A Data Cube Representation for Efficient Querying and Updating

Viet Phan-Luong

▶ **To cite this version:**

Viet Phan-Luong. A Data Cube Representation for Efficient Querying and Updating. 2016 International Conference on Computational Science and Computational Intelligence, Dec 2016, Las Vegas, Nevada, USA, United States. hal-01796026

HAL Id: hal-01796026
https://amu.hal.science/hal-01796026

Submitted on 2 Jun 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Data Cube Representation for Efficient Querying and Updating

Viet Phan-Luong
Aix-Marseille Université
LIF - UMR CNRS 7279
Marseille, France
Email: viet.phanluong@lif.univ-mrs.fr

***Abstract*—This paper presents an approach to data cube representation that allows for efficient computation of query response and data cube update. The representation is based on (i) a recursive construction of the power set of the fact table dimension scheme and (ii) a prefix tree structure for the compact storage of cuboids. Experimental results on large datasets show that the approach is efficient for data cube query and incremental update.**

***Keywords*-data warehouse; data cube; data mining;**

I. INTRODUCTION

In data warehouse, a data cube built on a fact table with n dimensions and m measures can be seen as the result of the set of the Structured Query Language (SQL) group-by queries over the power set of dimensions, with aggregate functions over the measures. The result of each SQL group-by query is an aggregate view, called a cuboid, over the fact table. The concept of data cube offers important interests to business intelligence as it provides aggregate views of data over multiple combinations of dimensions that help managers to make appropriate decision in their business.

Though the concept of data cube is simple, there are many important issues in computation and storage space because of the exponential number of cuboids. To tackle these issues, in [9], an I/O-efficient technique based upon a multiresolution wavelet decomposition is used to build an approximate and space-efficient representation of data cubes. To answer an OLAP query, instead of computing the exact response, an approximate answer is computed on this representations. In the iceberg data cube approach [8][10][17][19], instead of computing all aggregates, only those above certain thresholds are computed for the data cube. This approach does not allow to answer to all queries because data cubes are partially computed.

The other approaches search to represent the entire data cube with efficient methods for computation and storage [2][7][22]. The computing time and storage space can be optimized based on equivalence relations defined on aggregate functions [11] [16] or on the concept of closed itemsets in frequent itemset mining [15] or by reducing redundancies between tuples in cuboids, using tuple references [2] [11][13] [14][16][18][21]. In these approaches, the computation is usually organized on the complete lattice of subschemes of the fact table dimension scheme. The computation can traverse the complete lattice in a top-down or bottom-up manner. To create cuboids, the sort operation are used to reorganize tuples: tuples are grouped and the aggregate functions are applied to the measures. To optimize the storage space, only aggregated tuples with aggregated measures are directly stored on disk. Non-aggregated tuples are not stored but represented by references to the tuples where the non aggregated tuples are originated or to tuples in the fact table. The reduction of the storage space could increase the query response time. An efficient data cube representation should offer a good trade-off between the reduction and the query response time.

In the life cycle of a data warehouse, the fact table can growth with new tuples, and the data cube representation needs to be frequently updated. In [21], three update methods are implemented. (i) Merge method: build the data cube of the new tuples and merge it with the current data cube. (ii) Direct method: update each cuboid of the current data cube with the new tuples. (iii) Reconstruction method: reconstruct the entire data cube of the fact table updated with the new tuples. These methods are experimented on different data cube approaches to incrementally build the data cube, where the size of the new dataset is gradually changed: from 1% to 10% of the size of the current dataset.

A. *Contributions*

This work is developed on the paper [12] that presents a simple and efficient approach to represent the entire data cubes. The main idea of [12] is: Among the cuboids of a data cube, there are ones that can be easily and rapidly get from the others, with no important computing time. These others, called the prime and next-prime cuboids, are computed and stored on disk using an integrated binary search prefix tree structure for compact representation and efficient search.

In contrast to the other approaches, the approach [12] computes the data cube representation neither for a specific aggregate function, nor for a specific measure. But it computes a representation that allows for computing all cuboids with any measure and any aggregate function. In fact, for each cuboid in the representation, it computes an index: a set of rowIds that reference to tuples in the fact table.

In the present work, we study the efficiency of data cube query and update, based on this representation. The

contribution consists of:

– An algorithm for computing the response to group-by SQL queries with aggregate functions.

– An algorithm for data cube updating based on the direct method.

The paper is organized as follows. Section 2 recalls the main concepts and algorithms in [12], in particular, the concept of prime and next-prime schemes and cuboids. Section 3 presents an algorithm for computing the group-by query with aggregate functions on data cube. Section 4 presents an algorithm for data cube update. Section 5 reports the experimental results. Finally, conclusion is in Section 6.

II. Last-half data cube representation

This section recalls the main concepts and algorithms presented in [12].

A. A structure of the power set

A data cube over a dimension scheme S is the set of cuboids built over all subsets of S, that is the power set of S. As in most of existing work, dimensions (attributes) are encoded in integer, let us consider $S = \{1, 2, ..., n\}$, $n \geq 1$. The power set of S can be recursively defined as follows.

1) The power set of $S_0 = \emptyset$ (the empty set) is $P_0 = \{\emptyset\}$.
2) For $n \geq 1$, the power set of $S_n = \{1, 2, ..., n\}$ can be recursively defined as follows:

$$P_n = P_{n-1} \cup \{X \cup \{n\} \mid X \in P_{n-1}\} \qquad (1)$$

Let us call P_{n-1} the *first-half power set* of S_n and the second operand of P_n the *last-half power set* of S_n.

Example 1: For $n = 3$, $S_3 = \{1, 2, 3\}$, we have:
$P_0 = \{\emptyset\}$, $P_1 = \{\emptyset, \{1\}\}$, $P_2 = \{\emptyset, \{1\}, \{2\}, \{1, 2\}\}$,
$P_3 = \{\emptyset, \{1\}, \{2\}, \{1, 2\}, \{3\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$.

The first-half power set of S_3 is $P_2 = \{\emptyset, \{1\}, \{2\}, \{1, 2\}\}$ and the last-half power set of S_3 is $\{\{3\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$, got by adding 3 to each element of P_2.

B. Last-half data cube and first-half data cube

We call a scheme in the last-half power set a *prime scheme* and a cuboid over a prime scheme a *prime cuboid*. The prime schemes are characterized by the last dimension n. For efficient computing, the prime cuboids are computed by pairs. Each pair is composed of two prime cuboids: the scheme of the first one has dimension 1 and the scheme of the second one is obtained from the scheme of the first one by removing dimension 1. We call the second prime cuboid the *next-prime cuboid*.

The set of all cuboids over the prime (or next-prime) schemes is called the *last-half data cube*. The set of all remaining cuboids is called the *first-half data cube*. In this approach, the last-half data cube is computed and stored on disks. Cuboids in the first-half data cube are computed as queries based on the last-half data cube.

C. Integrated binary search prefix tree

For efficient storage and search in cuboids, [12] proposed to store cuboids in an integrated binary search prefix tree structure (BSPT). In C language, this structure is defined:

```
typedef struct bsptree Bsptree; // Binary search prefix tree
struct bsptree{ Elt data;          // data at a node
                Ltid *ltid;        // list of RowIds
                Bsptree *son, *lsib, *rsib; };
```

where son, $lsib$, and $rsib$ represent respectively the son, the left and the right siblings of nodes, and $ltid$ is reserved for the list of tuple identifiers ($RowId$). For efficient memory use, $ltid$ is stored only at the last node of each path in the BSPT. With this representation, each binary search tree contains all siblings of a node in the normal prefix tree.

Example 2:

Consider Table I that represents a fact table over the dimension scheme $ABCD$ and a measure M. Figure 1 represents the BSPT of the tuples over the scheme $ABCD$ of the fact table R1, where we suppose that with the same letter x, if $i < j$ then $xi < xj$, e.g., $a1 < a2 < a3$. In this figure, the continuous lines represent the son links and the dashed lines represent the lsib or rsib links.

Table I
FACT TABLE R1

RowId	A	B	C	D	M
1	a2	b1	c2	d2	m1
2	a3	b2	c2	d2	m2
3	a1	b1	c1	d1	m1
4	a1	b1	c2	d1	m3
5	a3	b3	c2	d3	m2

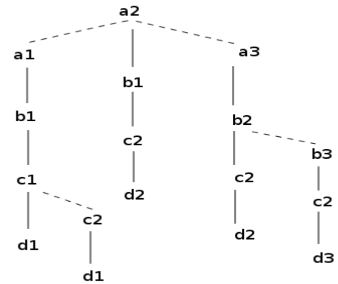

Figure 1. A Binary Search Prefix Tree

Algorithm Tuple2Ptree: Insert a tuple into a BSPT.
Input: A BSPT P, a tuple $ldata$, and a list of tids lti.
Output: The tree P updated with $ldata$ and lti.
Method:

If P is null then
create P with P->data = head(ldata),
P->son = P->lsib = P->rsib = NULL;
if queue(ldata) is null then P->ltid =lti
else P->son = Tuple2Pree(P->son, queue(ldata), lti);

Else if P->data > head(ldata) then
P->lsib=Tuple2Ptree(P->lsib, ldata,lti);
else if P->data < head(ldata) then
P->rsib=Tuple2Ptree(P->rsib, ldata,lti);
else if queue(ldata) is null then
P->ltid = append(P->ltid, lti);
else P->son = Tuple2Ptree(P->son, queue(ldata), lti);
return P;

In algorithm Tuple2Ptree, $head(ldata)$ returns the first element of $ldata$, $queue(ldata)$ returns the queue of $ldata$ after removing $head(ldata)$, and $append(P\text{->}ltid, lti)$ adds the list lti of tids to the list $ltid$ associated with node P.

Using the algorithm *Tuple2Ptree* we can create a BSPT on a table of tuples. In this BSPT, tuples are regrouped over the dimension scheme of the table, hence constitute a cuboid. As nodes corresponding to each attribute of tuples are organized in binary search tree structures, we can get the cuboid with groups of tuples in increasing (or decreasing) order.

In what follows, we consider a fact table R over a dimension scheme $S = \{1, 2, ..., n\}$ and the measures $m_1, ..., m_k$.

D. Computing the last-half data cube

The last-half data cube on R can be built as follows.

– Firstly, build the prime and next-prime cuboids based on the fact table, one over S and the other over $S - \{1\}$. To keep the computation track, the scheme S is added to a list called the *running scheme* and denoted by RS.

– In the sequence, for the currently pointed scheme X in RS, for each dimension $j \in X, j \neq 1$ and $j \neq n$, if $X - \{j\}$ is not yet in RS, then we append $X - \{j\}$ to RS, and build the cuboids over $X - \{j\}$ and $X - \{j\} - \{1\}$, based on the previously built cuboid over X.

In RS, each prime scheme is associated with information that allows to locate the corresponding prime and next-prime cuboids stored on disk.

Example 3: Generation of running scheme (RS).

Table II shows the simplified last-half data cube computation process on the fact table R over the dimension scheme $S = \{1, 2, 3, 4, 5\}$. It shows only the generation of the prime cuboid schemes. The prime schemes appended to RS are in the first column named RS. The final state of RS is $\{12345, 1345, 1245, 1235, 145, 135, 125, 15\}$. In Table II the schemes marked with x (e.g., 145x) are those already added to RS and are not re-appended to RS.

Table II
GENERATION OF THE RUNNING SCHEME OVER $S = \{1, 2, 3, 4, 5\}$.

RS	Generated		
12345	1345	1245	1235
1345	145	135	
1245	145x	125	
1235	125x	135x	
145	15		
135	15x		
125	15x		
15			

E. Data Cube representation

The data cube of R is represented by three components:

1) The last-half data cube of which the cuboids are precomputed and stored on disk in the format of BSPT.

2) RS : each prime scheme in RS has an identifier number that allows to locate the files corresponding to the prime and next-prime cuboids in the last-half data cube.

3) A relational table F over $RowId, M_1, ..., M_k$ that represents the measures associated with each tuple of R.

Clearly, such a representation reduces about 50% of the storage space of the entire data cube, because it mainly consists of the last-half data cube stored in the BSPT format.

F. Computing the first-half data cube

Let X be the scheme of a cuboid in the first-half data cube that we need to retrieve ($X \subset \{1, 2, ..., n\}$ and $n \notin X$). Based on the last-half data cube, the cuboid over X is computed as follows.

Let C be the stored cuboid over $X \cup \{n\}$ (C is in the last-half data cube). As tuples of a prime or next-prime cuboid are stored in the BSPT format, tuples with the same prefix are regrouped together. To build the cuboid over X based on C, we read C sequentially and for each group of the tuples with the same prefix over X, we create a tuple for cuboid over X and merge the lists of rowIds associated with different suffixes (over n) of the prefix to create the list of rowIds to associate with the created tuple. We call this process the *aggregate-projection* of the cuboid C on X.

III. QUERYING DATA CUBES

The general form of a group-by query on a fact table is:

```
Select ListOfDimensions, f1(m1),..,fi(mi)
From Fact_Table
Where ConditionsOnDimensions
Group by ListOfDimensions
Having ConditionsOn f1(m1), ..., fi(mi);
```

Where `f1(m1), ..., fi(mi)` are aggregate functions on measures `m1, ..., mi`.

As in this data cube representation (subsection II-E), each cuboid over a scheme X in the last-half data cube is an index table for groups of tuples of the fact table (regrouped over X), to get the cuboid over a scheme X with a specific measure m and a specific aggregate function f, we can process as follows.

1) If X is a prime or next-prime scheme, then
1.1) Use RS to get the cuboid over X; let C be this cuboid.
2) Else, let $Sch = X \cup \{n\}$ (Sch is prime or next-prime)
2.1) Use RS to get the cuboid over Sch; call it Cn.

2.2) Proceed the aggregate-projection of Cn on X to get the cuboid over X; let C be this cuboid.

3) For each tuple t in the cuboid C,

3.1) If t satisfies the condition of clause WHERE, then

3.1.1) Using the table F in the data cube representation (subsection II-E) and the rowIds associated with t to get the set of values of the measure m;

3.1.2) Compute $f(m)$ on this set of measures.

3.1.3) If $f(m)$ satisfies the HAVING condition, then put t and $f(m)$ into the response.

IV. Updating data cubes

When new data coming into the fact table, as the data cube is represented by its last-half, to update the data cube, we only need to update its last-half. As we do not walk the complete lattice of the cuboids, we can update each cuboid independently. The three methods of data cube update proposed in [21] can be applied to the last-half data cube representation. This work has implemented the data cube update by the direct method. For that, each cuboid of the current last-half cube is restored from disk to main memory, in a binary search prefix tree. The projection of the new tuples on the scheme of the stored cuboid is inserted into the tree. Precisely, we use the following algorithm to update the last-half cube.

Algorithm LastHalfCubeUpdate: Update the last-half data cube with new tuples.

Input: The data cube representation (LC, RS, F) of R, where LC the last-half data cube, RS is the running scheme, F the relational table, and a new fact table NR.

Output: $(LC', RS, F \cup NF)$ where LC' is the last-half data cube of the updated fact table $R \cup NR$.

Method:

For each Sch in RS do,

1) Let C be the prime cuboid over Sch; From LC, restore C in a $BSPT_1$.

2) For each tuple t of the new fact table NR, insert $t[Sch]$ (the restriction of t to Sch) into the $BSPT_1$, using algorithm Tuple2Ptree.

3) Save the $BSPT_1$ to disk.

4) Let Cn be the next-prime cuboid over $Sch - \{1\}$; From LC, restore Cn in a $BSPT_2$.

5) For each tuple $t \in NR$, insert $t[Sch - \{1\}]$ into the $BSPT_2$, using Tuple2Ptree.

6) Save the $BSPT_2$ to disk.

V. Experimental results

The present approach is implemented in C and experimented on a laptop with 8 GB memory, Intel Core i5-3320 CPU @ 2.60 GHz x 4, running Ubuntu 12.04 LTS. To get some ideas about its efficiency we recall here, as references, some experimental results in [21] (run on a Pentium 4 (2.8 GHz) PC with 512 MB memory under Windows XP), on real and synthetic datasets. The work [21] experimented many existing and well known methods for computing and representing data cube as Partitioned-Cube (PC), Partially-Redundant-Segment-PC (PRS-PC), Partially-Redundant-Tuple-PC (PRT-PC), BottomUpCube (BUC), Bottom-Up-Base-Single-Tuple (BU-BST), and Totally-Redundant-Segment BottomUpCube (TRS-BUC). For the present work, we report only the experimental results on two real datasets CovType [3] and SEP85L [4]. By reporting the results of [21], we do not want to really compare the present approach with those methods, because we do not have sufficient conditions to implement and to run those methods on the same system and machine. Moreover, in those methods, the data cube representation is computed for a specific measure and a specific aggregate function, whereas in the present approach, the representation is prepared for computing data cube with any measure and any aggregate function. Apart CovType and SEP85L, we also experimented on two other real datasets that are not used in [21]. These datasets are STCO-MR2010_AL_MO [5] and *OnlineRetail* [6].

CovType is a forest cover-type dataset with 581,012 tuples on ten dimensions with cardinality as follows: Horizontal-Distance-To-Fire-Points (5,827), Horizontal-Distance-To-Roadways (5,785), Elevation (1,978), Vertical-Distance-To-Hydrology (700), Horizontal-Distance-To-Hydrology (551), Aspect (361), Hillshade-3pm (255), Hillshade-9am (207), Hillshade-Noon (185), and Slope (67).

SEP85L is a weather dataset. It has 1,015,367 tuples on nine dimensions with cardinality as follows: Station-Id (7,037), Longitude (352), Solar-Altitude (179), Latitude (152), Present-Weather (101), Day (30), Weather-Change-Code (10), Hour (8), and Brightness (2).

STCO-MR2010_AL_MO is a census dataset with 640,586 tuples on ten integer and categorical attributes (transformed into integer attributes): RESPOP (9,953), CTYNAME (1,049), COUNTY (189), IMPRACE (31), STATE (26), STATENAME (26), AGEGRP (7), SEX (2), ORIGIN (2), SUMLEV (1).

OnlineRetail is a transaction data set of a UK-based online retail. This dataset has incomplete data, integer and categorical attributes. After verifying, transforming categorical attributes into integer attributes, we retain 393,127 complete data tuples on ten dimensions: CustomerID (4,331), StockCode (3,610), UnitPrice (368), Quantity (298), Minute (60), Country (37), Day (31), Hour (15), Month(12), and Year (2).

Table III presents the experimental results approximately got from the graphs in [21], where “avg QRT” denotes the average query response time and “Construction time” denotes the time to construct the (condensed) data cube. Note that, [21] did not specify whether the construction time includes the time to read/write data to files.

Table III
EXPERIMENTAL RESULTS IN [21]

Algorithms	Storage space	Construction time	avg QRT
CovType			
PC	#12.5 Gb	1900 sec	
PRT-PC	#7.2 Gb	1400 sec	
PRS-PC	#2.2 Gb	1200 sec	3.5 sec
BUC	#12.5 Gb	2900 sec	2 sec
BU-BST	#2.3 Gb	350 sec	
BU-BST+	#1.2 Gb	400 sec	1.3 sec
TRS-BUC	#0.4 Gb	300 sec	0.7 sec
SEP85L			
PC	#5.1 Gb	1300 sec	
PRT-PC	#3.3 Gb	1150 sec	
PRS-PC	#1.4 Gb	1100 sec	1.9 sec
BUC	#5.1 Gb	1600 sec	1.1 sec
BU-BST	#3.6 Gb	1200 sec	
BU-BST+	#2.1 Gb	1300 sec	0.98 sec
TRS-BUC	#1.2 Gb	1150 sec	0.5 sec

Table IV
EXPERIMENTAL RESULTS OF THIS WORK

CovType	Storage space	Run time	avg QRT
Last-Half Cube	7 Gb	1018 sec	
First-Half Cube	6,2 Gb	435 sec	
Data Cube	13,2 Gb	1453 sec	0.43 sec
SEP85L			
Last-Half Cube	2.8 Gb	444 sec	
First-Half Cube	2.6 Gb	172 sec	
Data Cube	5.4 Gb	616 sec	0.34 sec
STCO-MR2010_AL_MO			
Last-Half Cube	3.4 Gb	740 sec	
First-Half Cube	3.2 Gb	209 sec	
Data Cube	6.6 Gb	949 sec	0.20 sec
OnlineRetail			
Last-Half Cube	3 Gb	426 sec	
First-Half Cube	2.4 Gb	185 sec	
Data Cube	5.4 Gb	611 sec	0.18 sec

A. *On building last-half cube and querying first-half cube*

Table IV reports the results of the present work, where the term “run time” means the time from the start of the program to the time the last-half (or first-half) data cube is completely constructed, including the time to read/write input/output files. The avg QRT in Table IV corresponds to the average run time for retrieving a cuboid based on the precomputed and stored cuboids. That is, the average query response time for SEP85L is 172s/512 = 0.34 second and for CovType 435s/1024 = 0.43 second, because the cuboids in the last-half data cube are precomputed and stored, only querying on the first-half data cube needs computing.

B. *On query with aggregate functions*

Based on the last-half data cube representation, the group-by SQL queries with aggregate functions can be evaluated. The queries in the following form are experimented on CovType and SEP85L,

Table V
TOTAL RESPONSE TIME OF AGGREGATE FUNCTION QUERY

CovType	MAX	COUNT	SUM	AVG	VAR	MEAN
Last	484	467	481	552	531	503
First	352	422	444	514	497	446
Cube	836	889	925	1066	1028	949

SEP85L	MAX	COUNT	SUM	AVG	VAR	MEAN
Last	196	172	195	223	225	202
First	148	154	180	204	207	179
Cube	344	326	375	427	432	381

```
Select ListOfDimensions, f(m)
From Fact_Table
Group by ListOfDimensions;
```

where `f` is either COUNT, MAX, SUM, AVG, or VARIANCE. For each data set, we compute the query for all cuboids in each half. For instance, for the last-half data cube of CovType, we run the above query for 512 cuboids of the last-half. In Table V, Last, First, and Cube represent respectively the last-half, the first-half data cube and the data cube itself. A value in each column of Table V represents the total query response time in seconds on the corresponding aggregate function. The total time on each line is the sum of the query response time on all cuboids of the corresponding data cube part. For instance, for the last-half data cube of CovType, the total query response time is the sum of the query response time on 512 cuboids. In particular, the column (MEAN) represents average of the total query response time on the five aggregate functions. All time includes all computing time and the time to read/write the data/results from/to disk.

C. *On data cube update*

We experimented the data cube update on the same four datasets. Each one is divided into two parts: the first part is about 90% of the original dataset and the second part is about 10%. The first part is used to create the last-half data cube and the second part is used to update the created last-half. So, after updating, we have the same last-half data cube as we have recreated it with the entire original dataset. In such a way, we can compare the updating time with the time for recreating the last-half data cube with the entire updated dataset. Concretely, the results are reported in Table VI, where Part 1 and Part 2 represent the number of tuples in each part of data sets, Update represents the time in seconds for updating (it includes the time for restoring the current last-half cube in main memory, for updating it, and for saving the result on disk), Create represents the time that is copied from Table IV, and Earn is the time that we can save by the update in comparison with the recreation the last-half data cube on the entire updated fact table.

Table VI
RECREATE TIME VERSUS UPDATE TIME

Dataset	Part 1	Part 2	Create	Update	Earn
CovType	522909	58103	1018	928	90
SEP85L	913831	101536	444	372	72
STCO-M	576528	64507	740	332	408
OnlineR	353814	39313	426	369	57

VI. CONCLUSION

Based on the last-half representation of data cube, the experimental results on SQL group-by query with various aggregate functions show that:

(i) For CovType, it takes on average 949 seconds, while computing both the last-half and the first-half data cube takes 1453 seconds, we earn about 504 seconds (35%).

(ii) For SEP85L, it takes on average 381 seconds, while computing both the last-half and the first-half data cube takes 616 seconds, we earn about 235 seconds (38%).

On data cube update, the experimental results show that the time saving by the direct method, in comparison with the time for reconstructing the entire last-half representation, is considerable and varies from 9% to 55%.

This shows that the approach is interesting not only in computing time, storage space, and representation, but also in query response time and update time. With this approach, the incremental construction of the data cube representation can be more efficient than the entire reconstruction on the updated fact table.

REFERENCES

[1] S. Agarwal et al., “On the computation of multidimensional aggregates”, Proc. of VLDB’96, pp. 506-521.

[2] V. Harinarayan, A. Rajaraman, and J. Ullman, “Implementing data cubes efficiently”, Proc. of SIGMOD’96, pp. 205-216.

[3] J. A. Blackard, “The forest covertype dataset”, ftp://ftp. ics.uci. edu/pub/machine-learning-databases/covtype, [retrieved: April, 2015].

[4] C. Hahn, S. Warren, and J. London, “Edited synoptic cloud reports from ships and land stations over the globe”, http://cdiac. esd.ornl.gov/cdiac/ndps/ndp026b.html, [retrieved: April, 2015].

[5] 2010 Census Modified Race Data Summary File for Counties Alabama through Missouri http://www.census.gov/popest/research/modified/ STCO-MR2010_AL_MO.csv, [retrieved: September, 2016].

[6] Online Retail Data Set, UCI Machine Learning Repository, https://archive.ics.uci.edu/ml/datasets/Online+Retail, Source: Dr Daqing Chen, School of Engineering, London South Bank University, London SE1 0AA, UK, [retrieved: September, 2016].

[7] K. A. Ross and D. Srivastava, “Fast computation of sparse data cubes”, Proc. of VLDB’97, pp. 116-125.

[8] Beyer, K.S., Ramakrishnan, R.: Bottom-up computation of sparse and iceberg cubes, Proc. of ACM Special Interest Group on Management of Data (SIGMOD’99), 359-370.

[9] J. S. Vitter, M. Wang, and B. R. Iyer, “Data cube approximation and histograms via wavelets”, Proc. of Int. Conf. on Information and Knowledge Management (CIKM’98), pp. 96-104.

[10] J. Han, J. Pei, G. Dong, and K. Wang, “Efficient Computation of Iceberg Cubes with Complex Measures”, Proc. of ACM SIGMOD’01, pp. 441-448.

[11] L. Lakshmanan, J. Pei, and J. Han, “Quotient cube: How to summarize the semantics of a data cube,” Proc. of VLDB’02, pp. 778-789.

[12] V. Phan-Luong, “A Simple and Efficient Method for Computing Data Cubes”, Proc. of The 4th Int. Conf. on Communications, Computation, Networks and Technologies INNOV 2015, pp. 50-55.

[13] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, and Y. Kotidis, “Dwarf: shrinking the petacube”, Proc. of ACM SIGMOD’02, pp. 464-475.

[14] W. Wang, H. Lu, J. Feng, and J. X. Yu, “Condensed cube: an efficient approach to reducing data cube size”, Proc. of Int. Conf. on Data Engineering 2002, pp. 155-165.

[15] A. Casali, R. Cicchetti, and L. Lakhal, “Extracting semantics from data cubes using cube transversals and closures”, Proc. of Int. Conf. on Knowledge Discovery and Data Mining (KDD’03), pp. 69-78.

[16] L. Lakshmanan, J. Pei, and Y. Zhao, “QC-Trees: An Efficient Summary Structure for Semantic OLAP”, Proc. of ACM SIGMOD’03, pp. 64-75.

[17] D. Xin, J. Han, X. Li, and B. W. Wah, “Star-cubing: computing iceberg cubes by top-down and bottom-up integration”, Proc. of VLDB’03, pp. 476-487.

[18] Y. Feng, D. Agrawal, A. E. Abbadi, and A. Metwally, “Range cube: efficient cube computation by exploiting data correlation”, Proc. of Int. Conf. on Data Engineering 2004, pp. 658-670.

[19] Z. Shao, J. Han, and D. Xin, “Mm-cubing: computing iceberg cubes by factorizing the lattice space”, Proc. of Int. Conf. on Scientific and Statistical Database Management (SSDBM 2004), pp. 213-222.

[20] Y. Sismanis and N. Roussopoulos, “The complexity of fully materialized coalesced cubes”, Proc. of VLDB’04, pp. 540-551.

[21] K. Morfonios and Y. Ioannidis, “Supporting the Data Cube Lifecycle: The Power of ROLAP”, The VLDB Journal, 2008, 17(4), pp. 729-764.

[22] A. Casali, S. Nedjar, R. Cicchetti, L. Lakhal, and N. Novelli, “Lossless Reduction of Datacubes using Partitions”, In Int. Journal of Data Warehousing and Mining (IJDWM), 2009, Vol 5, Issue 1, pp. 18-35.