Star-Cubing: Computing Iceberg Cubes by Top-Down and Bottom-Up Integration*

Dong Xin Jiawei Han Xiaolei Li Benjamin W. Wah

University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA

Abstract

Data cube computation is one of the most essential but expensive operations in data warehousing. Previous studies have developed two major approaches, *top-down* vs. *bottom-up*. The former, represented by the *Multi-Way Array Cube* (called MultiWay) algorithm [25], aggregates simultaneously on multiple dimensions; however, it cannot take advantage of Apriori pruning [2] when computing *iceberg cubes* (cubes that contain only aggregate cells whose measure value satisfies a threshold, called *iceberg condition*). The latter, represented by two algorithms: BUC [6] and H-Cubing[11], computes the iceberg cube bottom-up and facilitates Apriori pruning. BUC explores fast sorting and partitioning techniques; whereas H-Cubing explores a data structure, H-Tree, for shared computation. However, none of them fully explores multi-dimensional simultaneous aggregation.

In this paper, we present a new method, Star-Cubing, 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.

*Work supported in part by U.S. National Science Foundation NSF IIS-02-09199, the University of Illinois, and Microsoft Research. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 29th VLDB Conference, Berlin, Germany, 2003

1 Introduction

Since the introduction of data warehousing, data cube, and OLAP [8], efficient computation of data cubes has been one of the focusing points in research with numerous studies reported. The previous studies can be classified into the following categories: (1) efficient computation of full or iceberg cubes with simple or complex measures [1, 25, 18, 6, 11], (2) selective materialization of views [13, 3, 9, 10, 21], (3) computation of compressed data cubes by approximation, such as quasi-cubes, wavelet cubes, etc. [4, 23, 20, 5], (4) computation of condensed, dwarf, or quotient cubes [15, 24, 22, 16], and (5) computation of stream "cubes" for multi-dimensional regression analysis [7].

Among these categories, we believe that the first one, *efficient computation of full or iceberg cubes*, plays a key role because it is a fundamental problem, and any new method developed here may strongly influence new developments in the other categories.

The problem of cube computation can be defined as follows. In an n-dimension data cube, a cell $a = (a_1, a_2, \ldots, a_n, c)$ (where c is a measure) is called an *m-dimensional cell* (i.e., a cell in an m-dimensional cuboid), if and only if there are exactly m ($m \leq n$) values among $\{a_1, a_2, \ldots, a_n\}$ 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. We first study the case that the measure c is the *count* of base cells, and *min_sup* (threshold) is the iceberg condition. Then we extend it to complex measures in Section 5.

Previous studies have developed two major approaches, *top-down* vs. *bottom-up*, for efficient cube computation. The former, represented by the *Multi-Way Array Cube* (called MultiWay) algorithm [25], aggregates simultaneously on multiple dimensions; however, it cannot take advantage of Apriori pruning when computing *iceberg cubes*. The latter, represented by two algorithms: BUC [6] and H-Cubing[11], computes the iceberg cube bottom-up and facilitates Apriori pruning. BUC explores fast sorting and partitioning techniques; whereas H-Cubing explores a data structure, H-Tree, for shared computation. However, none of them fully explores multi-dimensional simultaneous

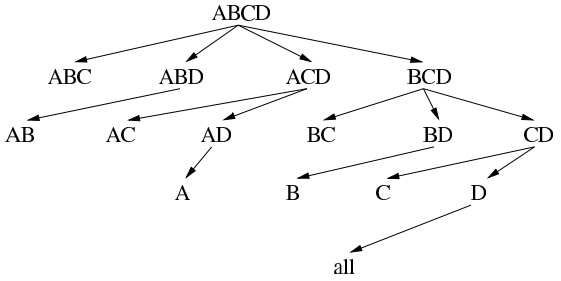

Figure 1: **Top-Down Computation**

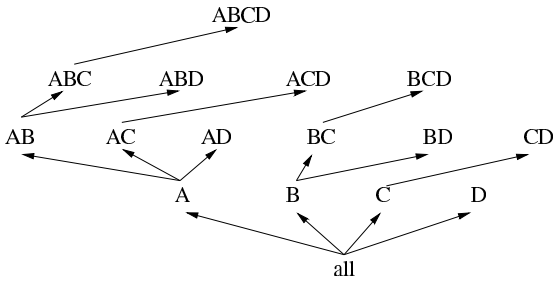

Figure 2: **Bottom-Up Computation**

aggregation.

Can we integrate the strength of the previous algorithms and develop a more efficient cubing method? In this paper, a new iceberg cubing algorithm, Star-Cubing, is proposed, 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. Our performance study shows that Star-Cubing outperforms the previous cubing algorithms in almost all the data distributions.

The remaining of the paper is organized as follows. In Section 2 the three major algorithms in cube computation are re-examined. In Section 3, we motivate the integration of the top-down and bottom-up computation, introduce the star-tree structure, and develop the Star-Cubing algorithm. Our performance study is presented in Section 4. A discussion on potential extensions is in Section 5, and we conclude our study in Section 6.

2 Overview of Cubing Algorithms

To propose our new algorithm, we first analyze each of the three popular cubing algorithms.

2.1 MultiWay

MultiWay [25] 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.

Taking `ABCD` as the base cuboid, Figure 1 shows that the results of computing cuboid `ACD` can be used to compute `AD`, which in turn can be used to compute `A`. This shared computation makes MultiWay perform aggregations simultaneously on multiple dimensions, which leads to the computation order shown in Figure 1, where intermediate aggregate values can be re-used for the computation of successive descendant cuboids.

The MultiWay algorithm is effective when the product 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. This is because the successive computation shown in Figure 1 does not have the *anti-monotonic* property [2, 17]: if a cell in `ABD` does not satisfy *min_sup*, one cannot assert that its "children cell" in the cuboid `AB` does not satisfy *min_sup* either since a cell in `AB` is likely to contain more base cells than that in `ABD`.

2.2 BUC

BUC [6] employs a bottom-up computation by starting at the apex cuboid and moving upward to the base cuboid, as shown in Figure 2. Cuboids with fewer dimensions now become parents of cuboids with more dimensions. BUC starts by reading the first dimension and partitioning it based on its distinct values. Then for each partition in the first dimension, it recursively computes the remaining dimensions.

The bottom-up computation order facilitates Apriori-based pruning. For example, if the count of a cell c in a cuboid `A` is smaller than *min_sup*, then the count of any descendant cells of c (with more dimensions, e.g., `AC`, `ACD`) can never be higher than *min_sup*. Thus the descendant cells of c can be pruned. This is implemented as follows: During partitioning, each partition's size is compared with *min_sup*. The recursion stops if the count does not satisfy *min_sup*.

The partition process is facilitated by a linear sorting method, CountingSort. CountingSort is fast because it does not perform any key comparisons to find boundaries. In addition, the counts computed during the sort can be reused to compute the group-bys.

Partitioning and sorting are the major costs in BUC's cube computation. Since recursive partitioning in BUC does not reduce the input size, both partition and aggregation are costly. Moreover, BUC is sensitive to skew in the data: the performance of BUC degrades as skew increases.

BUC is a divide-and-conquer algorithm: After a particular partition has been computed, all descendant cuboids are calculated before the algorithm switches to another partition. In the calculation of the de-

Algorithm	Simultaneous Aggregation	Partition & Prune
MultiWay	Yes	No
BUC	No	Yes
H-Cubing	Weak	Yes
Star-Cubing	Yes	Yes

Table 1: **Summary of Four Algorithms**

scendants, Apriori pruning is used to reduce unnecessary computation based on the anti-monotonic property, which was not possible in the top-down computation. However, unlike MultiWay, the results of a parent cuboid does not help compute that of its children in BUC. For example, the computation of cuboid AB does not help that of ABC. The latter needs to be computed essentially from scratch.

2.3 H-Cubing

H-Cubing [11] uses a hyper-tree structure, called H-Tree, to facilitate cube computation. Each level in the tree represents a dimension in the base cuboid. A base tuple (cell) of d-dimensions forms one path of length d (i.e., d nodes) in the tree. Nodes at the same level of tree that hold the same value are linked together via side-links. In addition, a Header Table is associated with each level to record the count of every distinct value in all the dimensions not below the current one, and provides links to the first node of the corresponding values in the H-Tree.

With this data structure, there are two methods to compute the cube: bottom-up (BOT) vs. top-down (TOP) tree traversal. In both methods, the algorithm starts at a particular level of the H-Tree, i.e., a particular dimension, and examines the group-by's that include that level and levels above it in the H-Tree. The aggregation is facilitated by the Header Table constructed in the initial pass of the data and Header Tables local to the current group-by. During the aggregation, if the *count* of a particular node is below *min_sup*, it skips itself and jumps to the next node via the side-link. The difference between the two traversal methods, H-Cubing-BOT and H-Cubing-TOP, is that the former starts the process at the bottom of the H-Tree, whereas the latter starts at the top.

One advantage of H-Cubing is that since the internal nodes of the H-Tree structure collapse duplicated data, shared processing and some simultaneous aggregation can be explored. Also, it computes less dimension combinations before proceedings to more dimensions, and thus leads to Apriori pruning for iceberg cube computation. However, similar to BUC, it cannot use the intermediate results at computing low dimensions to facilitate the computation of high dimensional cuboids.

3 Star-Cubing: An Integration of Top-Down and Bottom-Up Computation

To take advantage of the existing three cubing algorithms, we exploit the potentials of both the top-down and bottom-up models and a data structure similar to H-Tree. This leads to our proposal of the Star-Cubing algorithm. Before presenting the details of the new method, we first summarize the major computational properties of the four algorithms in Table 1. The properties of Star-Cubing will become clear in this section.

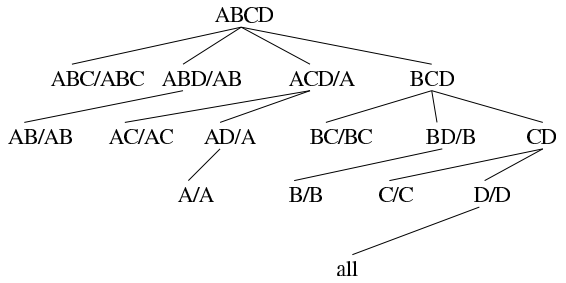

Figure 3: Star-Cubing: **Top-down computation with bottom-up growing shared dimensions**

The Star-Cubing algorithm explores both the top-down and bottom-up models: On the global computation order, it uses the top-down model similar to Figure 1. 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. In this section, we first introduce the new concepts and data structures and then present the algorithm.

3.1 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. We 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 1 is pruned in Figure 3 because AB was already computed in ABD/AB. Also, cuboid A extending from AD is pruned because it was already computed in ACD/A.

Before introducing computation on the shared dimensions, we have the following lemma.

Lemma 1. *If the measure of an iceberg cube is anti-monotonic, and if the aggregate value on a shared dimension does not satisfy the iceberg condition, all the cells extended from this shared dimension cannot satisfy the iceberg condition either.*

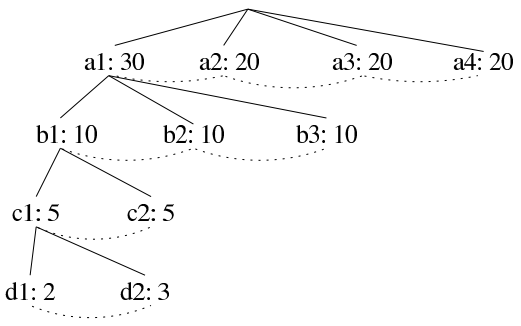

Figure 4: **A fragment of the base cuboid tree**

Rationale. Cells extended from a shared dimension contain more attributes (i.e., dimensions) than those in the shared dimension. These new attributes offer new partitions for the set of cells. If the shared dimension does not satisfy the iceberg condition, the new partitioned set contains no more tuples (base cells) than that in the shared dimension, and it will not satisfy the condition either, based on the property of anti-monotonicity [17]. Hence we have the lemma. ∎

As an example, if the value in the shared dimension `A` is `a1` and it fails to satisfy the iceberg condition, the whole sub-tree rooted at `a1CD/a1` (including `a1C/a1C`, `a1D/a1`, `a1/a1`) can be pruned.

The presence of the shared dimensions in the partial ordering of top-down computation makes it possible to take the advantages of the bottom-up evaluation. Similar to BUC, Lemma 1 will allow us to partition and prune based on shared dimensions. By integrating the top-down and bottom-up models, it gives us the best of both worlds. One critical requirement of the proposed method is that aggregate value of the shared dimensions must be calculated first. For example, before aggregating a partition in the cuboid `ABD`, the related partition must be aggregated in the shared dimension `AB` already. We will discuss how this can be achieved later.

3.2 Cuboid Trees

We use trees to represent individual cuboids. Figure 4 shows a fragment of the *cuboid tree* of the base cuboid `ABCD`. Each level in the tree represents a dimension, and each node represents an attribute. Each node has four fields: the attribute value, aggregate value, pointer(s) to possible descendant(s), and pointer to possible sibling. Tuples in the cuboid are inserted one by one into the tree. A path from the root to a node represents a tuple. For example, node `c2` in the tree has aggregate (count) value of 5, which indicates that there are five cells of value `{a1 b1 c2 *}`.

This representation collapses the common prefixes to save memory usage and allows us to aggregate the values at internal nodes. With aggregate values at internal nodes, one can perform pruning based on shared dimensions. For example, the cuboid tree of `AB` can be used to prune possible cells in `ABD`.

A	B	C	D	Count
$a1$	$b1$	$c1$	$d1$	1
$a1$	$b1$	$c3$	$d3$	1
$a1$	$b2$	$c2$	$d2$	1
$a2$	$b3$	$c3$	$d4$	1
$a2$	$b4$	$c3$	$d4$	1

Table 2: **Base (Cuboid) Table**: Before star reduction.

Dimension	Count $= 1$	Count ≥ 2
A	-	$a1(3), a2(2)$
B	$b2, b3, b4$	$b1(2)$
C	$c1, c2$	$c3(3)$
D	$d1, d2, d3$	$d4(2)$

Table 3: **One-Dimensional Aggregates**

3.3 Star Nodes and Star Trees

If the single dimensional aggregate on an attribute value p does not satisfy the iceberg condition, it is useless to distinguish such nodes in the iceberg cube computation. Thus the node p can be replaced by $*$ so that the cuboid tree can be further compressed. This motivates us to introduce the concepts of *star node* and *star tree*.

The node p in an attribute A is a **star node** if the single dimensional aggregate on p does not satisfy the iceberg condition; otherwise, p is a *non-star node*. A cuboid tree that consists of only non-star nodes and star(-replaced) nodes is called a **star-tree**.

Let's see an example of star-tree construction.

Example 1. A base cuboid table is shown in Table 2. There are 5 tuples and 4 dimensions. The cardinalities (number of distinct values) for dimensions `A`, `B`, `C`, `D` are 2, 4, 3, 4, respectively.

The one-dimensional aggregates for all attributes are shown in Table 3. Suppose $min_sup = 2$ (i.e., iceberg condition). Clearly, only attribute values `a1`, `a2`, `b1`, `c3`, `d4` satisfy the condition. All the other values are below the threshold and thus become star nodes. By collapsing star nodes, the reduced base table is Table 4. Notice the table contains two fewer rows and also fewer distinct values than Table 2.

Since the reduced base table is smaller than the original one, it is natural to construct the cuboid tree based on the reduced one. The resultant tree is the star-tree, which will be smaller due to the elimination of trivial data. ∎

To help identify which nodes are star-nodes, a **star-table** is constructed for each star-tree. Figure 5 shows the structure of the star-tree and its corresponding

A	B	C	D	Count
$a1$	$b1$	$*$	$*$	2
$a1$	$*$	$*$	$*$	1
$a2$	$*$	$c3$	$d4$	2

Table 4: **Compressed Base Table**: After star reduction.

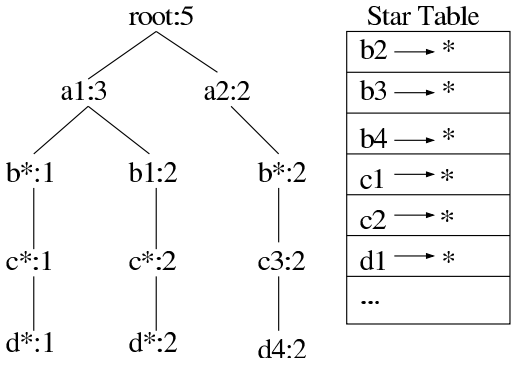

Figure 5: **Star Tree and Star Table**

star-table constructed from Table 4 (where only the start nodes are shown in the star-table). In actual implementation, one could use a bit-vector or a hash table to represent the star-table for fast lookup.

To ensure that the table reduction performed in Example 1 is correct, we need to show that the star-tree compression is lossless.

Lemma 2. *Given a specific iceberg condition, the compression performed for derivation of* star-tree *by collapsing star nodes is loseless.*

Rationale. If the single-dimensional aggregate for node p in a particular dimension cannot satisfy the iceberg condition, augmenting p with more dimensions cannot derive any new aggregate that can satisfy the iceberg condition, based on the Apriori property. Therefore, it is safe to replace that attribute with the star node, denoted by *. By doing so, the cardinality of the dimension will be smaller and the corresponding cuboid tree will be more compressed, but it will not lose any aggregate cell that satisfy the iceberg condition. ■

3.4 Multi-Way Star-Tree Aggregation

With the generated star-tree, one can start the process of aggregation by traversing in a top-down fashion. Traversal will be depth-first. The first stage (i.e., the processing of the first branch of the tree) is shown in Figure 6. The leftmost tree in the figure is the base star-tree. The subscripts by the nodes in the tree show the order of traversal. The remaining four trees are `BCD`, `ACD/A`, `ABD/AB`, `ABC/ABC` from left to right. They are the child trees of the base star-tree, and correspond to the second level in Figure 3. The subscripts in them correspond to the same subscripts in the base tree, and they denote the steps when they are created during the tree traversal. For example, when the algorithm is at step 1, the `BCD` child tree root is created. At step 2, the `ACD/A` child tree root is created. At step 3, the `ABD/AB` tree root and the `b*` node in `BCD` are created.

When the algorithm has reached step 5, the trees in memory are exactly as shown in Figure 6. Since the depth-first traversal has reached a leaf, it will start backtracking. Before traversing back, the algorithm notices that all possible nodes in the base dimension (`ABC`) have been visited. This means the `ABC/ABC` tree is complete so the count is output and the tree is destroyed. Similarly, upon moving back from `d*` to `c*` and seeing that `c*` has no siblings, the count in `ABD/AB` is also output and the tree is destroyed.

When the algorithm is at `b*` during the back-traversal, it notices that there exists a sibling in `b1`. Therefore, it will keep `ACD/A` in memory and perform depth-first search on `b1` just as it did on `b*`. This traversal and the resultant trees are shown in Figure 7. The two rightmost child trees are created again but now with the new values from the `b1` subtree. The trees that remained intact during the last traversal are reused and the new aggregate values are added on. For instance, another branch is added to the `BCD` tree.

Just like before, the algorithm will reach a leaf node at `d*` and traverse back. This time, it will reach `a1` and notice that there exists a sibling in `a2`. In this case, all child trees except `BCD` in Figure 7 are destroyed. Afterwards, the algorithm will perform the same traversal on `a2`. This is shown in Figure 8. Notice that `BCD` keeps growing while the others have started fresh.

There are several issues that we did not discuss or encounter here due to the simplicity of our base table. Also, we did not recursively build child trees. There could also be the case that a non-star node in the base tree could become a star-node in a child tree. We shall discuss these and more issues as follows.

3.4.1 Node Ordering

The star-tree data structure is a compressed version of the original data. It provides a good improvement in memory usage; however, in general, searching in a tree still takes time. For example, to search for a specific node in a particular level in Figure 5 would require $O(n)$ time where n is the number of nodes in the level or the cardinality of that dimension. To search for an entire tuple would require $O(dn)$ time where d is the number of dimensions.

In order to reduce such cost, the nodes in the star-tree are sorted in alphabetic order in each dimension. For example, in Figure 6, the nodes in the first level are sorted in the order of `a1`, `a2`. In general, all the levels will have the order of *, `p1`, `p2`, ..., `pn`. The position of the star-node can be anywhere but it is the first in our implementation since we believe that it may often occur more frequently than any other single node. This ordering allows a node to be easily located during tree traversal, and it only needs to be done once in the construction of the base cuboid. All the local star-trees will be generated from the base cuboid and thus will inherit the ordering.

3.4.2 Child Tree Pruning

A *child tree* is one level lower than the current tree, e.g., `a1CD/a1` is a child of the base-tree as shown in Figure 6. To improve cubing performance, one should prune useless child trees. There are two conditions that the current node must satisfy in order to generate

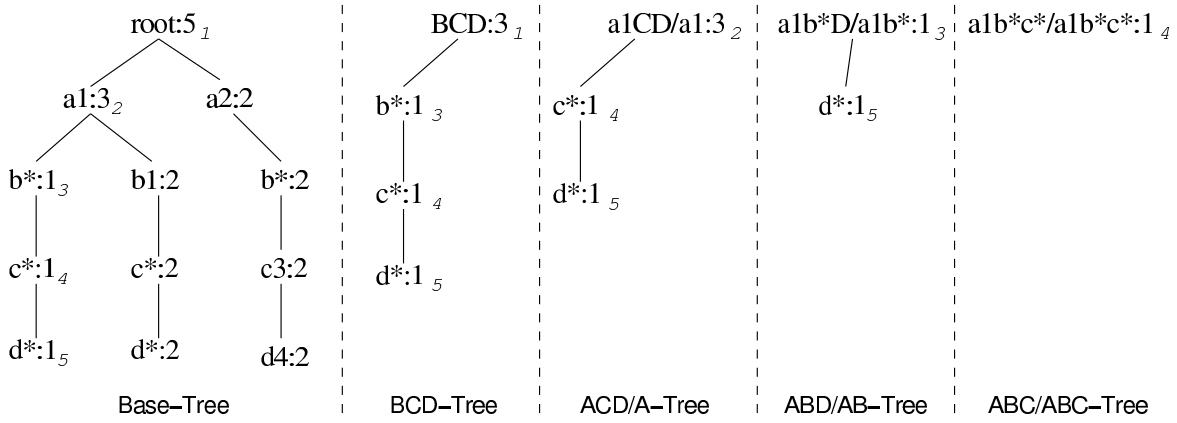

Figure 6: **Aggregation Stage One:** Processing of the left-most branch of Base-Tree

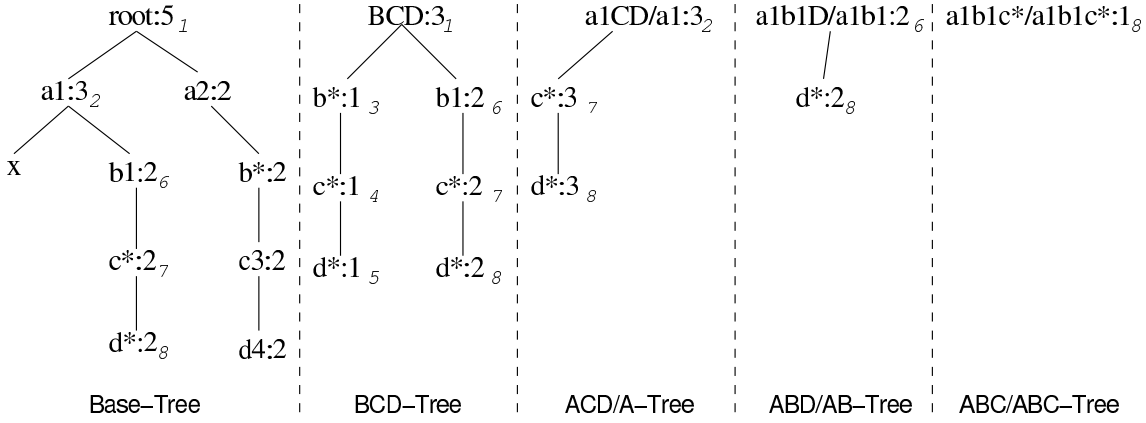

Figure 7: **Aggregation Stage Two:** Processing of the second branch of Base-Tree

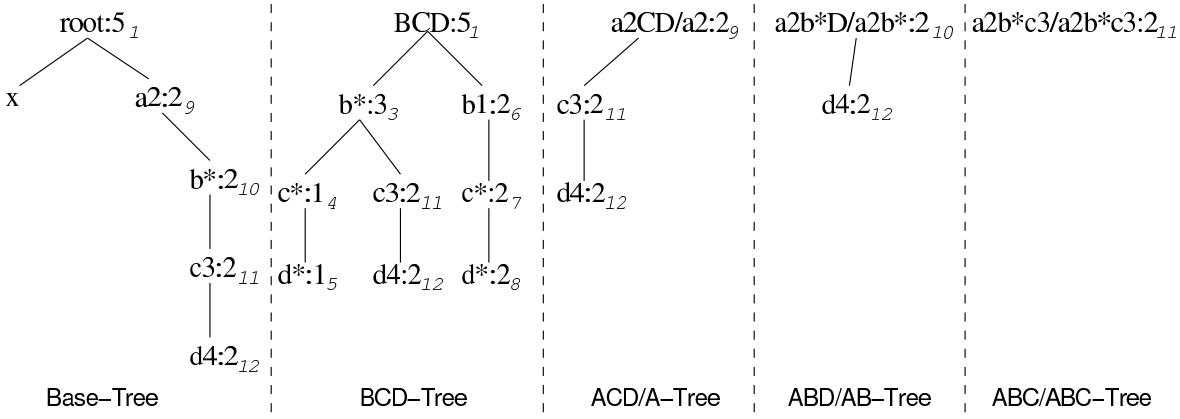

Figure 8: **Aggregation Stage Three:** Processing of the last branch of Base-Tree

child trees: (1) the measure of the current node must satisfy the iceberg condition; and (2) the tree to be generated must include at least one non-star (i.e., non-trivial) node. This is because if all the nodes were star nodes, it means that none of them satisfies *min_sup* and is of no use to the iceberg cube. Therefore, it would be a complete waste to compute them.

The tree pruning based on the second condition can also be observed in Figures 6–7. For example, the left sub-tree extending from node `a1` in the base-tree in Figure 6 does not include any non-star nodes. According to the second condition, the `a1CD/a1` sub-tree should not have been generated. They were still generated in the figure just for the illustration of the child tree generation process.

3.4.3 Star Nodes

It is possible that a non-star node in the base tree could become a star-node in a child tree. For example, a node could just barely satisfy the iceberg condition in the base tree, but when it is divided in the child trees, it no longer satisfies the condition. For this reason, it is required that all the nodes be checked again in the construction of child trees. That is, all the trees shown in Figures 6–8 have their own star table which will count the nodes and make them star-nodes where appropriate.

3.5 Memory Management

Due to the numerous construction and destruction of cuboid trees in the algorithm, memory management becomes an important issue. Instead of using the standard memory allocation command (e.g., `new` and `delete` in C++), we will instead maintain a *free node list*. During the construction of a tree, whenever a new node is needed, the algorithm will just request a node from the free node list. When deallocating a node, the algorithm will just add the node back to the list of free nodes.

To initialize the free node list, the algorithm will allocate kdn nodes into a *node buffer*, where d is the number of dimensions, n is the number of tuples, and k is a scaling factor dependent on the iceberg condition. The larger the minimum support is, the smaller k is. In practice, a value of 0.2 is usually sufficient for k. To begin, the free node list is empty. New nodes from the node buffer are added to the list whenever nodes are needed. When the free node list and node buffer are both empty, more nodes are acquired from the system memory.

This memory management strategy proves to be an effective optimization to the algorithm for two reasons: (1) with the free node list, memory allocation commands are replaced by pointer operations, which are much faster; and (2) by avoiding constantly allocating and de-allocating small memory chunks (nodes are small) in the system memory heap, fragmentation is avoided.

In practice, the total memory requirement is usually less than kdn. This is because the star-tree compresses the data.

Similar to the free node list, the algorithm maintains a free cuboid tree list as well.

3.6 Dimension Ordering

Similar to other iceberg cube algorithms, Star-Cubing is sensitive to the order of the dimensions. The goal of ordering dimensions is to prune the trees as early as possible. The internal nodes whose aggregate values do not satisfy the iceberg condition by the biggest value should be processed earlier.

For the best performance, the dimensions are ordered on cardinality in descending order. Please note that this ordering is in reverse of the dimension ordering of the H-Tree where dimensions are ordered on cardinality in ascending order (the dimension with the least cardinality is on the top, the closest to the root). The cardinality-descending ordering of the dimensions on the star-tree may lead to the generation of bigger initial (base) tree, but it leads to a better chance of early pruning. This is because the higher the cardinality, the smaller the partitions, and therefore the higher possibility that the partition will be pruned.

3.7 Star Table Construction

The star-tree is a major component of the algorithm. It collapses the attributes and makes the tree size shrink quickly. To build the star-tree, the star-table is needed. Although the cost to build the star-table is non-trivial, without it the total computation of the star-tree will be much more expensive.

There are, however, two situations where the star-table does not need to be computed. First, no star-table is needed in the computation of a *full cube* because there is no star node. Second, when a node is at a high level of the ancestor tree, i.e., the corresponding partition is fairly big, the chance for star node to appear at this level is slim. It is not beneficial to compute the star-table. One can use the aggregate value on the node to estimate whether a star-table should be computed.

3.8 Star-Cubing Algorithm

Based on previous discussions, the Star-Cubing algorithm is summarized as follows.

Algorithm 1 (Star-Cubing). Compute iceberg cubes by Star-Cubing.

Input: (1) A relational table R, and (2) an iceberg condition, *min_sup* (taking *count* as the measure)

Output: The computed iceberg cube.

Method: Each star-tree corresponds to one cube-tree node, and vice versa. The algorithm is described in Figure 9.

```
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
{
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 ≠ root)
5.          output cnode.count;
6.      if (cnode is a leaf)
7.          output cnode.count;
8.      else { // initiate a new cube-tree
9.          create C_C as a child of T's cube-tree;
10.         let T_C as C_C's star-tree;
11.         T_C.root's count = cnode.count;
12.     }
13. }
14. if (cnode is not a leaf)
15.     call starcubing(T, cnode.first_child);
16. if (C_C is not null) {
17.     call starcubing(T_C, T_C.root);
18.     remove C_C from T's cube-tree; }
19. if (cnode has sibling)
20.     call starcubing(T, cnode.sibling);
21. remove T;
}
```

Figure 9: The Star-Cubing algorithm

Analysis. With the step-by-step discussions in this section, the program is self-explanatory. Based on Lemmas 1 and 2, the algorithm derives the complete and correct iceberg cube with the input table R, and the iceberg condition, *min_sup*.

The efficiency of the algorithm is based on three major points: (1) 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. (2) It explores the multi-way tree aggregation. By scanning base tree once, it aggregates value on multiple children trees. (3) 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. ∎

4 Performance Analysis

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 we can achieve for the other three algorithms: MultiWay, BUC, and H-Cubing, based on the published literature. All the four algorithms were coded using C++ on an AMD Athlon 1.4GHz system with 512MB of RAM. The system ran Linux with a 2.4.18 kernel and gcc 2.95.3. The times recorded include both the computation time and the I/O time. Similar to other performance studies in cube computation [25, 6, 11], all the tests used the data set that could fit in main memory.

For the remaining of this section, $\mathcal{D}$ denotes the number of dimensions, $\mathcal{C}$ the cardinality of each dimension, $\mathcal{T}$ the number of tuples in the base cuboid, $\mathcal{M}$ the minimum support level, and $\mathcal{S}$ the skew or `zipf` of the data. When $\mathcal{S}$ equals 0.0, the data is uniform; as $\mathcal{S}$ increases, the data is more skewed. $\mathcal{S}$ is applied to all the dimensions in a particular data set.

4.1 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 four algorithms are compared with respect to tuple size (Figure 10), cardinality (Figure 11) and dimension (Figure 12). In the first experiment, we 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, we increased dimension number from 3 to 7 while keeping the cardinality of each dimension 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. We did 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. This phenomenon is also observed in [6] and [18]. 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.

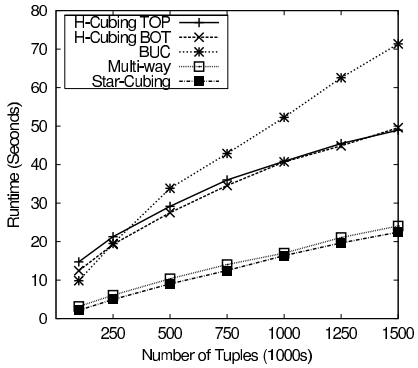

Figure 10: Full Cube Computation w.r.t. Tuple Size, where $\mathcal{D} = 5$, $\mathcal{C} = 5$, $\mathcal{S} = 0$, $\mathcal{M} = 1$

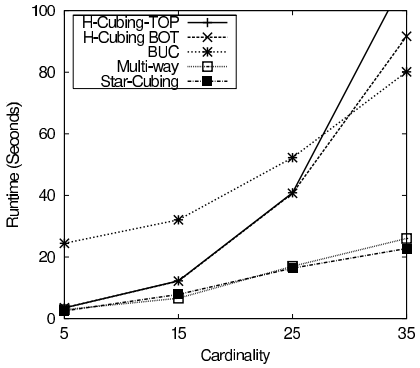

Figure 11: Full Cube Computation w.r.t. Cardinality, where $\mathcal{T} = 1M$, $\mathcal{D} = 5$, $\mathcal{S} = 0$, $\mathcal{M} = 1$

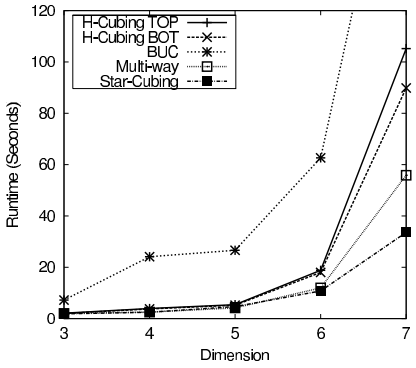

Figure 12: Full Cube Computation w.r.t. Dimension, where $\mathcal{T} = 1M$, $\mathcal{C} = 10$, $\mathcal{S} = 0$, $\mathcal{M} = 1$

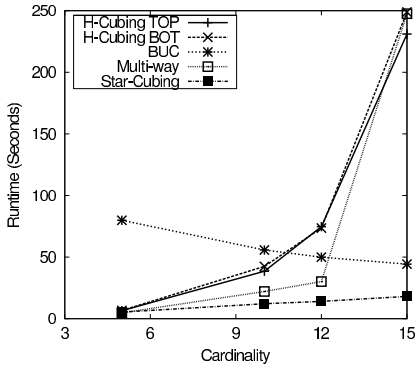

Figure 13: Iceberg Cube Computation w.r.t. Cardinality, where $\mathcal{T} = 1M$, $\mathcal{D} = 7$, $\mathcal{S} = 0$, $\mathcal{M} = 1000$

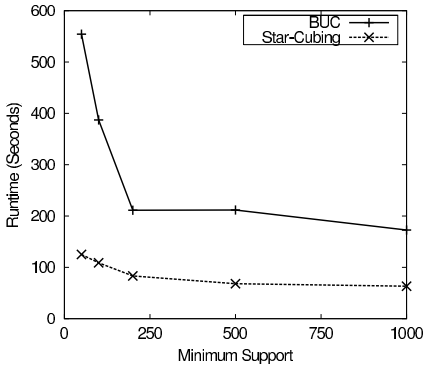

Figure 14: Star-Cubing vs. BUC w.r.t. Minsup, where $\mathcal{T} = 1M$, $\mathcal{D} = 10$, $\mathcal{C} = 10$, $\mathcal{S} = 0$

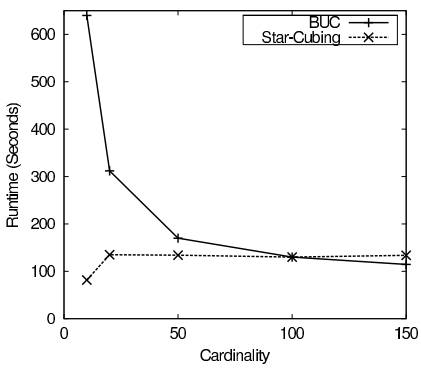

Figure 15: Star-Cubing vs. BUC w.r.t. Cardinality, where $\mathcal{T} = 1M$, $\mathcal{D} = 10$, $\mathcal{S} = 1$, $\mathcal{M} = 100$

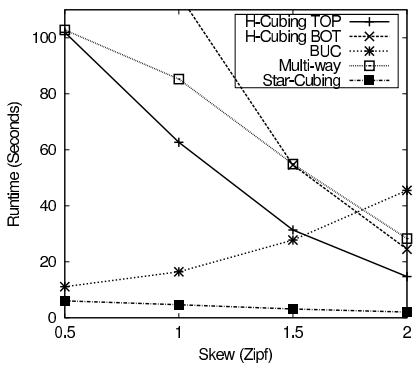

Figure 16: Data Skew, where $\mathcal{T} = 150K$, $\mathcal{D} = 10$, $\mathcal{C} = 8$, $\mathcal{M} = 1000$

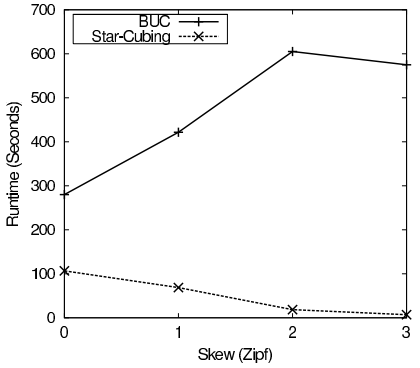

Figure 17: Star-Cubing vs. BUC w.r.t. Skew, where $\mathcal{T} = 1M$, $\mathcal{D} = 10$, $\mathcal{C} = 10$, $\mathcal{M} = 100$

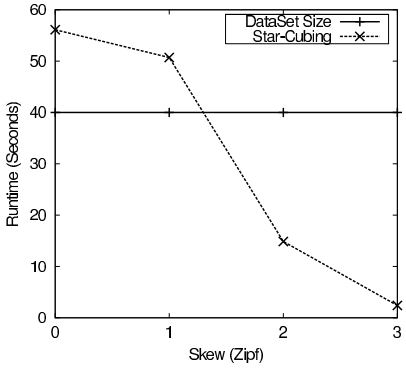

Figure 18: Memory Usage w.r.t. Skew, where $\mathcal{T} = 1M$, $\mathcal{D} = 10$, $\mathcal{C} = 10$, $\mathcal{M} = 100$

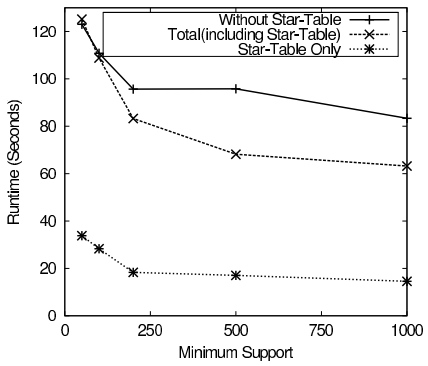

Figure 19: Star-Table Effectiveness, $\mathcal{T} = 1M$, $\mathcal{D} = 10$, $\mathcal{C} = 10$, $\mathcal{S} = 0$

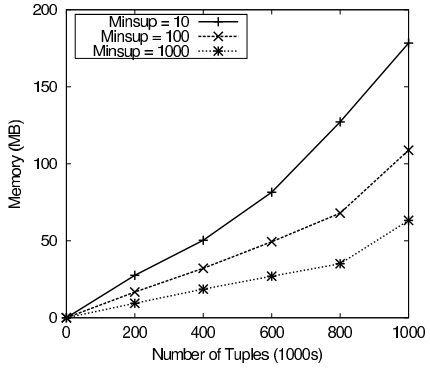

Figure 20: Scalability w.r.t. # Tuples, where $\mathcal{D} = 10$, $\mathcal{C} = 10$, $\mathcal{S} = 0$

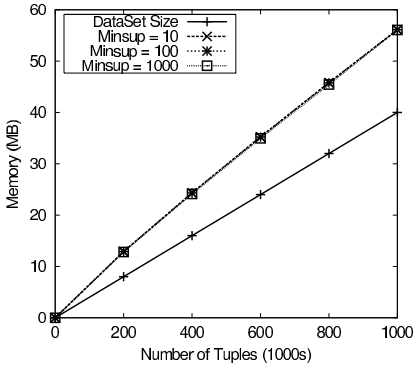

Figure 21: Memory Usage w.r.t. # of Tuples, $\mathcal{D} = 10$, $\mathcal{C} = 10$, $\mathcal{S} = 0$

4.2 Iceberg Cube Computation

The second set of 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. As seen in the previous experiments, both MultiWay and H-Cubing do not perform well in high dimension and high cardinality datasets. We compared BUC and Star-Cubing under high dimension and high cardinality individually. The results are shown in Figures 13–15.

The data set used in 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. 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 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 decreases. 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.

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.

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 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 Figure 15), 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.

4.3 Data Skew

In this section, we will show that skewness affects the performance of the algorithms. We 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.

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 workload to MultiWay. The two H-Cubing algorithms in Figure 16 start performing much better once $\mathcal{S}$ 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 $\mathcal{S}$ 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 Figure 17. The result is similar to 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) [11], 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.

4.4 Additional Star-Table Aggregation

Star-Cubing requires the construction of the star table in advance. The benefits of the star table are profound: it collapses the attributes dynamically and makes the star-tree shrink quickly. 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.

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%.

4.5 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. Figure 20 shows the scalability of Star-Cubing with respect to different *min_sup* level. The figure shows Star-Cubing is scalable with respect to database size.

Figure 21 shows the total memory requirement with respect to size of data sets. As seen from the figure, the total memory requirement is slightly larger than the original data size, and the total memory usage is almost the same for different *min_sup* levels. This is because the memory is mainly used by the base tree. The sub-trees are relatively small.

In summary, we have tested four cubing algorithms: MultiWay, BUC, H-Cubing, and Star-Cubing, with the variations of density, *min_sup*, cardinality and skewness. For dense data, Star-Cubing is always the best, MultiWay and H-Cubing are also good when dimensionality is low. For sparse data, both Star-Cubing and BUC are good candidates. Usually, Star-Cubing performs better than BUC. If the cardinality is low, Star-Cubing runs several times faster than BUC. However, if the cardinality goes really high, BUC performs better. For skewed data, Star-Cubing improves its performance when the data skew increases, while BUC's performance deteriorates. H-Cubing-Top also performs very well for low-dimensional skewed data. Although there is no all-around clear-cut winner; however, in most cases, Star-Cubing performs better or substantially better than others.

5 Discussion

In this section, we will discuss a few issues related to Star-Cubing and point out some research directions.

5.1 Handling Large Databases

All the data sets used in our performance tests can fit in main memory. One may wonder what may happen if the dataset cannot fit in memory. Actually, Star-Cubing does not require that the base star-tree fit in memory. This is because for any branch of the base star-tree, Star-Cubing will need to scan it only once, as demonstrated in Figures 6–8. Thus one can load the star-tree page by page. When a used star-tree page is swapped out, the space it occupies can be released since one will not need to visit it again in the cubing process. Thus the largest star-tree, which is the initial base tree, will not need to be in memory. Only the lower level, smaller trees will need to be kept in memory during cube computation. Please note that the memory management method proposed in Section 3.5 has taken this factor into consideration, with the additional optimization by designing our own efficient but simple memory management routines.

One may also consider the case that even the much smaller, non-base trees may not fit in memory, although we believe that such a chance is rare if the dimension ordering rule is observed. In this case one can adopt projection-based preprocessing similar to that in FPtree-based frequent pattern mining [12] and do cubing for each projected database.

5.2 Computing Complex Measures

Throughout the paper, we have used `count()` as the iceberg measure. Complex measures such as `average()` can be easily incorporated into our algorithm, based on the technique proposed in [11].

For example, for computing iceberg cube with the condition, "$min_sup(c) = k$ and $average(c) > v$", for each cell c, one may store top-k quant-info at each node of the tree and use the same technique as that proposed in [11, 15] to perform anti-monotonicity testing to filter out those unpromising nodes during the cube computation process. Computing other complex measures may adopt the similar techniques suggested in [11].

5.3 Materializing only Cube Shells in a High Dimensional Cube

Due to the nature of the exponential growth of the number of cells in a data cube with the growth of the number of dimensions, it is unrealistic to compute a full cube or even an iceberg one for high dimensional data cubes. Instead, one may compute only the cube "shells" in a high dimensional cube by materializing only those cuboids that consist of a small number of dimension combinations. That is, one may materialize only up to m-dimensional cuboids in an n-D cube, where m is a small integer, such as 5, but n could be nontrivial, such as 50.

Bottom-up processing, such as BUC and H-Cubing, can handle such "shell" computation naturally because it computes cuboids from low dimensional combinations to higher ones. However, the pure top-down cubing, such as MultiWay, will encounter its difficulty since it computes from high-dimension combination toward lower ones. Without computing from 50-dimensional cuboid to 49-, 48-, ..., one cannot reach small dimensional cuboid computation.

Star-Cubing solves this difficulty nicely by exploring the notion of share dimension. Since the maximum number of generated dimensions cannot be over 5 (suppose $m = 5$), instead of trying to project and generate unneeded dimension combinations, only the

shared dimensions with dimension no more than 5 will be generated and examined. Thus the computation is in the same spirit as bottom-up processing, and the derived algorithm should still be efficient, with no additional overhead.

6 Conclusions

For efficient cube computation in various data distributions, we have proposed an interesting cube computation method, 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. No previous cubing method has fully explored both optimization methods in one algorithm. Moreover, a new compressed data structure, star-tree, is proposed using star nodes. And a few other optimization techniques also contribute to the high performance of the method.

Our 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.

There are many interesting research issues to further extend the Star-Cubing methodology. For example, efficient computation of condensed or quotient cubes, computing approximate cubes, computing cube-gradients [14], and discovery-driven exploration of data cubes [19] using the Star-Cubing methodology are interesting issues for future research.

References

[1] S. Agarwal, R. Agreal, P. M. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. VLDB'96.

[2] R. Agrawal and R. Srikant. Fast algorithm for mining association rules. VLDB'94.

[3] E. Baralis, S. Paraboschi, and E. Teniente. Materialized view selection in a multidimensional database. VLDB'97, 98-12.

[4] D. Barbara and M. Sullivan. Quasi-cubes: Exploiting approximation in multidimensional database. SIGMOD Record, 26:12-17, 1997.

[5] D. Barbara and X. Wu. Using loglinear models to compress datacube. WAIM'00, 311-322.

[6] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. SIGMOD'99, 359–370.

[7] Y. Chen, G. Dong, J. Han, B. W. Wah, and J. Wang, Multi-Dimensional Regression Analysis of Time-Series Data Streams. VLDB'02.

[8] 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, 1:29–54, 1997.

[9] H. Gupta. Selection of views to materialize in a data warehouse. ICDT'97, 98-112.

[10] H. Gupta, V. Harinarayan, A. Rajaranman, and J. D. Ullman. Index selection of OLAP. ICDE'97, 208-219.

[11] J. Han, J. Pei, G. Dong, and K. Wang. Efficient computation of iceberg cubes with complex measures. SIGMOD'01, 1-12.

[12] J. Han, J. Pei, and Y. Yin. Mining Frequent Patterns without Candidate Generation. SIGMOD'00, 1-12.

[13] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. SIGMOD'96.

[14] T. Imielinski, L. Khachiyan, and A. Abdulghani. Cubegrades: Generalizing Association Rules. Data Mining and Knowledge Discovery, 6(3):219-258, 2002.

[15] L. V. S. Lakeshmanan, J. Pei and J. Han. Quotient Cubes: How to Summarize the Semantics of a Data Cube. VLDB'02.

[16] L. V. S. Lakeshmanan, J. Pei and Y. Zhao. QC-Trees: An Efficient Summary Structure for Semantic OLAP, SIGMOD'03.

[17] R. Ng, L. V. S. Lakeshmanan, J. Han and A. Pang. Exploratory Mining and Pruning Optimizations of Constrained Associations Rules, SIGMOD'98.

[18] K. Ross and D. Srivastava. Fast Computation of sparse datacubes. VLDB'97.

[19] S. Sarawagi, R. Agrawal, and N. Megiddo. Discovery-Driven Exploration of OLAP Data Cubes. EDBT'98.

[20] J. Shanmugasundaram, U. M. Fayyad, and P. S. Bradley. Compressed Data Cubes for OLAP Aggregate Query Approximation on Continuous Dimension. KDD'99.

[21] A. Shukla, P. M. Deshpande, and J. F. Naughton. Materialized view selection for multidimensional datasets. VLDB'99.

[22] Y. Sismanis, A. Deligiannakis, N. Roussopoulos and Y. Kotids. Dwarf: Shrinking the PetaCube. SIGMOD'02.

[23] J. S. Vitter, M. Wang, and B. R. Iyer. Data Cube approximation and histograms via wavelets. CIKM'98.

[24] W. Wang, J. Feng, H. Lu and J. X. Yu. Condensed Cube: An Effective Approach to Reducing Data Cube Size. ICDE'02.

[25] Y. Zhao, P. Deshpande, J. F. Naughton: An Array-Based Algorithm for Simultaneous Multidimensional Aggregates. SIGMOD'97.