Databases; OLAP; Range-sum query; Compression method; Prefix-sum cube

1. Introduction

On-line analytical processing (OLAP) systems manipulate very large volumes of historical data, typically modeled with a multi-dimensional data model known as a data cube, and provide decision support information to users. It is well known that the data cube is very sparse. Typically, valid cells in a data cube are between 0.0001% and 2% of all cells [6,7]. One of the most important query classes in OLAP is a range-sum query. The evaluation of the range-sum query consumes much time since it requires access to large volume of data. Generally, users of OLAP systems execute a sequence of queries interactively. Thus, OLAP systems should support efficient query evaluation, no matter how large the volume of data to be accessed. Therefore, to support the efficient evaluation of range-sum queries, the prefix-sum cube (PC) was proposed [5]. The PC enables all range-sum queries to be processed in constant time, regardless of the size of the range of a query. However, the PC has been criticized for its update overhead and space requirement [7–9]. Several approaches have been proposed to address update overhead problem [1–3,7,8], and the blocked PC was proposed to reduce the space requirement [5].

Our contributions

To reduce the space requirement of the PC, we propose a compression method for PCs, called the compressed SRPS. Our method is based on the space-efficient relative prefix-sum (SRPS) technique intended to reduce the update overhead [8]. The compressed SRPS is a lossless compression method, and allows queries to be evaluated without decompressing. To our knowledge, our work is the first attempt to losslessly compress prefix-sums in the OLAP environment.

2. Related work

The PC was proposed to achieve the efficient processing of range-sum queries in OLAP environ-

* Corresponding author.
E-mail address: hgkang@islab.kaist.ac.kr (H.-G. Kang).

0020-0190/$ – see front matter
doi:10.1016/j.ipl.2004.05.008

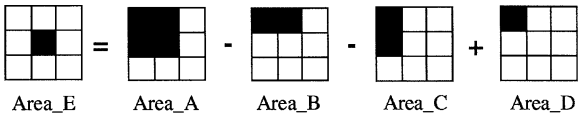

Fig. 1. Range-sum query processing in a prefix-sum cube.

ments [5]. To do this, the PC stores the prefix-sums of data. Given a cell $(c_1, \ldots, c_d)$ of a data cube, the prefix-sum value of the cell can be calculated using

$$\sum_{i_1=0}^{c_1} \sum_{i_2=0}^{c_2} \cdots \sum_{i_d=0}^{c_d} DC(i_1, \ldots, i_d),$$

where d is the dimensionality of the data cube, and $DC(i_1, i_2, \ldots, i_d)$ is the value of the cell $(i_1, i_2, \ldots, i_d)$ of the data cube. Fig. 1 illustrates the process for computing the answer of a range-sum query in a PC. The sum of Area_E can be computed with Area_A's sum, Area_B's sum, Area_C's sum and Area_D's sum. Therefore, a range-sum query can be processed by a constant number $(= 2^d)$ of cell accesses regardless of the size of the query range, where d is the dimensionality of the PC.

To solve the update cost problem of the PC, Riedewald et al. [8] proposed the SRPS technique that partitions a PC into a set of *boxes.* The prefix sums are computed and stored relative to the first cell of a box. By partitioning the PC, the SRPS technique can reduce the number of PC cells affected by an update to a cell of a data cube.

The blocked PC was proposed to reduce the space overhead of the PC [5]. The blocked PC stores a prefix-sum for each block of the corresponding data cube. The space for storing a blocked PC is much smaller than that for storing the corresponding PC. However, the query processing with the blocked PC requires the data cube. It takes more time than that with the PC since all the range-sum queries are answered by accessing and combining 2^d prefix-sums as well as some cells from the data cube, where d is the dimensionality of the blocked PC.

3. The compressed SRPS method

In this section, we present the compressed SRPS method that compresses the prefix-sums in boxes. Fig. 2 shows a box of a data cube and the prefix-sums

	0	1	2	3	4	5
0				4		
1						
2		5				
3					7	
4						
5						

(a) A box of a data cube

	0	1	2	3	4	5
0	0	0	0	4	4	4
1	0	0	0	4	4	4
2	0	5	5	9	9	9
3	0	5	5	9	16	16
4	0	5	5	9	16	16
5	0	5	5	9	16	16

(b) A box of the SRPS cube

Fig. 2. Examples of a box of a data cube and corresponding box of the SRPS cube.

corresponding to the cells in the box. Since the box of the data cube is sparse, the prefix-sums constructed from the box has many repeating values. To compress the box, we should represent the repeating values concisely. Our basic idea is that the repeating values can be represented by subcubes. A subcube can be defined as follows.

Definition 1 (*Subcube*). A subcube is part of a box. A subcube has the following properties.

- The dimensionality of a subcube is identical to the dimensionality of the box containing the subcube.
- All the cells in a subcube have the same value called $V_{subcube}$.
- Let the address of the first cell (cell with the smallest indices) be the first address, and the address of the last cell (cell with the largest indices) the last address. The last addresses of all the subcubes are identical to that of the box.

A subcube can be represented as a pair of the boundary of the subcube and the value $V_{subcube}$. Representing a subcube as a pair is very compact because values of all cells in the subcube are represented by $V_{subcube}$. The boundary of a subcube can be represented by two addresses, the first address and the last address. Fig. 3 shows subcubes which are extracted from the box in Fig. 2(b). Subcubes 2 and 3 are created by values 4 and 5 in the data cube shown in Fig. 2(a), respectively. Subcube 4 is created by the intersection of subcube 2 and subcube 3. The creation of subcube 5 is by value 7 in the data cube. A region outlined by dashed lines in Fig. 3 is the private region of the subcube. The concept of private regions is used in looking up the value stored at a cell of a compressed box, and will be explained in Section 4.

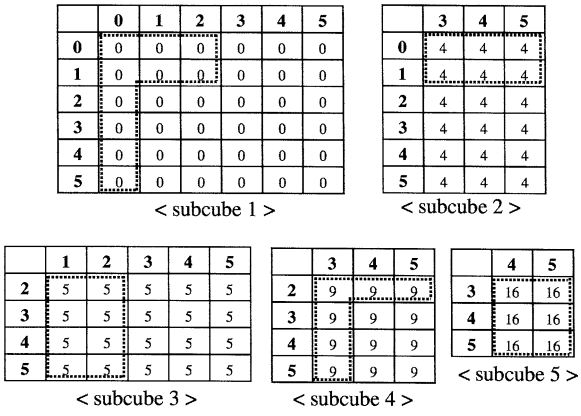

Fig. 3. Subcubes.

The address of a cell is composed of several coordinates. The number of coordinates of the address is equal to the dimensionality of the box. We can store an offset value instead of several values of the coordinates. The offset is a distance from the first cell to the cell in an uncompressed box [10]. The distance can be computed with either the row-major order or the column-major order. For example, with the row-major order, the offset of address $(3, 4)$ is $4 \times 6 + 3$. The offset of a subcube is the offset of the first cell of the subcube.

A compressed box consists of a sequence of subcubes. Let S be a compressed box, and $S[i]$ be the ith subcube of S. If $i < j$, the offset of $S[i]$ is smaller than that of $S[j]$. $S[i]$ is overlapped by $S[i + 1]$, $S[i + 2]$, $\ldots$, $S[n]$ where n is the number of subcubes in S.

Table 1 shows a compact representation of the subcubes depicted in Fig. 3. Subcube 2 is represented by boundary $((3, 0), (5, 5))$ and the value 4. Because all subcubes have the same last address, we do not need to store the last addresses of subcubes. Therefore, we only store the offset and $V_{subcube}$ for each subcube. The final result of compressing the box is "$(0, 0)$, $(3, 4)$, $(13, 5)$, $(15, 9)$, $(22, 16)$". The subcubes in the final result are ordered in an increasing order of the subcubes' offsets.

Table 1
A compact representation of the subcubes

Subcube	First address	Last address	Offset	$V_{subcube}$
1	(0, 0)	(5, 5)	0	0
2	(3, 0)	(5, 5)	3	4
3	(1, 2)	(5, 5)	13	5
4	(3, 2)	(5, 5)	15	9
5	(4, 3)	(5, 5)	22	16

Our method inputs a data cube partitioned into a set of boxes, and produces a compact representation of the prefix-sums for each box. The algorithm compresses the set of boxes one by one. Moreover, during compression of a d-dimensional box, the algorithm only keeps two $(d - 1)$-dimensional arrays which are part of the box.

The algorithm shown in Fig. 5 compresses a box of a two-dimensional data cube by scanning the box *only once.* For each cell, the algorithm computes the prefix-sum of the cell, and checks whether the PC's cell corresponding to the cell is the first cell of a subcube. If the PC's cell is the first cell of a subcube, the algorithm creates a compact representation of the subcube with the address and the value of the PC's cell. As shown in Fig. 4, the prefix-sum of the cell (i, j) can be computed with the prefix-sum of the cell $(i - 1, j)$, the prefix-sum of the cell $(i, j - 1)$, the value of the cell (i, j) and the prefix-sum of the cell $(i - 1, j - 1)$. If the prefix-sum of the cell (i, j) is different from the prefix-sum of the cell $(i - 1, j)$ and the prefix-sum of the cell $(i, j - 1)$, the cell (i, j) is the first cell of a subcube. As mentioned above, the computation for the prefix-sum of the cell requires the previous row and the current row of the PC only. Thus, in order to reduce the space requirement, the algorithm keeps only the previous row (=p_row) and the current row (=c_row) of the prefix-sums. The algorithm processes the first cell in a row at lines 4–6. The remaining cells in the row are processed at lines 7–11. The **for** statement in lines 12–13 copies the values of c_row into p_row to process the next row.

We can generalize the two-dimensional compression algorithm to the d-dimensional compression algorithm by replacing the $(2 - 1)$-dimensional arrays with $(d - 1)$-dimensional arrays. That is, the previous $(2 - 1)$-dimensional array (=p_row) is replaced

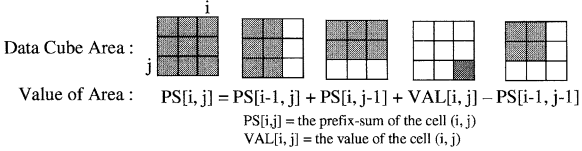

Fig. 4. Computation of a prefix-sum.

```
Algorithm Compressed SRPS
input: DC (a box of a two-dimensional data cube)
output: CB (a compressed box)
Begin
 1. Initialize p_row[ ] to zero // p_row is an array storing the previous row of the prefix-sums
 2. Create the subcube ((0, 0), DC[0][0])
 3. for j = 0 to Ysize − 1 do {// Ysize is the size of Y axis
 4.     c_row[0] := p_row[0] + DC[0][j] // c_row is an array storing the current row of the prefix-sums
 5.     if (c_row[0] ≠ p_row[0])
 6.         Create the subcube ((0, j), c_row[0])
 7.     for i = 1 to Xsize − 1 do {// Xsize is the size of X axis
 8.         c_row[i] := c_row[i − 1] + p_row[i] + DC[i][j] − p_row[i − 1]
 9.         if (c_row[i] ≠ p_row[i] and c_row[i] ≠ c_row[i − 1])
10.             Create the subcube ((i, j), c_row[i])
11.     }
12.     for k = 0 to n − 1 do
13.         p_row[k] := c_row[k]
14. }
End.
```

Fig. 5. The compression algorithm.

by the previous $(d-1)$-dimensional array, and the current $(2-1)$-dimensional array (=c_row) is replaced by the current $(d-1)$-dimensional array. In addition, the **for** statement at line 7 is replaced by $(d-1)$ **for** statements, and so is the **for** statement at line 12. The number of conditions at line 9 becomes d.

4. Looking up and updates

In this section, we present the lookup algorithm and the update algorithm. The lookup algorithm retrieves the value stored in a compressed box without decompressing it. This algorithm serves the core function for the evaluation of the range sum queries. First, we explain the lookup algorithm. Looking up a cell can be decomposed into two logical steps. The first step is to search the subcube that has the private region containing the cell. The private region of a subcube can be defined as follows.

Definition 2 (*Private region*). Let S be a compressed box, and $S[i]$ be the ith subcube of S. The private region of $S[i]$ is the part of the subcube that is *not* overlapped by other subcubes in S which have a greater offset.

By Lemma 4.1, the first step can be replaced by searching the subcube that has the largest offset among those of subcubes containing the input cell. The second step is to return $V_{subcube}$ of the subcube.

Lemma 4.1. *If a cell $c1$ is contained in the private region of a subcube $S[i]$, $S[i]$ has the largest offset among those of subcubes containing $c1$.*

Proof. If the offset of $S[i]$ is not the largest among those of subcubes containing $c1$, there is $S[j]$ such that the offset of $S[j]$ is greater than that of $S[i]$ and $c1$ is contained in $S[j]$. Since $S[j]$ overlaps $S[i]$, $c1$ is not contained in the private region of $S[i]$. This contradicts the assumption. Therefore, $S[i]$ has the largest offset among those of subcubes containing $c1$. □

Fig. 6 shows the lookup algorithm. Since the pairs of a compressed box are sorted by the offset, the number of comparisons can be dramatically reduced by using the binary search. The lookup algorithm is composed of two parts. The first part, lines 1–13, is the binary search to find the position to be started in the subsequent linear lookup. The first part excludes the subcubes with larger offsets than the offset of the lookup cell from the candidate subcubes. The second part, lines 14–18, scans the pairs from the position found by the first part to downside, picks up the subcube that has the largest offset among the subcubes that contain the cell, and returns $V_{subcube}$ of the subcube. The algorithm shown in Fig. 6 looks up

```
Algorithm LookUp
input: CB (compressed box)
input: x, y (coordinates of the cell to be looked up)
output: V_subcube
Begin
 1. offset := offset value computed by x and y
 2. low := 1
 3. high := the number of pairs in the CB
 4. while ((low + 1) < high) {
 5.     mid := (high + low)/2
 6.     offset_mid := offset value of the mid's pair in the CB
 7.     if (offset < offset_mid)
 8.         high := mid
 9.     else if (offset_mid < offset)
10.         low := mid
11.     else
12.         return V_subcube of the mid's pair in the CB
13. }
14. for (i = high; 0 ⩽ i; i = i − 1) {
15.     (x_i, y_i) := the first address of the ith pair in the CB
16.     if (x_i ⩽ x and y_i ⩽ y)
17.         return V_subcube of the ith pair in the CB
18. }
End.
```

Fig. 6. The lookup algorithm.

```
Algorithm Update
input: CB (compressed box)
input: addr (address of the updated cell)
input: value (value to be added)
Begin
1. Check CB whether there is a subcube whose first address is
   equal to addr
2. if (there is no subcube found in above scan) {
3.     Create a subcube subcube1 of addr (first address) and zero
       (V_subcube)
4.     Create new subcubes using the intersection of subcube1 and
       the subcubes in CB
5. Insert subcube1 and the new subcubes created in above step to
   CB
6. }
7. Add value to the V_subcube values of subcubes whose boundaries
   contain addr
End.
```

Fig. 7. The update algorithm.

a cell in a 2-dimensional box. The algorithm can be easily extended for a higher dimensional box.

The update algorithm shown in Fig. 7 first checks the compressed box whether there is a subcube whose first address is equal to the address of the updated cell. If not found, a new subcube corresponding to the cell is inserted to the compressed box. More subcubes may be created by the intersection of the new subcube and the already existing subcubes. Finally, the algorithm adds the value of the cell to the $V_{subcube}$ values of all the subcubes whose boundaries contain the cell.

5. Experiments

We conducted several experiments to show the quantitative effect of our method. The data cube used in the experiments is 4-dimensional, and the data stored in the data cube are randomly populated. We partitioned the corresponding PC into a set of disjoint *boxes* of equal size using the SRPS technique proposed in [8]. All the experiments were conducted on a Linux machine with a 700 MHz Pentium III processor, 512 MBytes main memory and a 20 GBytes hard disk.

First, we measured the compression ratio of our method. The compression ratio is expressed as $1 - B/A$, where A is the size of the original data and B is the size of the compressed data (higher is better). Fig. 8(a) shows the compression ratios on *boxes* of various sizes. Only 2% of cells in each *box* is valid. The result indicates that the compression ratio on the smallest *box* is very high. However, as the size of the *box* increases, the compression ratio decreases. The reason is that, as the size of the *box* increases, the number of subcubes created by the intersection of other subcubes also increases.

Fig. 8(b) shows the compression ratios on a *box* with the size of $5 \times 5 \times 5 \times 5$ as a function of the percentage of valid cells. When the percentage of valid cells is low, the compression ratio is very high. However, as the percentage of valid cells increases, the compression ratio decreases. This is also due to the reason mentioned above. Fig. 8(c) plots the compression ratios for boxes of various dimensionalities. Typically, as the dimensionality increases, the sparseness also increases sharply [4,7]. In this experiment, the length of one dimension is fixed to 5, and 2% of cells is valid when the dimensionality is 4. As the dimensionality increases, the compression ratio increases slowly.

In the next experiment, we measured the response times of range-sum queries executed on various cubes. Table 2 shows the parameters of the experiment. We randomly generated four synthetic data sets. Many data cubes contain many small clustered multi-dimen-

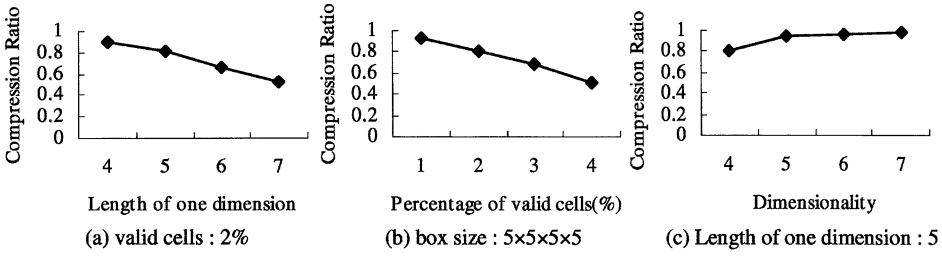

Fig. 8. Compression ratio.

Table 2
Experimental parameters

Parameter	Value
The size of the data cube	$155 \times 155 \times 155 \times 155$
Box size	$5 \times 5 \times 5 \times 5$
No. of *boxes*	923 521
Page size	8 KBytes
Buffer size	2000 pages
Minimum size of one dimension of a query range	7
Maximum size of one dimension of a query range	13

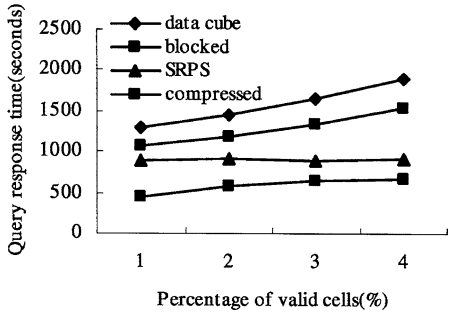

Fig. 9. Performance of query processing.

sional data (dense regions), with sparse points scattered around in the rest of the space [11]. To generate synthetic data sets which closely resemble real data, we used a nonzero value generation method which consists of two steps. The first step randomly generates nonzero values for random cells throughout the data cube, and the second step selects some dense regions and randomly generates nonzero values in those dense regions. The method generates 20% of the total valid cells in the first step and the remainder of the total valid cells in the second step. Each tuple in these data sets contains an attribute for each dimension and an attribute to store the measure value. Table 3 shows details of our generated data sets. Fig. 9 shows a comparison of the response times in executing 8000 range-sum queries which are randomly generated.

As shown in Fig. 9, the compressed SRPS method shows the best performance. The response time of range-sum queries executed on compressed SRPS cubes increases as the percentage of valid cells increases. The reason is that as the percentage of valid cells increases, the compression ratio decreases. Since the sizes of SRPS cubes with differing number of valid cells are the same, the query performance of the SRPS is nearly constant regardless of the percentage of valid cells. Like compressed SRPS, as the percentage of valid cells increases, the response times of range-sum queries executed on the data cube and the blocked PC increases.

Table 3
Synthetic data sets

	Data set A	Data set B	Data set C	Data set D
Percentage of valid cells	1%	2%	3%	4%
Compression ratios	0.93	0.81	0.68	0.51
No. of dense regions	1539	3078	4617	6156
Size of a dense region	$10 \times 10 \times 10 \times 10$			
No. of nonzero values in a dense region	3000			

6. Conclusion

We proposed the compressed SRPS method which was designed to reduce the space requirement of the PC. A distinguished feature of this method is that searches and updates can be done without decompressing. This method reduces the space requirement for storing prefix-sums, and improves the query performance.

Acknowledgements

We appreciate many useful comments of the reviewers. This research was supported by University IT Research Center Project.

References

[1] C.Y. Chan, Y.E. Ioannidis, Hierarchical prefix cubes for range-sum queries, in: Proc. VLDB, 1999, pp. 675–686.

[2] S.J. Chun, C.W. Chung, J.H. Lee, S.L. Lee, Dynamic update cube for range-sum queries, in: Proc. VLDB, 2001, pp. 521–530.

[3] S. Geffner, D. Agrawal, A.E. Abbadi, T.R. Smith, Relative prefix sums: an efficient approach for querying dynamic OLAP data cubes, in: Proc. ICDE, 1999, pp. 328–335.

[4] J. Han, Towards on-line analytical mining in large databases, SIGMOD Record 27 (1) (1998).

[5] C. Ho, R. Agrawal, N. Megiddo, R. Srikant, Range queries in OLAP data cubes, in: Proc. SIGMOD, 1997, pp. 73–88.

[6] R. Kimball, The Data Warehousing ToolKit, Wiley, New York, 1996.

[7] M. Riedewald, D. Agrawal, A.E. Abbadi, pCube: update-efficient online aggregation with progressive feedback and error bounds, in: SSDBM, 2000, pp. 95–108.

[8] M. Riedewald, D. Agrawal, A.E. Abbadi, R. Pajarola, Space-efficient data cubes for dynamic environments, in: DaWaK, 2000, pp. 24–33.

[9] J.S. Vitter, M. Wang, Approximate computation of multidimensional aggregates of sparse data using wavelets, in: Proc. SIGMOD, 1999, pp. 193–204.

[10] Y. Zhao, K. Ramasamy, K. Tufte, J.F. Naughton, Array-based evaluation of multi-dimensional queries in object-relational database systems, in: Proc. ICDE, 1998, pp. 241–249.

[11] D.W. Cheung, B. Zhou, B. Kao, H. Kan, S.D. Lee, Towards the building of a dense-region-based OLAP system, Data & Knowledge Engineering 36 (1) (2001) 1–27.