ISSN No. 0976-5697

Volume 2, No. 1, Jan-Feb 2011

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

Comparative study on the Mining Iceberg Cubes Algorithms

Dr. Pankaj Nagar
Department of Statistics
University of Rajasthan, Jaipur
Rajasthan(India)
pnagar121@gmail.com

Sheelesh Kumar Sharma*
Associate Professor
I.M.S. Ghaziabad
U.P. India
sheelesh@mail.com

Dr. V.P. Gupta
Professor
K.I.E.T.-Ghaziabad
U.P. India
vpguptajp@yahoo.com

***Abstract*:** A data warehouse is a collection of data for supporting of decision making process. Data cubes and on-line analytical processing (OLAP) have become very popular techniques to help users analyze data in a warehouse. An iceberg cube consists of only the set of group-bys whose aggregates are no less than a user-specified aggregate threshold, and does not compute a complete cube. Mining iceberg cubes is an important research problem in both online analytic processing (OLAP) and data mining. It can be to answer group-by queries, mine multidimensional association rules, and identify interesting subsets of the cube for pre-computation. A data warehouse is often organized in a schema of multiple tables, such as star schema or snowflake schema, in practice. Several algorithms, such as BUC, MultiWay (Y. Zhao et al., 1997), H-Cubing (Han et al., 2001), and Star-Cubing (Xin et al., 2003), have been proposed to compute iceberg cubes from data warehouse. This paper focus on comparative study in respect to the performance of all above mentioned algorithms. Researcher finds that MultiWay and H-Cubing do not perform well in high dimension and high cardinality datasets. Performance study demonstrates that Star-Cubing is a promising method.

Keyword: Mining, Algorithms, data warehouse, online analytic processing

I. INTRODUCTION

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 aggregate condition could be, for example, minimum support or a lower bound on average, min or max. 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. For a three dimensional data cube, with attributes A, B and C, the Iceberg-Cube problem may be represented as:

SELECT a, B, C, Count (*), SUM (X) FROM Table Name CUBE BY A, B, C
HAVING COUNT (*)>=minsup

Where minsup is the minimum support. Minimum support is the minimum number of tuples in which a combination of attribute values must appear to be considered **frequent**. It is expressed as a percentage of the total number of tuples in the input table.

When an Iceberg-Cube is constructed, it may also include those totals from the original cube that satisfy the minimum support requirement. The inclusion of totals makes Iceberg-Cubes more useful for extracting previously unknown relationships from a database. For example, suppose minimum support is 25% and we want to create an Iceberg-Cube using Table 1 as input. For a combination of attribute values to appear in the Iceberg-Cube, it must be present in at least 25% of tuples in the input table, or 2 tuples. The resulting Iceberg-Cube is shown in Table 2. Those cells in the full cube whose counts are less than 2, such as ({P1, L1, C1},1), are not present in the Iceberg-Cube.

Table 1: Sample Database Table

Part	StoreLocation	Customer
P1	L1	C1
P1	L2	C2
P1	L3	C3
P2	L3	C4
P2	L3	C4
P2	L3	C5
P2	L4	C4
P3	L5	C5

Table 2: Sample Iceberg-Cube with Minimum Support of at least 25%

Combination	Count
{P1, ANY, ANY}	3
{P2, ANY, ANY}	4
{ANY, L3, ANY}	4
{ANY, ANY, C4}	3
{P2, L3, ANY}	3
{P2, ANY, C4}	3
{ANY, L3, C4}	2
{P2, L3, C4}	2

II. ALGORITHMS FOR MINING ICEBERG-CUBES

A. Apriori

The APRIORI algorithm uses **candidate combinations** to avoid counting every possible combination of attribute values. For a combination of attribute values to satisfy the minimum support requirement, all subsets of that combination must also satisfy minimum support. The candidate combinations are found by combining only the frequent attribute value combinations that are already known. All other possible combinations are automatically eliminated because not all of their subsets would satisfy the minimum support requirement.

To do this, the algorithm first counts all single values on one pass of the data, then counts all candidate combinations of the frequent single values to identify frequent pairs. On the third pass over the data, it counts candidate combinations based on the frequent pairs to determine frequent 3-sets, and so on. This method guarantees that all frequent combinations of k values will be found in k passes over the database.

B. MultiWay

MultiWay is an array-based top-down cubing algorithm. It uses a compressed sparse array structure to load the base cuboid and compute the cube. In order to save memory usage, the array structure is partitioned into chunks. It is unnecessary to keep all the chunks in memory since only parts of the group-by arrays are needed at any time. By carefully arranging the chunk computation order, multiple cuboids can be computed simultaneously in one pass. The MultiWay algorithm is effective when the products of the cardinalities of the dimensions are moderate. If the dimensionality is high and the data is too sparse, the method becomes infeasible because the arrays and intermediate results become too large to fit in memory. Moreover, the top-down algorithm cannot take advantage of Apriori pruning during iceberg cubing, i.e., the iceberg condition can only be used after the whole cube is computed.

Top-down computation (tdC) computes an Iceberg-Cube by traversing down a multi-dimensional lattice formed from the attributes in an input table. The lattice represents all combinations of input attributes and the relationships between those combinations. Figure 1 shows a 4-Dimensional lattice of this type. The processing path of tdC is shown in Figure 2. The algorithm begins by computing the frequent attribute value combinations for the attribute set at the top of the tree, in this case ABCD. On the same pass over the data, tdC counts value combinations for ABCD, ABC, AB and A, adding the frequent ones to the Iceberg-Cube. This is facilitated by first ordering the database by the current attribute combination, ABCD. tdC then continues to the next leaf node in the tree, ABD, and counts those attribute value combinations. For n attributes, there are 2^{n-1} possible combinations of those attributes, which are represented as the leaf nodes in the tree. If no pruning occurs, then tdC examines every leaf node, making 2^{n-1} passes over the data. Pruning can occur when no attribute value combinations are found to be frequent for a certain combination of attributes.

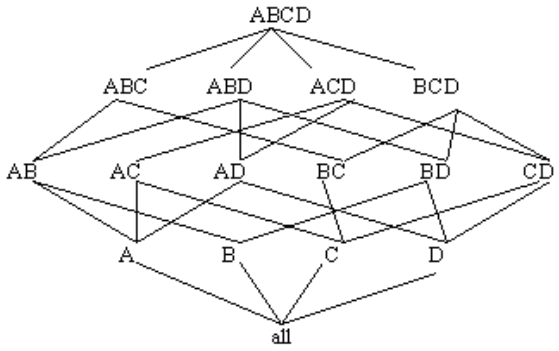

Figure 1: 4-Dimensional Lattice

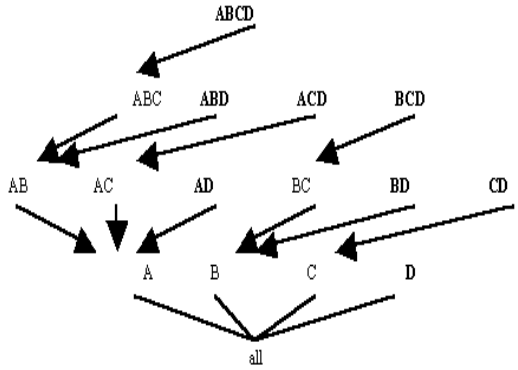

Figure 2: Processing Tree of tdC for Four Attributes

C. Bottom-Up Computation

The bottom-up computation algorithm (BUC) repeatedly sorts the database as necessary to allow convenient partitioning and counting of the combinations without large main memory requirements. BUC begins by counting the frequency of the first attribute in the input table. The algorithm then partitions the database based on the frequent values of the first attribute, so that only those tuples that contain a frequent value for the first attribute are further examined. BUC then counts combinations of values for the first two attributes and again partitions the database so only those tuples that contain frequent combinations of the first two attributes are further examined, and so on.

For a database with four attributes, A, B, C and D, the processing tree of the BUC algorithm is shown in Figure 3. As with the tdC processing tree, this is based on the 4-dimensional lattice from Figure 1. The algorithm examines attribute A, then partitions the database by the frequent values of A, sorting each partition by the next attribute, B, for ease of counting AB combinations. Within each partition, BUC counts combinations of attributes A and B. Again, once the frequent combinations are found, the database is partitioned, this time on frequent combinations of AB, and is sorted by attribute C. When no frequent combinations are found, the algorithm traverses back down the tree and ascends to the next node. For example, if there are no frequent combinations of AC, then BUC will examine combinations of AD next. In this way, BUC prunes passes over the data. As with the tdC algorithm, BUC prunes most efficiently when the attributes are ordered from highest to lowest cardinality.

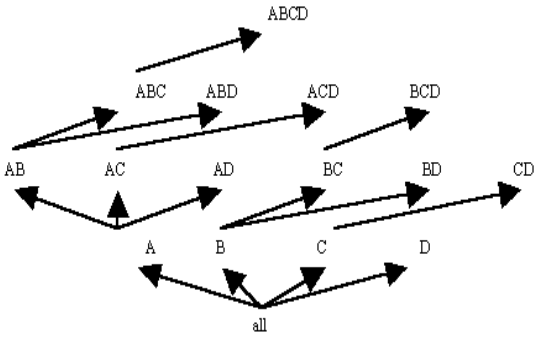

Figure 3: Processing Tree of BUC for Four Attributes

D. H-Cubing

H-cubing (Han et al., 2001) uses a hyper-tree data structure called H-tree to compress the base table. Then, the H-tree can be traversed bottom-up to compute iceberg cubes. It also can prune unpromising branches of search using monotonic iceberg conditions. oreover, a strategy was developed by Han et al. (2001) to use weakened but monotonic conditions to approximate non-monotonic conditions to compute iceberg cubes.

The strategies of pushing non-monotonic conditions into bottom-up iceberg cube computation were further improved by K. Wang et al. (2003). A new strategy, divide-and-approximate, was developed. The general idea is that the weakened but monotonic condition can be made up for each sub-branch search and thus the approximation and pruning power can be stronger.

E. Star-Cubing

Xin et al. (2003) developed **Star-Cubing** by extending H-tree to Star-Tree and integrating the top-down and bottom-up search strategies. Feng et al. (2004) proposed another interesting cubing algorithm, Range Cube, which uses a data structure called range trie to compress data and identify correlation in attribute values. All of the previous studies on computing iceberg cubes make an implicit assumption: a universal base table is materialized. However, this assumption may not be always true in practice – many data warehouses are stored in tens or hundreds of tables. It is often unaffordable to compute and materialize a universal base table for iceberg cube computation. This observation motivates the study in this section.

III. COMPARISON OF THE ALGORITHMS

A. Performance Analysis

All the four algorithms were coded using C++ on Desktop 1.4GHz system with 512MB of RAM. The times recorded include both the computation time and the I/O time. For the remaining of this section, D denotes the number of dimensions, C the cardinality of each dimension, T the number of tuples in the base cuboid, M the minimum support level, and S the skew or zipf of the data. When S equals 0.0, the data is niform; as S increases, the data is more skewed. S is applied to all the dimensions in a particular data set.

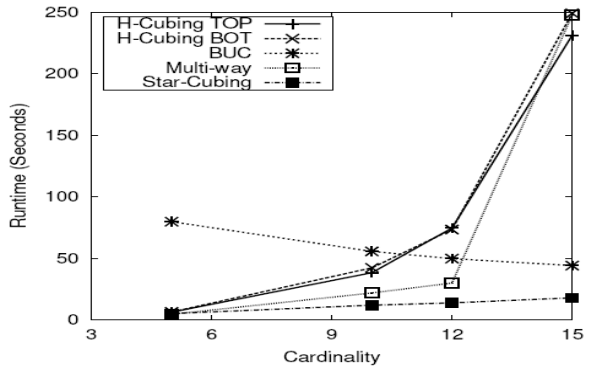

Figure 4: Iceberg Cube Computation w.r.t. Cardinality, where T = 1M, D= 7, S=0, M=1000.

Experiments compare the four algorithms for iceberg cube computation. Except MultiWay, all the algorithms tested use some form of pruning that exploits the anti-monotonicity of the count measure.

We compared BUC and Star-Cubing under high dimension and high cardinality individually. The results are shown in Figures 4. The data set used in Figure 4 had 1000K tuples with 7 dimensions and 0 skew. The min_sup was 1000. The cardinality of each dimension was increased from 5 to 15. We can see that BUC and Star-Cubing performed better in sparse data.

We further compared these two algorithm with higher dimension and cardinality. In Figure 5, the data set had 1000K tuples with 10 dimensions, each with cardinality of 10. The skew of data was 0. At the point where min_sup is 1000,

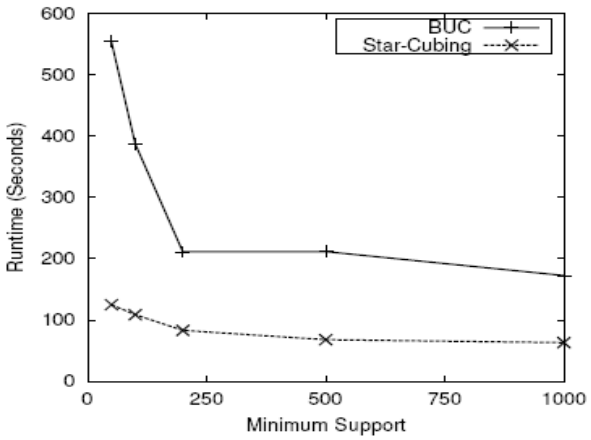

Figure 5: Star-Cubing vs. BUS w.r.t. Minsup, where T=1M, D=10, C=10, S=0.

Star-Cubing decreases the computation time more than 50% comparing with BUC. The improvements in performance get much higher when the min_sup level decreases. Star-Cubing runs around 5 times faster than BUC. The I/O time no longer dominates the computation here.

Figure 6 shows the performance comparison with increasing cardinality.

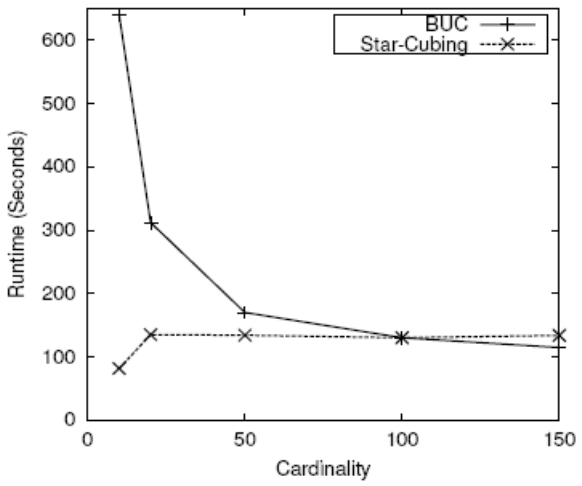

Figure 6: Star-Cubing vs. BUS w.r.t. Cardinality, where T = 1M, D= 10, S=1, M=100.

Star-Cubing is not sensitive to the increase of cardinality; however, BUC improves its performance in high cardinality due to sparser conditions. Although a sparser cube enables Star-Cubing to prune earlier, the star tree is getting wider. The increase in tree size requires more time in construction and traversal, which negates the effects of pruning. We suggest switching from Star-Cubing to BUC in the case where the product of cardinalities is reasonably large compared to the tuple size.

In this experiment, for 1000K tuple size, 10 dimensions, and minimum support level of 100, if data skew is 0, the algorithm should switch to BUC when cardinality for each dimension is 40, if data skew is 1 (shown in Figure 6), the switching point is 100.

The reason that the switching point increased with data skew is that skewed data will get more compression in star-tree, thus will achieve better performance. We will show more detailed experiments in the next section.

Table: 3 Summary of the Algorithms

Algorithm	**Simultaneous Aggregation**	**Partition and Prune**
Apriori	No	Yes
MultiWay	Yes	No
BUC	No	Yes
H-Cubing	Weak	Yes
Star-Cubing	Yes	Yes

IV. FINDING AND SUGGESTION

During this study researcher found the following findings and suggestion.

A. MultiWay and H-Cubing do not perform well in high dimension and high cardinality datasets.
B. Star-Cubing decreases the computation time more than 50% comparing with BUC.
C. Star-Cubing runs around 5 times faster than BUC.
D. Star-Cubing is not sensitive to the increase of cardinality; however, BUC improves its performance in high cardinality due to sparser conditions.
E. We suggest switching from Star-Cubing to BUC in the case where the product of cardinalities is reasonably large compared to the tuple size.

V. CONCLUSION

Performance study demonstrates that Star-Cubing is a promising method. For the full cube computation, if the dataset is dense, its performance is comparable with MultiWay, and is much faster than BUC and H-Cubing. If the data set is sparse, Star-Cubing is significantly faster than MultiWayand H-Cubing, and faster than BUC, in most cases. For iceberg cube computation, Star-Cubing is faster than BUC, and the speedup is more when the min sup decreases. Thus Star-Cubing is the only cubing algorithm so far that has uniformly high performance in all the data distributions.

VI. REFERENCES

[1] Zhao, Y., Deshpande, P. M., & Naughton, J. F. (1997). An array-based algorithm for simultaneous multidimensional aggregates. In Sigmod '97: Proceedings of the 1997 acm sigmod international conference on management of data (pp. 159–170). New York, NY, USA: ACM Press.

[2] Han, J., Pei, J., Dong, G., & Wang, K. (2001, May). Efficient computation of iceberg cubes with complex measures. In Sigmod '01: Proceedings of the 2001 acm sigmod internationalconference on management of data (pp. 1–12). Santa Barbara, California.

[3] Xin, D., Han, J., Li, X., & Wah, B. W. (2003). Star-cubing: Computing iceberg cubes by topdown and bottom-up integration. In Vldb '03: Proceedings of the 2003 vldb internationalconference on very large data bases.

[4] Feng, Y., Agrawal, D., Abbadi, A. E., & Metwally, A. (2004). Range cube: Efficient cube computation by exploiting data correlation. In Icde'04: Proceedings of the 2004 ieee international conference on data engineering (p. 658-670).