IJCSE International Journal of Computer Sciences and Engineering Open Access
Research Paper Volume-6, Issue-1 E-ISSN: 2347-2693

Precomputing Shell Fragments for OLAP using Inverted Index Data Structure

D. Datta[1*], A. Koley[2], A. Sarkar[3], S. Chatterjee[3]

[1*]Department of Computer Science, St. Xavier's College, Kolkata, India
[2]Department of Computer Science, Banaras Hindu University, Varanasi, India
[3]Deloitte Consulting US-India Pvt. Ltd, Hyderabad, India

[*]*Corresponding Author: debabrata.datta@sxccal.edu*

Available online at: www.ijcseonline.org

Received: 22/Dec/2017, Revised: 31/Dec/2017, Accepted: 19/Jan/2018, Published: 31/Jan/2018

***Abstract*—** Efficient methods to generate data cubes for On-Line Analytical Processing or OLAP are required for query processing and data analysis. OLAP involves multidimensional analysis of data and as well as selectively extracting and viewing data from different perspectives or points of view. In OLAP, a complex query can lead to many scans of the base relational database, leading to poor performance. This research paper provides an algorithm for the data cube generation suitable for OLAP systems in a fast way. The OLAP cube structure, based on aggregation operations and capable of fast retrieval of data, is extensively explored. The inverted index data structure, which is a mapping from content to index of the said content in any indexed data storage system, is used as an efficient tool for shell fragment computation. A study of efficiency and trade-offs involved in terms of processing complexity and storage space when compared to full cube computation are also provided here.

***Keywords*—** OLAP, data cube, cube shell, shell fragmentation, inverted index data structure, multidimensional analysis

I. INTRODUCTION

A data warehouse is a semantically consistent data store that serves as a physical implementation of a decision support data model [2]. It stores the information an enterprise needs to make strategic decisions. A data warehouse is also often viewed as architecture, constructed by integrating data from multiple heterogeneous sources to support queries, analytical reporting, and decision making. It is a subject-oriented, integrated, time-variant, and non-volatile collection of data for supporting decision making process.

Data warehouse systems serve users or knowledge workers in the role of data analysis and decision making. Such systems can organize and present data in various formats in order to accommodate the diverse needs of different users. These systems are known as Online Analytical Processing (OLAP) systems. A data warehouse is usually modelled by a multidimensional data structure, called a data cube, in which each dimension corresponds to an attribute or a set of attributes in the schema, and each cell stores the value of some aggregate measure such as count or sum. A data cube provides a multidimensional view of data and allows the pre-computation and fast access of summarized data.

There have been many efficient cube computation algorithms proposed, such as ROLAP-based multi-dimensional aggregate computation [3], multi-way array aggregation [7], BUC [6], H-cubing [8], and Star-cubing [9]. Since computing the whole data cube not only requires a substantial amount of time but also generates a huge number of cube cells, there have also been many studies on partial materialization of data cubes [5], iceberg cube computation [6,8,9], computation of condensed, dwarf, or quotient cubes [10,11,12] , and computation of approximate cubes [13]. Besides large data warehouse applications, there are other kinds of applications like bioinformatics, survey-based statistical analysis, and text processing; all of them need the OLAP-styled data analysis.

In OLAP, a complex query can lead to many scans of the base relational table, leading to poor performance. Also, aggregation functions are very popular in OLAP queries. Thus, a new data structure based on aggregation operations and capable of fast retrieval of data was desired. An OLAP cube is a data structure that overcomes the limitation of relational model by providing fast query processing. A data cube allows data to be modelled and viewed perspectives of multiple dimensions. In practice, they are often pre-computed for fast OLAP. Using the pre-computed cubes' shell fragments, cuboid cells of the required data cube

can be dynamically assembled and computed online, resulting in a faster OLAP.

A full data cube of high dimensionality needs massive storage space and unrealistic computation time. One possible solution is to compute a thin cube shell, which is a computation process of considering only a few dimensions from all possible dimensions. For example, all cuboids with 5 dimensions or less in a 50-dimensional data cube, resulting in a thin cube shell of size 5. The resulting cuboid set would require much less computation time and storage space than the full 50-dimensional data cube.

However, there is a major drawback of this approach. Such a thin cube shell does not support 6 or more dimensional OLAP.

Instead of computing a cube shell, only few portions or fragments of it may also be computed. Although a data cube may contain many dimensions, most OLAP operations are performed on only a small number of dimensions at a time. Instead, it is more natural to locate some cuboids of interest at first and then to drill along one or two dimensions to examine the changes of a few related dimensions. This implies that if multidimensional aggregates can be computed quickly on a small number of dimensions inside a high-dimensional space, a faster OLAP can be achieved without materializing the original high-dimensional data cube. Instead, a semi-online computation model with certain pre-processing may offer a more feasible solution. Given a base cuboid, some quick preparation computation can be done offline. After that, a query can be computed online using the pre-processed data. The shell fragment approach follows such a semi-online computation strategy. It explores the inverted index data structure, which is popular in information retrieval and Web-based information systems.

The paper discusses about research work going on this field in the next section. Section three describes the main algorithms involved in the present research work. The next section explain the results obtained and the final section concludes the present discussion with a brief statement about the scope of work in the future.

II. BACKGROUND STUDY

Data cube computation is an essential task in data warehouse implementation. The pre-computation of all or part of a data cube can greatly reduce the response time and enhance the performance of online analytical processing. However, such computation is challenging because it may require substantial computational time and storage space. There are many methods for data cube computation, few of them are explained below:

a. Multiway Array Aggregation

The multiway array aggregation method computes a full data cube by using a multidimensional array as its basic data structure [7]. It is a typical MOLAP approach that uses direct array addressing, where dimension values are accessed via the position or index of their corresponding array locations. Hence, multiway cannot perform any value-based reordering as an optimization technique.

b. BUC: Iceberg Cube Computation

An iceberg cube contains only those cells of the data cube that meet an aggregate condition. It is called an iceberg cube because it contains only some of the cells of the full cube, like the tip of an iceberg. The purpose of the iceberg cube is to identify and compute only those values that will most likely be required for decision support queries. The aggregate condition specifies which cube values are more meaningful and should therefore be stored. This is one solution to the problem of computing versus storing data cubes. BUC, or Bottom Up Construction is an algorithm for the computation of sparse and iceberg cubes [6]. Unlike MultiWay, BUC constructs the cube from the apex cuboid toward the base cuboid. This allows BUC to share data partitioning costs. This processing order also allows BUC to prune during construction, using the Apriori property. Partitioning and sorting are the major costs in BUC's cube computation. Since recursive partitioning in BUC does not reduce the input size, both partitioning and aggregation are costly. Moreover, BUC is sensitive to skew in the data. Performance degrades as skew increases.

c. Star-Cubing

The Star-Cubing algorithm explores both the bottom-up and top-down computation models [9]. On the global computation order, it uses the bottom-up model. However, it has a sub-layer underneath based on the top-down model, which explores the notion of shared dimensions. This integration allows the algorithm to aggregate on multiple dimensions while still partitioning parent group-by's and pruning child group-by's that do not satisfy the iceberg condition. Star-Cubing is able to prune the indicated cuboids because it considers shared dimensions. Because the shared dimensions are identified early on in the tree expansion, re-computing them can be avoided later. Star-Cubing is sensitive to the ordering of dimensions, as with other iceberg cube construction algorithms. For best performance, the dimensions are processed in order of decreasing cardinality. This leads to a better chance of early pruning, because the higher the cardinality, the smaller the partitions, and therefore the higher possibility that the partition will be pruned. Star-Cubing can also be used for full cube computation. When computing the full cube for a dense data set, Star-Cubing's performance is comparable with MultiWay and is much faster than BUC. If the data set is sparse, Star-Cubing is significantly faster than MultiWay and faster than BUC, in most cases. For iceberg cube computation, Star-Cubing is faster than BUC, where the data are skewed and the speed-up factor increases as min sup decreases.

III. WORK DESCRIPTION

The work as described in this paper has explored the inverted index data structure, which is popular in information retrieval and Web-based information systems. Given a high-dimensional data set, the dimensions are partitioned into a set of disjoint dimension fragments; each fragment is then fragmented into its corresponding inverted index representation, and then cube shell fragments are constructed while keeping the inverted indices associated with the cube cells. Using the pre-computed cubes' shell fragments, cuboid cells of the required data cube can be dynamically assembled and computed online. This is made efficient by set intersection operations on the inverted indices. The shell fragments are negligible in both storage space and computation time in comparison with the full data cube. Although a data cube may contain many dimensions, most OLAP operations are performed on only a small number of dimensions at a time. This is because it is not realistic for anyone to comprehend the changes of thousands of cells involving tens of dimensions simultaneously in a high-dimensional space at the same time. Instead, it is more natural to first locate some cuboids by certain selections and then drill along one or two dimensions to examine the changes of a few related dimensions. Most analysts only need to examine the space of a small number of dimensions once they select them. Stemming from the above motivation, compute the shell fragments of data cubes are computed. This is made efficient by set intersection operations on the inverted indices.

The cuboid generation process starts with taking the base cuboid as input. The inverted index data structure is generated based on this base cuboid. A suitable fragment length, say f, is chosen or determined by analysis of previous data. The set of dimensions are split into fragments of length f, except the length of last one, which may be less than or equal to f. All possible combinations of the dimensions in each fragment are generated separately for each fragment. These combinations are used to compute sets of transaction lists, henceforth referred to as TID, of dimension values that appear together by referencing the inverted index data structure. The measure to be aggregated is then aggregated based on the intersection sets of the TID lists obtained using the combination of values. The following is the flowchart describing the system:

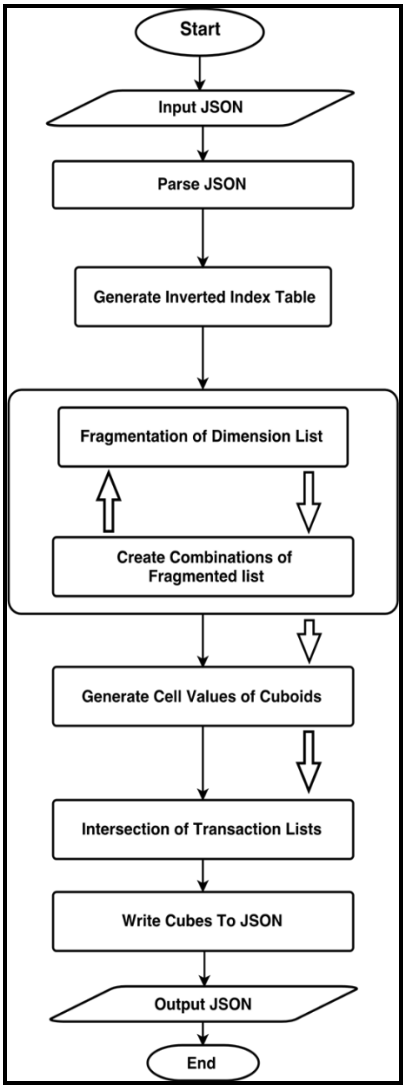

Figure 1: Flowchart of the proposed system

The main component of the proposed system is the generation of inverted index data structure which is a specialised data structure used as a tool for efficient data retrieval. Most modern data retrieval systems from large scale search engines to localised searching mechanisms built into operating systems use inverted index in some form or the other. At the basic level, an inverted index is a mapping from the contents of an indexed storage system (such as a database or a document) to its location in the system.

The inverted index data structure, shown in figure 3 was constructed using an algorithm that requires just one pass of the input table or the base cuboid. All the tuples in the input table are read one at a time. For each tuple, all the dimension values in that tuple are read one at a time. For each dimension value, the inverted index table is searched for that value. If an entry for that dimension value is found, the tuple

ID of the tuple in the input table where that value was read from is inserted in the postings list for the entry. If no match for the dimension value is found after the entire inverted index table (in its current state) is searched, a new entry/ tuple for the new dimension is created with its posting list containing the tuple ID of the tuple in the input table where it was found.

View Table: InputTable

TableName InputTable

Dimensions: From index 1 to index 5 (Ignore this line when displaying output cubes)

TID	A	B	C	D	E	item_count	sum
1	a1	b1	c1	d1	e1	5	70
2	a1	b2	c1	d2	e1	3	10
3	a1	b2	c1	d1	e2	8	20
4	a2	b1	c1	d1	e2	5	40
5	a2	b1	c1	d1	e3	2	30

Measures : 6=item_count, 7=sum,

Figure 2: Transaction List or Input Table

TID	AttributeV...	TIDList	ListSize
0	a1	[1, 2, 3]	3
1	b1	[1, 4, 5]	3
2	c1	[1, 2, 3, 4, 5]	5
3	d1	[1, 3, 4, 5]	4
4	e1	[1, 2]	2
5	b2	[2, 3]	2
6	d2	[2]	1
7	e2	[3, 4]	2
8	a2	[4, 5]	2
9	e3	[5]	1

Figure 3: Inverted Index

Computation of the shell fragments of data cubes includes the following sub-problems:

a) Construction of the Inverted Index:

The construction of the Inverted Index Data Structure based on the Input Table follows the algorithm in Figure 4. Each tuple in the input table is read one at a time. For each tuple read, the values of each dimension are traversed one at a time (only dimension values considered, not measures. The dimStartIndex and dimEndIndex values serve to separate the dimensions from the measures and are constant for a particular table.).

For each dimension value in the Input Table, all the tuples of the entire Inverted Index Table is searched for an occurence of that dimension value, i.e., a check is performed to see if a tuple in the IID exists for that dimension value. If such an entry is found the TID of the tuple from which the dimenson value was read is added to the TID list against the entry of the dimsion value in the IID and the length of the TID list is incremented by one. If, however, an entry for the read dimension value is not found in the IID table, a new tuple is added for it. The TID of the tuple in the Input Table from which it was read is added in the TID list (ie, the TID list has only one element when a new tuple is added.) and the list count is initialized to one.

This process continues until all tuples in the Input Table have been read and processed. When that is done, the IID will have been generated.

b) Fragmentation and Combination

Since, the algorithm requires generating shell fragments from which cuboids are calculated the set of dimensions of the input table are partitioned into independent groups of dimensions, called fragments. For example, If the dimension list was {A,B,C,D,E,F,G} and the fragment length was 3, the set of fragments would be { (A,B,C), (D,E,F), (G) }.

```
procedure generateIID()
1   for each tuple, tup, in input table do
2       for each dimension, d, (form dimStartIndex to
            dimEndIndex) in Input Table do
3           INITIALIZE flag = 0
4           for each tuple, iidtup, in Inverted Index
                Table do
5               if there is an iidtup for d then
6                   ADD TID of tup to list of iidtup
7                   INCREMENT count of iidtup by 1
8                   SET flag = 1
9               end if
10          end for
11          if flag equals 0 then
12              CREATE new iidtup for d
13              SET count = 1
14              INITIALIZE list with TID of tup
15              ADD iidtup to IID table
16          end if
17      end for
18  end for
```

Figure 4: Pseudocode for the generation of Pseudo IID

If fragment length was 4, the partitions would be { (A,B,C,D), (E,F,G) }. In more advanced implementations of the algorithm the fragment length may be algorithmically determined from historical data. The mathematical function of combination is then applied to the dimension set in each fragment. Each fragment generates a set of combinations of dimensions. For example, if combination is performed on fragment (A,B,C), the combinations of dimensions generated will be AB, AC, BC of length 2 each and ABC of length 3. The length of the combinations starts from 2 and go up to the length of the fragment being processed. Figure 5 depicts the process.

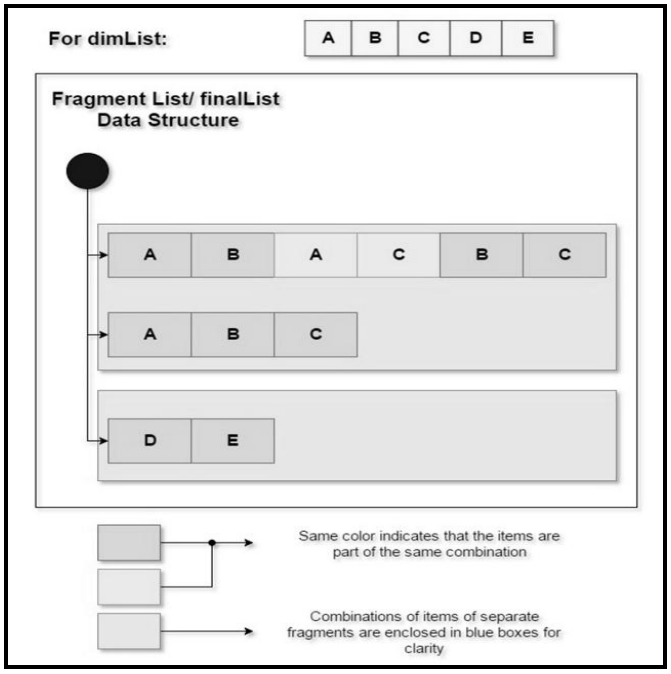

Figure 5: Visualization of how data are handled during fragmentation and combination

c) Computation of Shell Fragments by Intersection

The inverted index can be generalized to multiple dimensions where one can store TIDLists for combinations of attribute values across different dimensions. The cuboids are generated using the combination of dimensions (generated by processing the fragments) to calculates sets of dimension values which occur together one or more times. The tuples in which they occur are recorded by intersecting the sets of TIDs in which the dimension values occur independently i.e., the TID list column of the Inverted Index Data Structure. We then build a local fragment cube by intersecting their corresponding TIDLists and computing the aggregates of the measures using the intersection sets to determine which tuples to consider for aggregation.

If 'f' is the fragment length, then the number of cuboids for every possible combination of each fragment is given by:

$$\binom{f}{f} + \binom{f}{f-1} + \cdots + \binom{f}{2} \quad \text{——— (1)}$$

To find other cuboids, the above expression is multiplied by the floor value of (d / f) where 'd' is the number of dimensions and 2 is added to the expression for the base and apex cuboids. Therefore, the above expression is modified to the following form:

$$\left\{\binom{f}{f} + \binom{f}{f-1} + \cdots + \binom{f}{2}\right\} \left\lfloor \frac{d}{f} \right\rfloor + 2 \quad \text{——— (2)}$$

The above expression is the final one for the number of cuboids generated when the number of dimensions is divisible by the fragment length. If it is not divisible, then for the remainder dimensions expression (1) is used. If k is the number of remainder dimensions and N_c is the total number of cuboids generated, then (2) becomes

$$N_c = \left\{\binom{f}{f} + \binom{f}{f-1} + \cdots + \binom{f}{2}\right\} \left\lfloor \frac{d}{f} \right\rfloor + \left\{\binom{k}{k} + \binom{k}{k-1} + \cdots + \binom{k}{2}\right\} + 2 \quad \text{——— (3)}$$

From the Binomial Theorem,

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n$$

Using the above formula, (3) can be simplified to

$$N_c = (2^f - f - 1)\left\lfloor \frac{d}{f} \right\rfloor + (2^k - k - 1) + 2$$

$$or, \quad N_c = (2^f - f - 1)\left\lfloor \frac{d}{f} \right\rfloor + 2^k - k + 1 \quad \text{——— (4)}$$

where k = d mod f and k < f.

Equation (4) is the final equation that is used to calculate the number of cuboids generated depending on the fragment length and the number of dimensions.

IV. RESULTS AND ANALYSIS

If the input to the algorithm is a Base Cuboid having 'n' dimensions having 'r' number of tuples, an inverted index data structure will be generated mapping each of the (n × r) dimension values to their transaction index in the base cuboid. The fields in the inverted index table can be changed based on implementation conditions but the very basic ones required for it to be of use are TID, dimension value, mentioned as AttributeValue, TIDList (corresponding to the value), length of the TIDList, mentioned as ListSize. This set of fragments is used to generate cube cells using a combination algorithm and then are further used to generate the cubes using set intersection. The process is illustrated as in the next page.

TID	AttributeV...	TIDList	ListSize
0	a1	[1, 2, 3]	3
1	b1	[1, 4, 5]	3
2	c1	[1, 2, 3, 4, 5]	5
3	d1	[1, 3, 4, 5]	4
4	e1	[1, 2]	2
5	b2	[2, 3]	2
6	d2	[2]	1
7	e2	[3, 4]	2
8	a2	[4, 5]	2
9	e3	[5]	1

Figure 6: The inverted index data structure

TID	AttributeV...	TIDList	ListSize
0	a1	[1, 2, 3]	3
1	b1	[1, 4, 5]	3
2	c1	[1, 2, 3, 4, 5]	5
3	d1	[1, 3, 4, 5]	4
4	e1	[1, 2]	2
5	b2	[2, 3]	2
6	d2	[2]	1
7	e2	[3, 4]	2
8	a2	[4, 5]	2
9	e3	[5]	1

Figure 7: The IID. The highlighted rows are used to form one cell of Cuboid ABC

The implementation of computing shell fragments is relatively efficient than most cube shell computations. One option is to calculate thin cube shells. Suppose, cubes are to be generated for a 50-dimensional data cube of shell size 5, then computation of ${}^{50}C_5 + {}^{50}C_4 + {}^{50}C_3 + {}^{50}C_2 + 50 = 2369935$ cuboids is required.

Cell	TIDList	item_count
[a1, b1, c1]	[1]	5
[a1, b2, c1]	[2, 3]	11
[a2, b1, c1]	[4, 5]	7

Figure 8: The Cuboid, ABC. The highlighted cell is formed by intersecting the TIDs from the highlighted rows in Figure 7.

The number of cuboids generated when shell fragments are pre-computed depends on the value of the fragment length and the total number of dimension. The following formula is used to calculate the total number of cuboids that are generated is given by:

$$N_c = (2^f - f - 1)\left\lfloor \frac{d}{f} \right\rfloor + 2^k - k + 1$$

Using the example of the 50-dimensional data cube, the total number of cuboids that is generated in this case is given by:

$$N_c = (2^5 - 5 - 1)\left\lfloor \frac{50}{5} \right\rfloor + 2 = 262$$

The efficiency is evident from the number of cuboids that is generated in both cases. If both cases are compared, then efficiency increases with an increase in the number of dimension. So, for very large data cubes, the second method of computation of shell fragments is more efficient. The computational complexity for pre-computing shell fragments with the implemented approach is given by $O(2^n)$. Here, f is the fragment length. So, the computational complexity increases with an increase in the fragment length and the pre-computation of shell fragments as an exponential time complexity.

Number of Dimension	Fragment Length	Number of Cuboids	Total Execution Time (in seconds)
5	1	2	0.02
	2	4	0.07
	3	7	0.045
	4	13	0.21
	5	28	0.48
15	1	2	0.05
	2	9	0.09
	3	22	0.43
	4	39	0.57
	5	80	2.96
	6	120	87.7
18	1	2	0.02
	2	11	0.07
	3	26	0.21
	4	47	0.20
	5	84	0.39
	6	173	5.22
25	1	2	0.03
	2	14	0.19
	3	34	0.15
	4	68	2.29
	5	132	9.03

Figure 9: Comparative study of the execution time by changing the dimensions and fragment lengths

Figure 9 shows the execution time for different dimensions with different fragment lengths on a basic test machine having Intel(R) Core(TM) i5 processor at 1.70GHz and 4GB of physical memory having x64-based architecture.

V. CONCLUSION and Future Scope

To conclude, in the Shell Fragment approach, given a high-dimensional data set, the dimensions are partitioned into a set of disjoint dimension fragments. After that, each fragment is converted into its corresponding inverted index representation. Finally, cube shell fragments are constructed while keeping the inverted indices associated with the cube cells.

With this approach, for high-dimensional OLAP, the total space that is needed to store such shell fragments is negligible in comparison with a high dimensional cube, so is the online computation overhead which can be performed from the available shell fragments. The algorithm, thus, forces a trade-

off between pre-computation of entire cubes, which require both unrealistic time complexities and large storage but once the process is done no more computation is required. Furthermore, a small computation time during query processing for drastically low pre-computation times and storage required. A systematic study of the applications of this approach could be a promising direction for future research. This algorithm can be used is any situation where fast OLAP is required.

REFERENCES

[1] X. Li, J. Han, and H. Gonzalez, "High-dimensional OLAP: A minimal cubing approach", In Proceedings of 30th International Conference on VLDB, pp. 528 – 539, 2004.

[2] Chaudhari, S., U. Dayal, "An overview of Data Warehousing and OLAP Technology", ACM SIGMOD, pp. 65 – 74, 1997.

[3] S. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan and S. Sarawagi. On the computation of multidimensional aggregates. In Proceedings of 22nd International Conference on VLDB, pp. 506 – 521, 1996.

[4] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow and H. Pirahesh. "Data cube: A relational aggregation operator generalizing group-by, cross-tab and subtotals", Data Mining and Knowledge Discovery, pp. 29 – 54, 1997.

[5] V. Harinarayan, A. Rajaraman, and J. D. Ullman, "Implementing data cubes efficiently", ACM SIGMOD, pp. 205 – 216, 1996.

[6] K. Beyer and R. Ramakrishnan, "Bottom-up computation of sparse and iceberg cubes", ACM SIGMOD, pp. 359 – 370, 1999.

[7] Y. Zhao, P. M. Deshpande, and J. F. Naughton, "An array-based algorithm for simultaneous multidimensional aggregates", ACM SIGMOD, pp. 159 – 170, 1997.

[8] J. Han, J. Pei, G. Dong, and K. Wang, "Efficient computation of iceberg cubes with complex measures", ACM SIGMOD, pp. 1 – 12, 2001.

[9] D. Xin, J. Han, X. Li, Z> Shao and B. W. Wah, "Computing iceberg cubes by top-down and bottom-up integration, The StarCubing Approach". IEEE Transactions on Knowledge and Data Engineering, Vol. 19, Issue – 1, pp. 111 – 126, 2007.

[10] W. Wang, H. Lu, J. Feng, and J. X. Yu, "Condensed cube: An effective approach to reducing data cube size", In Proceedings of 18th International Conference on Data Engineering, 2002.

[11] Y. Sismanis, N. Roussopoulos, A. Deligianannakis, and Y. Kotidis, "Dwarf: Shrinking the petacube". ACM SIGMOD, pp. 464 – 475, 2002.

[12] L. V. S. Lakshmanan, J. Pei, and J. Han, "Quotient cube: How to summarize the semantics of a data cube", In Proceedings of 28th International Conference on VLDB, pp. 778 – 789, 2002.

[13] D. Barbara and M. Sullivan, "Quasi-cubes: Exploiting approximation in multidimensional databases. CM SIGMOD, pp. 12 – 17, 1997.

Authors Profile

Mr. D Datta pursued Master of Technology from University of Calcutta, India and he is currently pursuing his Ph.D. in Technology from the same university. He is an Assistant Professor in the department of Computer Science, St. Xavier's College (Autonomous), Kolkata, India He is a life member of IETE. He has published more than 20 research papers in reputed international journals and conferences His main research work focuses on Data Analysis. He has more than 10 years of teaching experience and has more than 4 years of Research Experience.

Mr A Koley pursued B.Sc. in Computer Science from St. Xavier's College (Autonomous), Kolkata, India and is currently doing his Masters in Computer Science from Banaras Hindu University, Varanasi, India.

Miss A Sarkar pursued B.Sc. in Computer Science from St. Xavier's College (Autonomous), Kolkata, India and is currently working as an analysist at Deloitte Consulting US-India Pvt. Ltd, Hyderabad, India.

Mr. S Chatterjee pursued B.Sc. in Computer Science from St. Xavier's College (Autonomous), Kolkata, India and is currently working as an analysist at Deloitte Consulting US-India Pvt. Ltd, Hyderabad, India.

