Efficient Computing Star Cubes by Top-Down and Bottom-Up Integration

Beena Mahar
MSC (CS), MCA, M Phil (CS)
Assistant Professor IT (IITM, D-29 Janak Puri Delhi)
Ph: 8800150356(M)
beenamahar@gmail.com

ABSTRACT
Data cube computation is essential task in data warehouse implementation. The precomputation of all or part of a data cube can greatly reduce the response time and enhance the performance of on-line analytical processing. There are several methods for cube computation, several strategies to cube materialization and some specific computation algorithms, namely Multiway array aggregation, BUC, C. Star Cubing the computation of shell fragments, and the computation of cubes involving complex measures.
In this paper, I compute the Star-Cube[2] that integrates the strengths of the previous three algorithms and performs aggregations on multiple dimensions simultaneously. It utilizes a star-tree structure, extends the simultaneous aggregation methods, and enables the pruning of the group-by's that do not satisfy the iceberg condition. Our performance study shows that Star-Cubing is highly efficient and outperforms all the previous methods in almost all kinds of data distributions.

KEYWORDS
Data Cubes, Data Cube Algorithms, Star Cubing, MM Cubing, C, Multiway Array Aggregation, BUC

1. INTRODUCTION
Star- cubing that integrates the strengths of the algorithms and performs aggregations on multiple dimensions simultaneously. It utilizes a star-tree structure, extends the simultaneous aggregation methods, and enables the pruning of the group-by's that do not satisfy the iceberg condition.
The problem of cube computation can be defined as follows.
In an n-dimension data cube, a cell a = (a1, a2, ……, An, c) (where c is a measure) is called an m-dimenisonal cell (i.e., a cell in an m-dimensional cuboid), if and only if there are exactly m (m < n) values among {a1, a2, ……, an} which are not *. It is called a base cell (i.e., a cell in a base cuboid) if m = n, otherwise, it is an aggregate cell. Given a base cuboid, our task is to compute an iceberg cube, i.e., the set of cells which satisfies an iceberg condition, or the full cube if there is no such condition. Measure c is the count of base cells, and min sup (threshold) is the iceberg condition

Description
Star-Cubing, which integrates the top-down and bottom-up cube computation and explores both multi-dimensional aggregation and the Apriori pruning. A new data structure, star-tree, is introduced that explores lossless data compression and prunes unpromising cells using an Apriori-like dynamic subset selection strategy. The Star-Cubing[2] algorithm explores both the top-down and bottom-up models: On the global computation order, it uses the top-down model similar to Figure1. However, it has a sub-layer underneath based on the bottom-up model by exploring the notion of shared dimension. This integration allows the algorithm to aggregate on multiple dimensions while still partition parent group-by's and prune child group-by's that do not satisfy the iceberg condition.

Fig a): Top-Down Computation

Fig b): Bottom up Computation
Figure, shows the spanning tree marked with the shared dimensions. For example, ABD/AB means cuboid ABD has shared dimension AB, ACD/A means cuboids ACD has shared dimension A, and so on.
Fig c): Star cubing: Top-down computation with bottom-up growing shared dimensions [3]

Shared Dimensions: An observation of Figure 1 may disclose an interesting fact: all the cuboids in the left-most sub-tree of the root include dimensions ABC, all those in the second sub-tree include dimensions AB, and all those in the third include dimension A. Call these common dimensions the shared dimensions of those particular sub-trees. Based on this concept, Figure 1 is extended to Figure 3, which shows the spanning tree marked with the shared dimensions. For example, ABD/AB means cuboid ABD has shared dimension AB, ACD/A means cuboids ACD has shared dimension A, and so on.

The introduction of shared dimensions facilitates shared computation. Since the shared dimension is identified early in the tree expansion, there is no need to compute them later. For example, cuboid AB extending from ABD in Figure a is pruned in Figure c because AB was already computed in ABD/AB. Also, cuboid A extending from AD is pruned because it was already computed in ACD/A.

Star-Cubing Algorithm
The algorithm derives the complete and correct iceberg cube with the input table R, and the iceberg condition, min sup. The Star-Cubing algorithm is summarized as follows.

Algorithm (Star-Cubing). Compute iceberg cubes by Star-Cubing.

Input: (1) A relational table R, and (2) an iceberg condition, min sup. The Star-Cubing algorithm is summarized as follows.

Output: The computed iceberg cube.

Method: Each star-tree corresponds to one cube-tree node, and vice versa. The algorithm is described in Fig a).

BEGIN scan R twice, create star-table S and star-tree T,
output count of T . root,
call starcubing(T, T . root),
END

procedure starcubing(T, cnode)/ cnode: current node f
1. for each non-null child C of T's cube-tree
2. insert or aggregate cnode to the corresponding position or node in C's star-tree,
3. if (cnode.count > min sup) {
4. if (cnode not equal root)
5. output cnode.count,
6. if (cnode is a leaf)
7. output cnode.count,
8. else { // initiate a new cube-tree
9. create CC as a child of T's cube-tree,
10. let TC as CC's star-tree,
11. TC.root’s count = cnode.count,
12. }
13. }
14. if (cnode is not a leaf)
15. call starcubing(T, cnode: first child),
16. if (CC is not null) {
17. call starcubing( TC, TC.root),
18. remove CC from T's cube-tree, }
19. if (cnode has sibling)
20. call starcubing(T, cnode . sibling),
21. remove T,
}

Fig D): The Star-Cubing algorithm
The efficiency of the algorithm is based on three major points:
- It uses iceberg pruning. With a tree structure, each node in the base tree is a potential root of child tree. The aggregate value of that root can be tested on the iceberg condition and unnecessary aggregates are avoided.
- It explores the multi-way tree aggregation. By scanning base tree once, it aggregates value on multiple children trees.
- It uses star-tree compression. The algorithm explores the star-nodes under the iceberg threshold and builds star-table for each tree. The star-nodes make tree shrink quickly. Thus both computation time and memory requirement are reduced.

2. PERFORMANCE STUDY
To check the efficiency and scalability of the proposed algorithm, a comprehensive performance study is conducted by testing our implementation of Star-Cubing against the best implementation it can achieve for the other three algorithms: Multi Way, BUC, and H-Cubing

1. Full Cube Computation
The first set of experiments compare Star-Cubing with all the other three algorithms for full cube computation. The performance of the other algorithms are compared with respect to tuple size Fig e) (Figure 10), cardinality Fig e) (Figure 11) and dimension Fig e) (Figure 12).
In the first experiment, randomly generated data sets with 5 dimensions, varying the number of tuples from 1000K to 1500K. In the second experiment, we varied the cardinalities for each dimension from 5 to 35. Finally, increased dimension number from 3 to 7 while keeping the cardinality of each at 10. The tuple size for latter two datasets was 1000K. All the data were uniformly distributed, i.e., skew was 0. The experimental results are shown in Figures 10-12. Do not use more dimensions and greater cardinality because in high dimension and high cardinality datasets, the output of full cube computation gets extremely large, and the output I/O time dominates the cost of computation. Moreover, the existing curves have clearly demonstrated the trends of the algorithm performance with the increase of dimensions and cardinality.

There are three main points that can be taken from these results:

First, Star-Cubing and MultiWay are both promising algorithms under low dimensionality, dense data, uniform distribution, and low minimum support. In most cases, Star-Cubing performs slightly better than MultiWay. The performance of MultiWay degraded quickly when dimension increased.

Second, in those cases, BUC showed the worst performance. BUC was initially designed for sparse data set. For dense data, the cost of partition is high, and the overall computation time increases.

Third, the two H-Cubing algorithms performed progressively worse as cardinality increased. This is because when cardinality is high, the H-Tree built from the initial data is wider and traversal on the H-Tree to maintain the links costs more time. Although Star-Cubing uses a similar tree structure as H-Tree,
Star-Cubing generates sub-trees during the computation and the tree sizes are shrinking quickly.

2. Iceberg Cube Computation [2]
The second set of experiments compare the others algorithms for iceberg cube computation. Except Multi-Way, all the algorithms tested use some form of pruning that exploits the anti-monotonicity of the count measure.

As seen in the previous experiments, both MultiWay [4] and H-Cubing do not perform well in high dimension and high cardinality datasets.

Figure 13: Iceberg Cube Computation w.r.t. Cardinality, where $T = 1M$, $D = 7$, $S = 0$, $M = 1000$

Figure 14: Star-Cubing vs. BUC w.r.t. MinSup, where $T = 1M$, $D = 10$, $C = 10$, $S = 0$

Figure 15: Star-Cubing vs. BUC w.r.t. Cardinality, where $T = 1M$, $D = 10$, $S = 1$, $M = 100$

Fig F): Ice Cube Computation

Now compared BUC and Star-Cubing under high dimension and high cardinality individually. The results are shown in Fig f)(Figures 13 – 15). The data set used in Fig f)(Figure 13) 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. It can see that BUC and Star-Cubing performed better in sparse data. In Fig f)(Figure 14), 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, Star-Cubing decreases the computation time more than 50% comparing with BUC. The improvements in performance get much higher when the min sup level de-creases. For example, when min sup is 50, Star-Cubing runs around 5 times faster than BUC.

The I/O time no longer dominates the computation here Fig f)(Figure 15) shows the performance comparison with increasing cardinality. 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. Its suggest switching from Star-Cubing to BUC in the case where the product of cardinalities is reasonably large compared to the tuple size. In our 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 Fig f)(Figure 15)), the switching point is 100.
3. Data Skew
The skewness affects the performance of the algorithms. Here use Zipf to control the skew of the data, varying Zipf from 0 to 3 (0 being uniform). The input data had 1000K tuples, 10 dimensions, and cardinality of 10 for each dimension, and the min sup was 100. Fig g) (Figure 16) shows the computation time for the four algorithms. Skewed data made MultiWay, H-Cubing and Star-Cubing perform better. BUC is the only one that degrades. MultiWay improved because many chunked arrays now hold a zero count while other chunks hold a very big count. The array indices with zero count do not need to be processed at all while the bigger counts do not increase the

Fig G): Data Skew
workload to Multi-Way. The two H-Cubing algorithms in Fig g)(Figure 16) start performing much better once S was around 1:5. This can be explained by the size of the H-Tree. With skewed data, each node in the H-Tree is further reduced in size because not all values in each dimension appear now. So as S increases, the H-Tree grew thinner. Similarly, skewed data also makes the star-tree thinner and thus achieve better performance. We also compared BUC with Star-Cubing in sparse dataset in Fig g)(Figure 17). The result is similar to Fig g)(Figure 16).

BUC’s performance degraded with increased skew while Star-Cubing improved. Even if the duplicate collapsing code was added to BUC (BUC-Dedup), BUC still degraded until the duplications compensated for the loss of pruning. Finally, Figure 18 shows the memory usage of Star-Cubing comparing with the original data size.

4. Additional Star-Table Aggregation
Star Cubing requires the construction of the star table in advance.

Fig H): Star Table Effectiveness
The benefits of the star table are profound: it collapses the attributes dynamically and makes the star-tree shrink quickly. T
There are additional costs that come with this construction, but we will show that it is not a major expense in the context of computing the iceberg cube. Furthermore, without the star-table, the algorithm as a whole will suffer. Fig h) (Figure 19) shows the comparison of computation times between Star-Cubing with and without star tables. When the min sup is 10, both perform similarly, however, when the min sup gets larger, star table contributes to reduce the size of star-tree, thus reduces the computation time. The proportion of time used in constructing the star-table over the total run time is less than 30%.

5. Scalability
Using dimension of 10, cardinality of 10, skew of 0, minimum support of 100, we generated several data sets with up to 1000K tuples.
Cube computation factorizes the lattice space. It is well recognized that data cubing often produces huge outputs. Two popular efforts devoted to this problem are (1) iceberg cube, [7] where only significant cells are kept, and (2) closed cube, where a group of cells which preserve roll-up/drill-down semantics are loss-lessly compressed to one cell. Due to its usability and importance, efficient computation of closed cubes still warrants a thorough study.

Star-Cubing, that integrates the strength of both top-down and bottom-up cube computation, and explores a few additional optimization techniques. Two optimization techniques are worth noting: (1) shared aggregation by taking advantage of shared dimensions among the current cuboid and its descendant cuboids; and (2) prune as soon as possible the unpromising cells during the cube computation using the anti-monotonic property of the iceberg cube measure.

There are three closed iceberg cubing algorithms: C-Cubing (MM), C-Cubing (Star), and C-Cubing (StarArray), with the variations of cardinality, skew, min sup, and data dependence. The Star family algorithms perform better when min sup is low. C-Cubing (MM) is good when min sup is high. The switching point of min sup increases with the dependence in the data. High dependence incurs more c-pruning, thus it benefits the Star algorithms. Comparing C-Cubing (Star) and C-Cubing (StarArray), the former is better if the cardinality is low; otherwise, C-Cubing (StarArray) is better.

**FUTURE WORK**

As for future work, we discuss the related work and possible Extensions if approach. For efficient computation of star cubes, we have proposed an aggregation-based c-checking approach, C-Cubing. With this approach, we proposed and implemented three algorithms: C-Cubing (MM), C-Cubing (Star) [6] and C-Cubing (Star Array). All the three algorithms outperform the previous approach. Among them, we have found C-Cubing(MM) is good when iceberg pruning dominates the computation, whereas the Star family algorithms perform better when c-pruning is significant.

- Incorporating constraints with various cube computation
- Dealing with holistic functions
- Applying different compression technique to compress cube
- Supporting incremental and batch updates

**REFERENCES**


[2] D. Xin, J. Han, X. Li, B. W. Wah. Star-Cubing: Computing Iceberg Cubes by Top-Down and Bottom-Up Integration. VLDB’03.

[4] Z. Shao et al. MM-Cubing: Computing Iceberg Cubes by Factorizing the Lattice Space. SSDBM'04