Efficient Evaluation of Sparse Data Cubes

By: Lixin Fu

Abstract:

Computing data cubes requires the aggregation of measures over arbitrary combinations of dimensions in a data set. Efficient data cube evaluation remains challenging because of the potentially very large sizes of input datasets (e.g., in the data warehousing context), the well-known curse of dimensionality, and the complexity of queries that need to be supported. This paper proposes a new dynamic data structure called SST (Sparse Statistics Trees) and a novel, interactive, and fast cube evaluation algorithm called CUPS (Cubing by Pruning SST), which is especially well suitable for computing aggregates in cubes whose data sets are sparse. SST only stores the aggregations of non-empty cube cells instead of the detailed records. Furthermore, it retains in memory the dense cubes (a.k.a. iceberg cubes) whose aggregate values are above a threshold. Sparse cubes are stored on disks. This allows a fast, accurate approximation for queries. If users desire more refined answers, related sparse cubes are aggregated. SST is incrementally maintainable, which makes CUPS suitable for data warehousing and analysis of streaming data. Experiment results demonstrate the excellent performance and good scalability of our approach.

Article:

1. Introduction

In the past decade there has been continuous interest by the research community on data warehousing, OLAP (On-line Analytical Processing) and data cubes [5, 6, 9]. How to compute and store data cubes is of particular importance. Since most OLAP queries involve only aggregates in the form of group-bys or cube-bys instead of detailed information, we call these types of queries *cube queries*. These queries compute the aggregates (SUM, COUNT, M1N, MAX, etc.) of measures over an *arbitrary* combination of the dimensions and their hierarchical levels. For example, if a retail warehouse contains three dimensions, time, location, and product, a cube query could be “How many computers are sold in Raleigh, NC and Atlanta, GA between January and March of year 2001?”

Efficient computation of data cube is a fundamental and required function in analytical applications. It forms the basis for generating various reports and for complex data analysis. Moreover, efficient data cube computation has important applications in designing a new bred of data mining algorithms. The new SST data structure exhibits the following important advantages over existing approaches:

- Efficiency: SST only stores aggregations (instead of detail records) and dense data cubes. After initialisation of SST by one pass of data, accurate approximation can be instantly obtained without any I/O operations.
- Interactive: users can refine their query answers or stop evaluation at any time.
- Self-indexing: the dense cubes stored in in-memory SST and the sparse cubes stored on disk are all already sorted automatically. This eliminates the large overhead of indexing the data cubes or views.
- Scalable: CUPS can deal with many dimensions and large domain sizes. It is scalable in large number of records. In addition, CUPS is easily modified to parallel algorithms.

The rest of this paper is organized as follows. The related work is summarized in section 2. In sections 3 and 4, we present the data structure of SST and the cube query processing algorithms. The results of the performance experiments are given in section 5. Concluding remarks are provided in Sec. 6.

2. Related Work

The literature related to OLAP and data cube is rich. To deal with the sparse data, Yihong Zhao et al. proposed the chunking method and sparse data structure for sparse chunks [20]. Sanjay Goil and Alok Choudhary proposed PARS1MONY to parallelize MOLAP for better performance [8]. Also in the MOLAP camp, CubiST (Cubing with Statistics Trees) for the first time computes and maintains all the data cubes including supercubes together in one compact data structure called statistics tree (ST) for dense data [21]. ST is a *static* data structure. Once the dimensions and their domain sizes are given, the tree configuration is determined irrespective to the contents of the records. When ST can fit into memory, CubiST is an excellent choice. But in most real world applications, the requirement that the whole ST fit into memory is unrealistic. CUPS addresses this major drawback of scalability so that it is especially scalable and suitable for sparse data. To optimise the cube queries that have constraints on arbitrary hierarchy levels, [22] selects and materializes a family of statistics trees for dense data. CUPS is interactive, which is a very attractive feature in data exploration and data streaming applications. In addition, in this paper we will address the I/O issues (e.g. paging and matching) that papers [21, 22] did not.

Materialized views are commonly used to speedup cube queries. A greedy algorithm over the lattice structure to choose views for materialization is given [12]. Other views can be computed from them on-the-fly. View maintenance problem is addressed by [14, 10, 19, 16]. Bitmap and B^+-tree are two popular indexing schemes that are used by most systems. Bitmap is suitable for dimensions with small number of values. Encoded bitmap is an improvement for large domain size dimensions [7]. B^+-tree is an indexing structure for dimensions with large cardinalities. B^+-tree is one- dimensional structure while SST is multidimensional.

Another method is to restrict cubing only on group-bys that use HAV1NG COUNT(*) > X, where X is greater than some threshold [4]. K. Beyer and R. Ramakrishnan develop a new algorithm BUC (Bottom-Up Cubing) to solve the new minsup-cube problem. Similar to some ideas in [17], BUC builds the CUBE bottom-up; i.e., it builds the CUBE by starting from a group-by on a single attribute, then on two attributes, and so on. BUC belongs to the ROLAP camp, focusing on the row operations e.g. sorting. External sorting and large intermediate files slow down this type of algorithms. Another major drawback of BUC is that it is not incrementally maintainable.

T. Johnson and D. Shasha [13] propose cube trees and cube forests for cubing. SST differs from cube trees in that it stores all the aggregates in the leaves and internal nodes contain special star pointers. Due to the complexity and long response times, some algorithms give a quick approximation instead of an exact answer that requires much more time. Sampling is often used in estimations [11, 3]. Vitter and Wang use wavelets to estimate aggregates for sparse data [18]. An interesting idea of relatively low cost is to refine self-tuning histograms by using feedback from query execution engine [2]. However, the loss of accuracy in this algorithm is unacceptable for high skewed data.

3. Sparse Statistics Trees

3.1 Tree Structure

SSTs are multi-way trees with k+1 levels, where k is the number of dimensions. Each level corresponds to a dimension except the last level. The root is at level 0. The leaves on the last level (i.e. level k) contain aggregates. The internal nodes are used to direct the access paths for the queries. A node at level h (h = 0, 1, ..., k-1) contains (index, pointer) pairs. The indexes represent the domain values of the corresponding dimension and the pointers direct query paths to the next level. An additional special index value called *star value* represents the ALL value of dimension h; the corresponding *star pointer* is used to direct the path to the next level for a cube query that has no constraints for dimension h.

All leaves are automatically sorted and indexed by the corresponding root-to-leaf paths. SST stores the *full cube* (i.e. containing *all* data cubes) in its leaves if it fits in memory. In the example SST of Fig. 1, there are three dimensions and the domain value indexes are labelled along their corresponding pointers. Leaves shown in boxes contain the aggregation values of the paths from the root to the leaves. Without loss of generality, in this paper we implement COUNT function. Other aggregation functions e.g. SUM needs only minor changes. Data cubes with no or few stars consume most of the storage space but are rarely queried. When main memory is exhausted, these sparse cubes will be cut off and stored on disk. In this sense, SST makes optimal usage of memory space and is used to speed up most frequent queries. Another important observation is that SST represents a *super view*, binding all the views (or arrays) with different combinations of dimensions together into one compact structure. As a MOLAP data structure, SST preserves the advantages of ROLAP in the sense that it only stores nonempty data cubes in summary tables for sparse data.

3.2 Dynamic Generation and Maintenance of the SST Tree

The SST is dynamically generated. While scanning, the input records are inserted into the tree one by one. Whenever a new record is inserted, starting from the root for the first dimension, we first search its index fields. If the index is already there, we simply follow its pointer to the next level node. If not, a new node will be created as its child and a new entry of (index, pointer) pair will be inserted. If necessary, a new child is also created for the star index entry. At the same time we always follow the star pointer to the next level.

We proceed recursively until level k-1 is reached. The SST after inserting the first record (6, 9, 5) and the second record (20, 1, 3) is shown in Fig. 1. The newly created or accessed pointers are shown in dashed arrows. The pseudo-code of inserting one record into SST is shown in Fig. 2 by calling recursive function insert(root, 0, record). To generate SST, we first create a root with an empty list and then repeatedly insert all the input records.

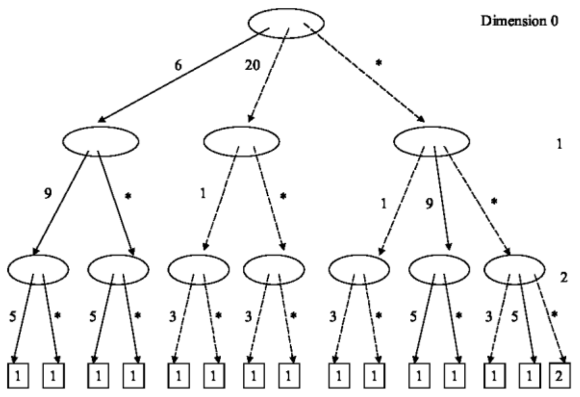


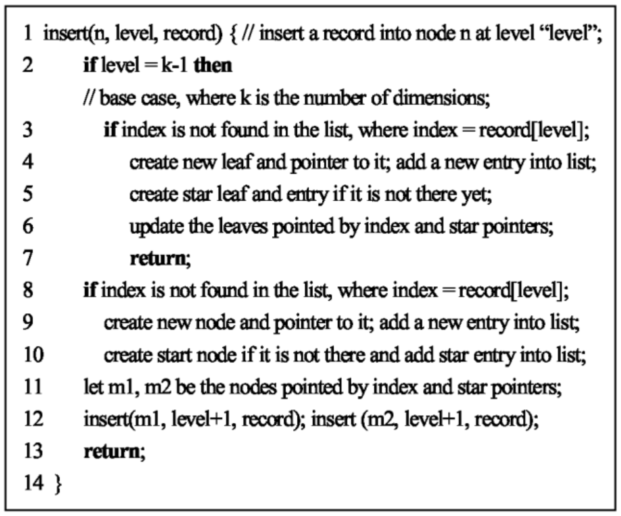
Fig. 1. The SST after inserting first two records.

Even though SST only stores the nonempty aggregates, it may still not fit in memory at some point during the insertion process. Our strategy is to cut sparse leaves off and store them on disks so that they can be retrieved later to refine answers.

3.3 Cutting Sparse Nodes and Generating Runs

The total number of nodes in the SST is maintained during the construction and maintenance. Once it reaches a certain threshold (we say, SST is *full*), a cut phase starts. One idea is to cut off sparse leaves whose COUNT values are less than a threshold *minSupp*.

Continuing the example of Fig. 1, two more records (6, 9, 3) and (20, 9, 3) are inserted (Fig. 3). If we set minSupp to 2, all the nodes and pointers shown in the dashed sub-graphs will be cut. Notice that when a node (e.g., B) has no children (i.e., has an empty list) after cut, it must also be cut and its pair entry in its parent (e.g., A) should be deleted. All memory space resulting from cut is reclaimed for later use. All the leaves being cut are stored on disk in the form of (path, aggregation) pairs, where



```
insert(n, level, record) { // insert a record into node n at level "level";
    if level = k-1 then
    // base case, where k is the number of dimensions;
        if index is not found in the list, where index = record[level];
            create new leaf and pointer to it; add a new entry into list;
            create star leaf and entry if it is not there yet;
            update the leaves pointed by index and star pointers;
            return;
    if index is not found in the list, where index = record[level];
        create new node and pointer to it; add a new entry into list;
        create start node if it is not there and add star entry into list;
    let m1, m2 be the nodes pointed by index and star pointers;
    insert(m1, level+1, record); insert (m2, level+1, record);
    return;
}
```

Fig. 2. Recursive insertion algorithm.

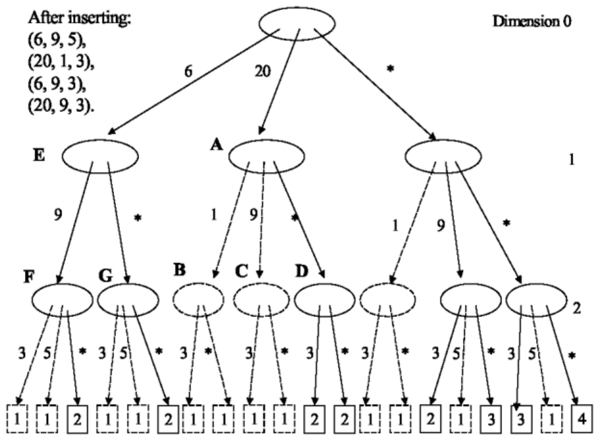


Fig. 3. SST after inserting two more records and before cutting.

path is the root-to-leaf index combination and aggregation in our example is the COUNT value. All the pairs resulting from the same cut phase form a *run*, which are naturally sorted by paths.

4. Evaluation of Ad-Hoc Cube Queries

4.1 Approximate Query Evaluation through SST

From any cube query over k dimensions, we extract the constrained domain values and store them using k vectors v_0, v_1, ..., $v_{k-1,}$ each for a dimension. If dimension i is absent, v_i contains one single ALL value. We call these vectors *selected value sets* (SVS) of the query. Starting from the root of SST, our algorithm follows along the paths directed by the SVS until it will encounter related leaves. If a path stops at a node n at level s where the sth domain value is not in the index list of n, the algorithm returns 0 because there is no dense cube for this path. The aggregation of the visited leaves is returned to users as an approximation. We discuss the methods for refining query answers in the next subsection.

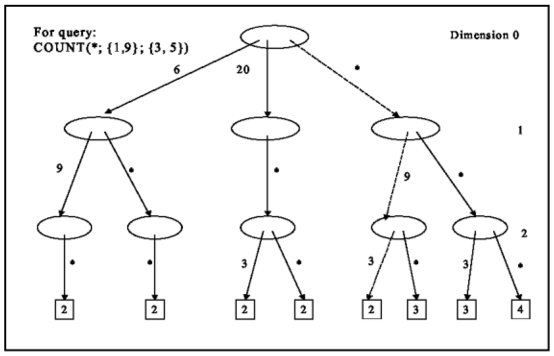


Fig. 4. The SST after the cut phase.

In our running example, suppose a cube query COUNT (*; {1, 9}; {3, 5}) is submitted and the SST shown in Fig. 4 is used to evaluate the query. We first compute its SVS: $v_0 = \{*\}$, $v_1 = \{1, 9\}$, and $v_2 = \{3, 5\}$. Starting from the root, follow the star pointer to the next level node. The path stops because the first domain value of v_1 (=1) is not in the list. The query result is still 0. Check the second value of v_2 and follow the pointer corresponding to its index 9. Further tracking along the pointer for 3 leads to the fall-off leaf (the matched path *-9-3 is shown in dashed lines). After adding it up, the result becomes 2. Following the pointer of the next value of v_3 does not lead to a match. A quick approximate query result of 2 is returned. Next, we can refine the result by matching the paths with the runs generated in Sec. 3.3. The result is updated from 2 to 3 (after aggregating path *-1-3 in the run), and finally to the exact answer 4 (after aggregating path *-9-5).

4.2 Interactively Refining Query Answers

When the run files are large and the queries involve a large number of data cubes, retrieving and aggregating the related sparse cubes are nontrivial. The cubes are accessed by the paths defined by $v_0 \times v_1 \times \ldots \times v_{k-1}$ from SVS. We store them in a vector *Q_cube*. For each run, refining query answers becomes a problem of matching the cubes in *Q_cube* with the (path, value) pairs stored in the run file and aggregate those matched cubes as the refinement for this run. Of course, it is inefficient to scan the run file for matches with *Q_cube*. Instead, we segment the run file into pages and only retrieve the matched pages. During the run generation, a *pageTable* is created and stored in memory. The entries of *pageTable* are the cube paths of the first cube of each page.

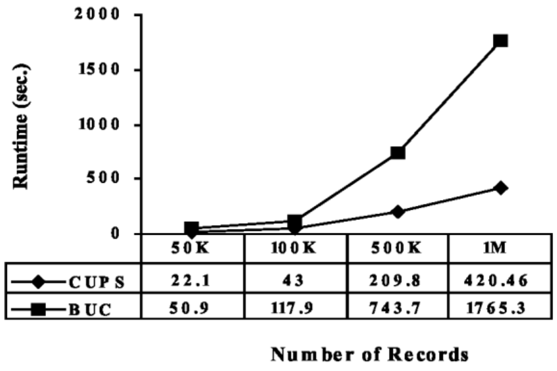
5. Simulation Results

5.1 Setup for Simulation

We have conducted comprehensive experiments on the performance of our CUPS algorithms by varying the number of records, and the degrees of data skew in a comparison with BUC. We decided to use BUC since the performance of BUC was better than its closest competitor MemoryCube. We have implemented BUC based on a description given in the paper [4]. However, this paper only implements the internal BUC. That is, it assumes that all the partitions are in the memory. Obviously, this is not adequate since in real world applications the space for the partitions and other intermediate results easily exceed the common available memory. Therefore, to investigate its overall behaviour including I/O operations we implement both internal BUC and external BUC. All experiments were conducted on a Dell Precision 330 with 1.7GHZ CPU, 256MB memory, and the Windows 2000 operating system.

5.2 Varying Number of Records

We first use uniformly distributed random data over five dimensions with cardinality of 10 each. The number of records increases from 50, 000 to 1,000,000 (data set sizes from 1megabytes to 20 megabytes). Correspondingly, the data density (defined as the average number of records per cell) increases from 0.5 to 10. The threshold *minSupp* and available memory are set to 2 and 10 megabytes respectively. The pattern of runtimes is the same for larger parameters and data sets. The runtimes are shown in Fig. 5. CUPS is about 2-4 times faster than BUC. The runtimes are the times for computing the data cubes.

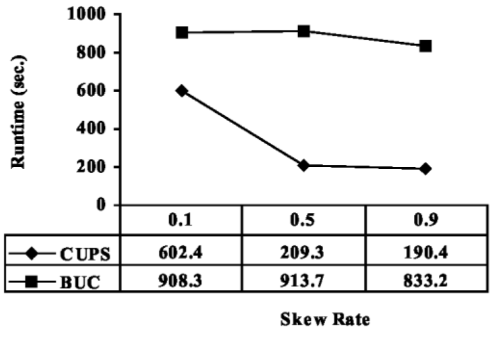


	50K	100K	500K	1M
CUPS	22.1	43	209.8	420.46
BUC	50.9	117.9	743.7	1765.3

Fig. 5. Varying number of records.

5.3 Varying Degrees of Data Skew

In this set of experiments, instead of using uniform data sets we change the degrees of data skew. Real world data sets are often highly skewed. For example, the number of computers sold in Chicago, 1L last month was probably much higher than in Greensboro, NC. Our data sets all contain 100,000 records and have 5 dimensions with the same size of 20. The average data density is 1/32.



	0.1	0.5	0.9
CUPS	602.4	209.3	190.4
BUC	908.3	913.7	833.2

Fig. 6. Varying degrees of data skew.

To control the degree of data skew, we use a measure *skew_rate* which is the percentage of input records that have values in 0 - *skew—max* for *all* dimensions, where *skew_max* is a parameter (we fixed it to 5 in the experiments). The rest of records have random values within domain ranges (0-19 in our example). The larger *skew—rate* and the smaller *skew_max* are, the more skewed the data set is. The test results of using *skew_rate* from 0.1 to 0.9 are show in Fig. 6. For very sparse and skewed data sets, the performance of CUPS is significantly better than BUC. Although the data is very sparse, some regions and subspaces are still dense.

6. Conclusion

In this paper, we presented a new data structure called SST. Based on SST, new cubing algorithm CUPS has been given. This method deals with data sparseness by only storing nonempty cubes and retaining only dense cubes in memory. Duplicate records are aggregated into leaves without storing the original data even the record Ids. For complex relational databases with high dimensionality and large domain sizes, to free more space, the algorithm dynamically cuts out high-dimensional sparse units that are rarely or never queried. Our optimal one-pass initialization algorithm and internal query evaluation algorithm ensure fast set-up and instant responses to the very complex cube queries. Our paging and "zigzag" matching techniques speedup the query refinement computation. Comparisons with the BUC algorithms have confirmed the benefits and good scalability of CUPS.

References

1. S. Agarwal, R. Agrawal, P. Deshpande, J. Naughton, S. Sarawagi, and R. Ramakrishnan. On the Computation of Multidimensional Aggregates. In *Proceedings of the International Conference on Very Large Databases (VLDB'96),* Mumbai (Bomabi), India, 1996.
2. Ashraf Aboulnaga and Surajit Chaudhuri. Self-tuning histograms: building histograms without looking at data. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data (SIGMOD '99),* Philadelphia, PA, pages 181-192, June 1999.
3. Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala. Congressional samples for approximate answering of group-by queries. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD '00),* Dallas, TX, pages 487-498, May 2000.
4. Kevin Beyer, Raghu Ramakrishnan. Bottom-up computation of sparse and Iceberg CUBE. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data (SIGMOD '99),* Philadelphia, PA, pages 359-370, June 1999.
5. E. F. Codd, S. B. Codd, and C. T. Salley. Providing OLAP (on-line analytical processing) to user-analysts: An IT mandate. *Technical Report*, www.arborsoft.com/OLAP.html.
6. S. Chaudhuri and U. Dayal. An Overview of Data Warehousing and OLAP Technology, *SIGMOD Record*, **26**(1): 65-74, 1997.
7. C. Y. Chan and Y. E. Ioannidis. Bitmap Index Design and Evaluation. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data (SIGMOD '98)*, Seattle, WA, pages 355-366, 1998.
8. Sanjay Goil, Alok N. Choudhary. PARSIMONY: An Infrastructure for Parallel Multidimensional Analysis and Data Mining. *Journal of Parallel and Distributed Computing* 61(3): 285-321, 2001.
9. 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 Sub-Totals, *Data Mining and Knowledge Discovery*, **1**(1): 29-53, 1997.
10. A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query Processing in Data Warehousing Environments. In *Proceedings of the Eighth International Conference on Very Large Databases (VLDB '95),* Zurich, Switzerland, pages 358-369, 1995.

11. Phillip B. Gibbons, Yossi Matias. New sampling-based summary statistics for improving approximate query answers. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data (SIGMOD '98)*, Seattle, WA, pages 331-342, 1998.
12. V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, **25**(2): 205-216, 1996.
13. T. Johnson and D. Shasha, "Some Approaches to Index Design for Cube Forests," *Bulletin of the Technical Committee on Data Engineering, IEEE Computer Society*, **20**:1, pp. 27-35, 1997.
14. D. Lomet, *Bulletin of the Technical Committee on Data Engineering*, **18**, IEEEE Computer Society, 1995.
15. Wolfgang Lehner, Richard Sidle, Hamid Pirahesh , Roberta Wolfgang Cochrane. Maintenance of cube automatic summary tables. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD '00),* Dallas, TX, pages 512-513, May 2000.
16. Inderpal Singh Mumick, Dallan Quass, Barinderpal Singh Mumick: Maintenance of Data Cubes and Summary Tables in a Warehouse. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data (SIGMOD '97),* Tucson, AZ, pages 100- 111, 1997.
17. Ramakrishnan Srikant, Rakesh Agrawal. Mining quantitative association rules in large relational tables. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data (SIGMOD '96),* Montreal, Canada, pages 1-12, 1996.
18. Jeffrey Scott Vitter, Min Wang. Approximate computation of multidimensional aggregates of sparse data using wavelets. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data (SIGMOD '99),* Philadelphia, PA, pages 193-204, June 1999.
19. W. P. Yan and P. Larson. Eager Aggregation and Lazy Aggregation. In *Proceedings of the Eighth International Conference on Very Large Databases (VLDB '95)*, Zurich, Switzerland, pages 345-357, 1995.
20. Y. Zhao, P. M. Deshpande, and J. F. Naughton. An Array-Based Algorithm for Simultaneous Multidimensional Aggregates. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, **26**(2): 159-170, 1997.
21. Fu, L. and Hammer, J. CUBIST: A New Algorithm For Improving the Performance of Ad-hoc OLAP Queries. in *ACM Third International Workshop on Data Warehousing and OLAP, Washington, D. C, USA, November*, 2000, 72-79.
22. Hammer, J. and Fu, L. Improving the Performance of OLAP Queries Using Families of Statistics Trees. in *3rd International Conference on Data Warehousing and Knowledge Discovery DaWaK 01, September, 2001, Munich, Germany*, 2001, 274-283.