Computing Iceberg Cubes by Top-Down and Bottom-Up Integration: The StarCubing Approach

Dong Xin, *Student Member*, *IEEE*, Jiawei Han, *Senior Member*, *IEEE*, Xiaolei Li, Zheng Shao, and Benjamin W. Wah, *Fellow*, *IEEE*

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* versus *bottom-up*. The former, represented by the *MultiWay Array Cube* (called the `MultiWay`) algorithm [30], aggregates simultaneously on multiple dimensions; however, it cannot take advantage of a priori pruning [2] when computing *iceberg cubes* (cubes that contain only aggregate cells whose measure values satisfy a threshold, called the *iceberg condition*). The latter, represented by `BUC` [6] , computes the iceberg cube bottom-up and facilitates a priori pruning. `BUC` explores fast sorting and partitioning techniques; however, it does not fully explore multidimensional simultaneous aggregation. In this paper, we present a new method, `Star-Cubing`, that integrates the strengths of the previous two 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-bys that do not satisfy the iceberg condition. Our performance study shows that `Star-Cubing` is highly efficient and outperforms the previous methods.

Index Terms—Data warehouse, data mining, online analytical processing (OLAP).

1 Introduction

Efficient computation of data cubes has been one of the focusing points in research since the introduction of data warehousing, OLAP, and data cube [9]. Previous studies can be classified into the following categories:

1. efficient computation of full or iceberg cubes with simple or complex measures [1], [6], [13], [21], [30],
2. computation of compressed data cubes by approximation, such as quasi-cubes, wavelet cubes, etc., [4], [5], [23], [27],
3. computation of condensed, dwarf, or quotient (closed) cubes [17], [18], [26], [28], [29],
4. selective materialization of views [3], [10], [11], [15], [19], [24], and
5. computation of stream "cubes" for multidimensional 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 there may strongly influence new developments in the other categories. The quotient iceberg cube can be efficiently computed by adding a closedness measure on an iceberg cubing algorithm [29]. The efficient cubing algorithm can also improve the performance of the GROUPING SET operator (partial cube) [8] in commercial database systems. Due to the similarity of iceberg cubing and frequent pattern mining [14], we believe the principles developed here also benefit frequent pattern mining algorithms.

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* if $m = n$; otherwise, it is an *aggregate cell*. Given a base table R, our task is to compute an *iceberg cube* [6], i.e., the set of cells which satisfies an iceberg condition, or the *full cube* if there is no such condition. An iceberg condition is the constraint to prune the uninteresting cells. We first study the case that measure c is the *count* of cells, and $c \geq min_sup$ (min_sup is a user given threshold) is the iceberg condition (i.e., only cells whose *counts* are no less than min_sup are of interest). We extend this to complex measures in Section 6.

Previous studies have developed two principles for efficient cube computation: 1) *sharing computations as much as possible* and 2) *pushing the iceberg constraint as deep as possible*. The former aims at reducing the amortized cost for computing each cell, and the latter tries to prune unnecessary computations. In general, the algorithms speed up the computation by sharing partitions, sorts, or partial sorts for group-bys with common dimensions. Methods in this line can be traced from *PipeSort*, *PipeHash*, and *Overlap*

- *D. Xin, J. Han, and X. Li are with the Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801. E-mail: {dongxin, hanj, xli10}@uiuc.edu.*
- *Z. Shao is with Yahoo! Corporation, 2811 Mission College Blvd., Santa Clara, CA 95054. E-mail: zshao@yahoo-inc.com.*
- *B.W. Wah is with the Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, Urbana, IL 61801. E-mail: wah@uiuc.edu.*

Manuscript received 16 Apr. 2006; revised 17 July 2006; accepted 24 Aug. 2006; published online 20 Nov. 2006.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-0174-0406.

1041-4347/07/$20.00 © 2007 IEEE Published by the IEEE Computer Society

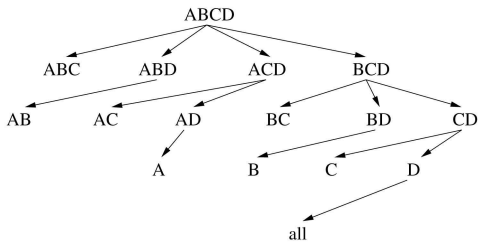


Fig. 1. Top-down computation.

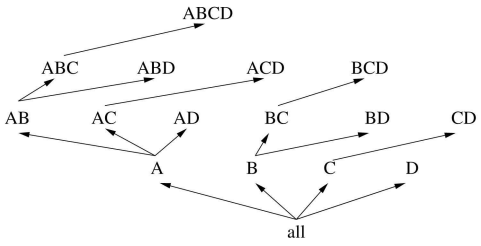


Fig. 2. Bottom-up computation.

[1], *MultiWay Array Cube* [30], *PartitionCube*, and *MemoryCube* [21] to recent work on *optimization with Grouping Set query* [8]. All these works share a common computation rule that low dimension group-bys are aggregated from high dimension group-bys (we refer this as a *top-down* approach). On the other hand, the *iceberg constraint* [6], [13] proposes a new optimization opportunity for cube computation. Like many frequent pattern mining algorithms [14], the iceberg constraint can take advantage of a priori pruning which requires the computation order to be from low dimension to high dimension (we refer to this as *bottom-up*). Unfortunately, the two approaches follow conflicting orders during the computation and cannot exploit both principles simultaneously.

Can we integrate the strength of the previous algorithms and develop a more efficient cubing method? In this paper, we propose a new iceberg cubing algorithm, `Star-Cubing`, which integrates the top-down and bottom-up cube computation and explores both multidimensional aggregation and the a priori pruning. A new data structure, star-tree, is introduced that explores lossless data compression and prunes unpromising cells using an a priori-like dynamic subset selection strategy. Our performance study shows that `Star-Cubing` outperforms the previous cubing algorithms in almost all data distributions.

The remainder of the paper is organized as follows: Section 2 reviews the related work and reexamines two representative algorithms in cube computation. In Section 3, we present the principle of integration of the top-down and bottom-up computation. Section 4 introduces the star-tree structure and develops the `Star-Cubing` algorithm. Our performance study is presented in Section 5. Section 6 discusses the potential extensions, and Section 7 is the conclusion.

2 Related Work

In iceberg cube computation, the top-down and bottom-up approaches were considered as mutually exclusive. In this paper, we present a new integrated method which exploits both computation principles. Before proposing our new algorithm, we first review the typical cubing algorithms in both *top-down* and *bottom-up* categories.

2.1 Top-Down Approach

There are many algorithms in these categories; we select `MultiWay` [30] as an example. `MultiWay` [30] is an array-based top-down cubing algorithm. It uses a compressed sparse array structure to load the base table 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, Fig. 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 allows `MultiWay` to perform aggregations simultaneously on multiple dimensions. This leads to the computation order shown in Fig. 1, where intermediate aggregate values can be reused for the computation of successive descendant cuboids. In Fig. 1, every node in the tree is a *cuboid*, and the root node is the *base cuboid*. We call the tree shown in Fig. 1 a *top-down cuboids spanning tree*.

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 a priori 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 Fig. 1 does not have the *antimonotonic* property [2], [20]: 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` contains more base cells than that in `ABD`.

2.2 Bottom-Up Approach

We select `BUC` [6] as a representative algorithm for this category. `BUC` employs a bottom-up computation by starting at the apex cuboid (all) and moving upward to the base cuboid, as shown in Fig. 2. Cuboids with fewer dimensions now become parents of cuboids with more dimensions. We call the tree shown in Fig. 2 a *bottom-up cuboids spanning tree*. `BUC` starts by reading the full fact table and partitioning it based on the distinct values in the first dimension. Then, for each partition, it recursively computes the remaining dimensions.

The bottom-up computation order facilitates a priori-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` and `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

TABLE 1
Summary of Three Algorithms

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

compared with min_sup, and the recursion stops if the size does not satisfy min_sup.

The partition process is facilitated by a linear sorting method, `CountingSort`. When the cardinality is not too large, `CountingSort` is fast since it does not perform any key comparisons to find boundaries. Otherwise, `BUC` uses quick sort. 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 partitioning and aggregation are costly. Moreover, `BUC` is sensitive to skew in the data: Performance 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 descendants, a priori pruning is used to reduce unnecessary computation based on the antimonotonic 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.

Besides the iceberg constraint, there is much work exploiting the closure semantic in multidimensional cube space [17], [18], [25], [26], [28], [29]. The closure semantic proposes a different constraint on data cube output and can also be pushed deep to enable early pruning. This problem is orthogonal to our work in this paper. In fact, closure pruning can be efficiently integrated into many existing cubing algorithms by computing an additional closedness measure, as demonstrated by [29].

3 Integration of Top-Down and Bottom-Up Computation: The Principle

In this section, we introduce the principle of integrating top-down and bottom-up computation. The implementation of this principle, `Star-Cubing`, will be discussed in the next section. Before presenting the details of the new method, we first summarize the major computational properties of the three algorithms in Table 1. The properties of `Star-Cubing` will become clear in this section.

3.1 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 Fig. 1. However, it has a sublayer underneath based on the bottom-up model by exploring the notion of *shared dimensions*.

An observation of Fig. 1 may disclose an interesting fact: All the cuboids in the left-most subtree of the root include dimensions `ABC`, all those in the second subtree include dimensions `AB`, and all those in the third include dimension `A`. We call these common dimensions the *shared dimensions* of those particular subtrees. Recall, we termed Fig. 1 as the *top-down cuboids spanning tree* and Fig. 2 as the *bottom-up cuboids spanning tree*. The definition of the shared dimensions is as follows:

Definition 1 (Shared Dimensions). *A set of shared dimensions is a set of the prefix dimensions of the parent node and the set of maximum prefix dimensions of all nodes in the subtree in a top-down cuboids spanning tree.*

Based on this concept, Fig. 3a is extended to Fig. 3b, 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.

Notice that a set of shared dimensions is also a cuboid. In fact, node `ACD/A` represents two cuboids: one is `ACD` and the other is `A`. The `ACD` cuboid is the *upper bound* of all the cuboids in the subtree such that all those cuboids will be computed directly or indirectly from it. The `A` cuboid is the *lower bound* of all the cuboids in the subtree since all those cuboids include it. We promote cuboid `A` up to a higher level to facilitate the iceberg pruning. In fact, the integration of top-down and bottom-up computation is achieved by the integrated node `ACD/A`, where `ACD` is used for simultaneous aggregation and `A` is used for partition and prune.

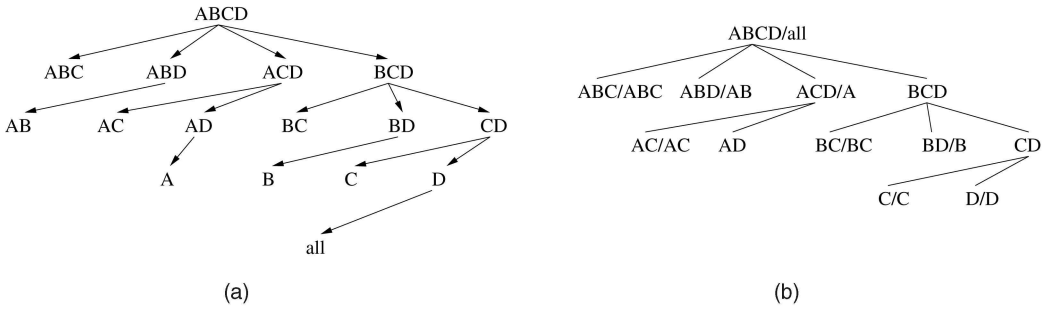


Fig. 3. `Star-Cubing`: Top-down computation with bottom-up growing shared dimensions. (a) Top-down and (b) integration of top-down and bottom-up.

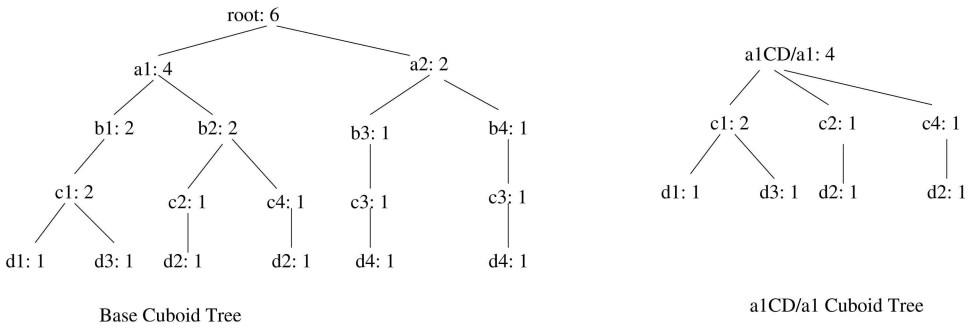


Fig. 4. Fragments of the base cuboid tree and a1CD/a1 tree.

We refer to a cuboid with shared dimensions as *integrated cuboid* from here on.

Since the cuboid corresponding to the shared dimensions is identified early in the tree expansion, there is no need to compute them later. For example, node AB extending from ABD in Fig. 3a is pruned in Fig. 3b because AB was already computed in ABD/AB. Also, cuboid A extending from AD is pruned because it was already computed in ACD/A.

3.2 Cuboid Tree

The shared dimensions build relationships between cuboids by linking cells in between them. For example, if cell a_1 does not satisfy iceberg condition (thus, does not exist in cuboid A), a_1 will not appear in any cuboids in the subtree under node ACD/A. To be used for pruning, cells in the shared dimensions should be computed before the whole subtree. This gives us the motivation that we can use a tree structure to represent a cuboid. For example, given the dimension order as A, B, C, D, we can get the supports of ABCD cells on the leaves. At the same time, the supports of A, AB, and ABC cells are available on the internal nodes. From this point of view, the shared dimensions should also be a prefix of parent node in cuboids spanning tree.

Fig. 4 shows fragments of the *cuboid trees* of the base cuboid ABCD and an integrated cuboid ACD/A (i.e., the cuboid ACD with the shared dimension A). Each level in the tree represents a dimension, and each node represents an attribute. For simplicity, we refer to the node with an attribute value p as node p thereafter. Each node has four fields: the attribute value, the aggregate value, the pointer to the left most child, and the pointer to possible sibling. A path from the root to a node represents a cell. For example, node c1 in the tree has aggregate (count) value of 2, which indicates that the cell a1b1c1* has value 2. This representation collapses the common prefixes to save memory usage and allows us to aggregate the values at internal nodes.

While there is only one tree instance for the base cuboid, there are multiple tree instances for other cuboids. The instances are partitioned by the attributes in shared dimensions. For example, the ACD/A cuboid tree has multiple instances, e.g., a1CD/a1 and a2CD/a2. This can be viewed as having multiple partitions of cuboids. Similarly, ABD/AB has multiple instances, e.g., a1b1D/a1b1 and a1b2D/a1b2. Furthermore, multiple instances of a cuboid tree are not kept in memory at the same time. They are created and computed one by one according to the partitions. We will discuss the detail of computation order later.

3.3 Iceberg Pruning

The presence of the shared dimensions in the partial ordering of top-down computation makes it possible to take advantage of the bottom-up evaluation. Similar to BUC, Lemma 1 will allow us to partition and prune based on shared dimensions.

Lemma 1. *If the measure of an iceberg cube is antimonotonic, and if the aggregate value on the shared dimensions does not satisfy the iceberg condition, all the cells extending from this set of the shared dimensions cannot satisfy the iceberg condition either.*

Rationale. Cells belonging to the cuboids under a set of shared dimensions contain more attributes (i.e., dimensions) than those in the shared dimensions. These new attributes create new partitions for the set of cells, where each partition is guaranteed to be no larger than the base set. Thus, the aggregate value in the shared dimensions does not satisfy the iceberg condition, the newly partitioned set cannot either, based on the property of antimonotonicity [20].

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

We point out that the selection of the shared dimensions is not unique. As an example, Fig. 5 shows a different spanning tree with less pruning power. We are interested in what is the best cuboids spanning tree and how to find the best pruning strategy.

Definition 2 (Regular Spanning Tree). *A top-down cuboids spanning tree is a regular spanning tree if:*

1. *The root of the spanning tree is the base cuboid containing all dimensions.*

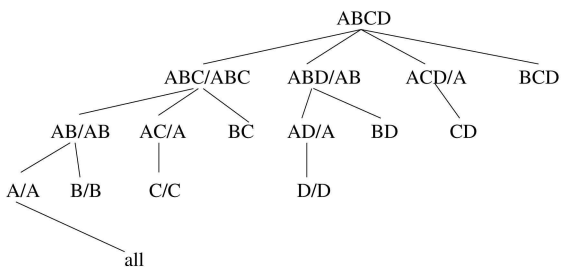


Fig. 5. An alternative cuboids spanning tree.

TABLE 2
Base (Cuboid) Table: Before Star Reduction

A	B	C	D	Count
$a1$	$b1$	$c1$	$d1$	1
$a1$	$b1$	$c1$	$d3$	1
$a1$	$b2$	$c4$	$d2$	1
$a1$	$b2$	$c2$	$d2$	1
$a2$	$b3$	$c3$	$d4$	1
$a2$	$b4$	$c3$	$d4$	1

2. *The children of a node are cuboids with one less dimension.*
3. *The order of dimensions is fixed among all cuboids.*

Among all possible regular spanning trees, we use Algorithm 1 to construct our target spanning tree.

Algorithm 1 The algorithm to construct a regular spanning tree
Input: (1) A set of cuboids S, and (2) Dimension order O
Output: A regular spanning tree

`Sort` dimensions on all cuboids in S according to O;
`Find` the cuboid R with the largest number of dimensions;
`call` $Partition(R, S - \{R\})$;

`procedure` $Partition(R, S)$// R, the root cuboid, S the set of cuboids to be partitioned
1: `for each` nonempty prefix p of R (ordered from long to short)
2: `set` $NewS$ as empty
3: `for each` cuboid c in S
4: `if` p is c's prefix
5: `remove` c from S and insert c into $NewS$
6: `find` the cuboid U with the largest number of dimensions in $NewS$
7: `find` the cuboid L with the least number of dimensions in $NewS$
8: `set` U/L (an integrated node) as a child of R
9: `call` $Partition(U, NewS - \{U, L\})$

The correctness of the algorithm is straightforward. One can show that the above algorithm generates a spanning tree as shown in Fig. 3b.

3.4 Simultaneous Aggregation

Given the cuboid spanning tree, we now discuss how to achieve simultaneous aggregation. We use a depth-first search to traverse a parent cuboid tree, and at the same time, multiple children cuboid trees are updated simultaneously. The parent tree will be traversed only once. At each node of the parent tree, the algorithm makes the following two decisions: 1) Whether it should create a new child tree? For example, when the algorithm visits node `a1` of the base cuboid tree in Fig. 4, the support of `a1` in shared dimension `A` is available. According to the cuboids spanning tree, the node `ACD/A` is under consideration. If `a1` satisfies the iceberg condition, then the cuboid tree `a1CD/a1` will be created. Note that `a1` will be in the root, and the dimensions involved in the aggregation of this tree are only `C` and `D`. This cuboid tree will aggregate all cells of the partial cuboid `a1CD` by rolling up on the dimension `B`. 2) How to aggregate the information on the node to the multiple children trees? For example, when the algorithm visits node `c1`, the information on `c1` will be aggregated to child cuboid tree `a1CD/a1`. It will not be pushed to child cuboid tree `a1b1D/a1b1` since `C` does not belong to that cuboid.

When the traversal algorithm finishes a subbranch in the parent cuboid tree, the underlying child cuboid tree has been constructed completely. For example, when the traversal algorithm returns to node $a1$, the child tree `a1CD/a1` has been computed fully. We then consider `a1CD/a1` as a new parent cuboid tree and use a recursive procedure to compute `a1C/a1C` and `a1D/a1`. A detailed description of the multiway tree aggregation is presented in next section.

TABLE 3
One-Dimensional Aggregates

Dimension	Count ¡ 2	Count $\geq$ 2
A	-	$a1, a2$
B	$b3, b4$	$b1, b2$
C	$c2, c4$	$c1, c3$
D	$d1, d3$	$d2, d4$

4 Star-Cubing: An Efficient Implementation Using Star Tree

In this section, we discuss the implementation details of the `Star-Cubing` algorithm.

4.1 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 *star* ("$*$") so that the cuboid tree can be further compressed. This motivates us to introduce the concepts of *star node* and *star tree*.

A 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 *nonstar node*. A cuboid tree that consists of only nonstar nodes and star (-replaced) nodes is called a *star-tree*. Let us look at an example of star-tree construction.

Example 1. A base cuboid table is shown in Table 2. There are six tuples and four dimensions. The cardinalities (number of distinct values) for dimensions `A`, `B`, `C`, and `D` are 2, 4, 4, and 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`, `b2`, `c1`, `c3`, `d2`, and `d4` satisfy the condition. All the other

TABLE 4
Compressed Base Table: After Star Reduction

A	B	C	D	Count
$a1$	$b1$	$c1$	$*$	2
$a1$	$b2$	$*$	$d2$	2
$a2$	$*$	$c3$	$d4$	2

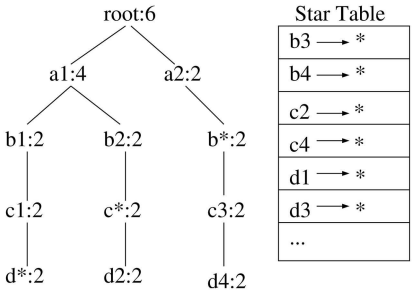


Fig. 6. Star tree and star table.

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 three 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 it. The resultant tree is the star-tree, which will also be smaller.

To help identify which nodes are star-nodes, a *star-table* is constructed for each star-tree. Fig. 6 shows the structure of the star-tree and its corresponding star-table constructed from Table 4 (where only the star nodes are shown in the star-table). In 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 (with regard to the iceberg cube computation).

Lemma 2. *Given a specific iceberg condition, the compression performed for derivation of star-tree by collapsing star nodes is lossless (with regard to the iceberg cube computation).*

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 a priori 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 cells which satisfy the iceberg condition.

Note *star* is different from *all*, all attributes whose single dimension aggregates are less than iceberg condition are collapsed to *star*, while *all* includes all attributes no matter whether their aggregates exceeds the iceberg conditions or not. In the case where the star node does not have any sibling, star is equivalent to all. Since star attribute values will not appear in output, we further restrict that star attribute values not be included in shared dimensions.

4.2 Multiway 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 Fig. 7. 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`, and `ABC/ABC` from left to right. They are the *child trees* of the base star-tree, and correspond to the second level in Fig. 3. Correspondingly, we call the base tree the *parent tree*. 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 `b1` node in `BCD` are created.

When the algorithm has reached Step 5, the trees in memory are exactly as shown in Fig. 7. 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 shared dimension (`a1b1c1`) have been visited. This means the `ABC/ABC` tree is complete so the count is output and the tree is destroyed. (In our implementation, child tree `ABC/ABC` is not created since there will be no

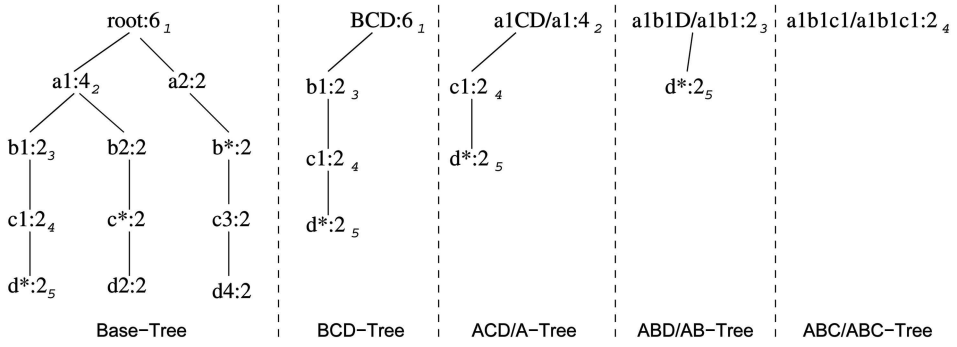


Fig. 7. **Aggregation Stage One**: Processing of the left-most branch of base-tree.

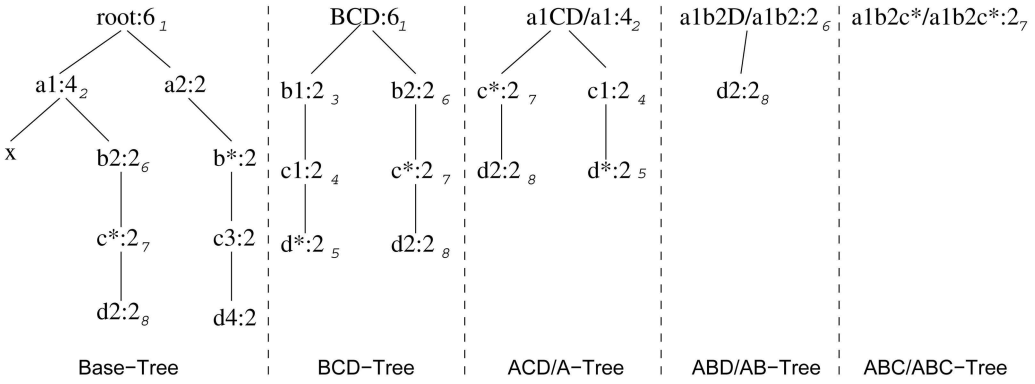


Fig. 8. **Aggregation Stage Two**: Processing of the second branch of Base-Tree.

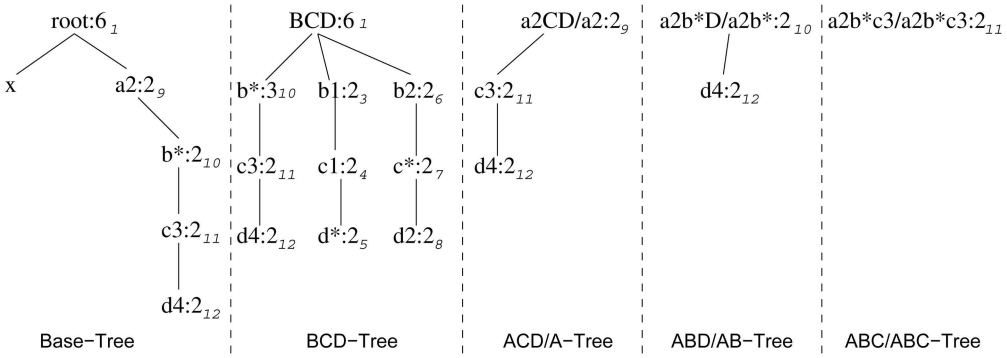


Fig. 9. **Aggregation Stage Three**: Processing of the last branch of Base-Tree.

further aggregation on it. The count will be output directly from Base-Tree. It was still generated in the figure just for the illustration of the child tree generation process). Similarly, upon moving back from `d*` to `c1` and seeing that `c1` has no siblings, the count in `ABD/AB` is also output and the tree is destroyed.

When the algorithm is at `b1` during the back-traversal, it notices that there exists a sibling in `b2`. Therefore, it will keep `ACD/A` in memory and perform depth-first search on `b2` just as it did on `b1`. This traversal and the resultant trees are shown in Fig. 8. The two rightmost child trees are created again but now with the new values from the `b2` 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 `d2` 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 Fig. 8 are destroyed. Afterward, the algorithm will perform the same traversal on `a2`. This is shown in Fig. 9. 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. It could also be the case that a nonstar node in the base tree could become a star-node in a child tree. We shall discuss these and more issues as follows.

4.2.1 Node Ordering

The star-tree data structure is a compressed version of the original data. However, searching in a tree still takes time. For example, to search for a specific node in a particular level in Fig. 6 requires $O(n)$ 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 this cost, the nodes in the star-tree are sorted in alphabetical order in each dimension. For example, in Fig. 7, the nodes in the first level are sorted in the order of `a1`, `a2`. In general, all the levels will have the order of $*, \texttt{p1}, \texttt{p2}, \ldots, \texttt{pn}$. The position of the star-node can be anywhere but it is the first in our implementation since we believe that it may occur more frequently than any other single node. This ordering allows a node to be easily located during tree traversal. All the local star-trees will be generated from the base cuboid and, thus, will inherit the ordering.

4.2.2 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 Fig. 7. To improve cubing performance, one should prune useless child trees. There are three conditions that the current node must satisfy in order to generate child trees: 1) The measure of the current node must satisfy the iceberg condition. 2) The tree to be generated must include at least one nonstar (i.e., nontrivial) 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. 3) There is no star node in the shared dimensions. This is because all values in shared dimensions are part of the output, and the star node can never be part of the output.

The first condition is trivial. The tree pruning based on the last two conditions can also be observed in Fig. 7, Fig. 8, and Fig. 9. For example, the child tree extending from `a1b1` in the base-tree in Fig. 7 does not include any nonstar nodes. According to the second condition, the `a1b1D/a1b1` child tree should not have been generated. In Fig. 8, the star node `c*` appears in the shared dimensions of `a1b2c*/a1b2c*` child tree, and `b*` appears in the shared dimensions of `a2b*D/a2b*` child tree. They were still generated in the figures just for the illustration of the child tree generation process.

4.2.3 Star Nodes

It is possible that a nonstar node in the base tree could become a star-node in a child tree. For example, an attribute value of a node could just barely satisfy the iceberg condition in the base tree, but when it is partitioned 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 Fig. 7, Fig. 8, and Fig. 9 have their own star table which will count the nodes and make them star-nodes where appropriate.

4.3 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 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 with, the free node list is empty. New nodes from the node buffer are added to the list whenever nodes are no longer needed. When the free node list and node buffer are both used up, 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 deallocating small memory chunks 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.

4.4 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. A common strategy is to order the dimensions in cardinality-descending order. 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 is, the smaller the partitions are and, therefore, the higher the possibility that the partition will be pruned.

However, sometimes the cardinality-descending ordering may be too coarse to catch the different distribution of each dimension. For example, given two dimensions in a data set with 100k tuples, one has a cardinality of 10,000 with very big skew (almost all tuples lie on small number values), and the other has a cardinality of 9,000 with uniform distribution. For a reasonably given min_sup (i.e., 100), it is desirable to order the second dimension before the first dimension. Motivated by this, we propose another ordering strategy based on data distribution. Suppose we have a data set whose dimensions have the same cardinality but have different skews. In this case, obviously the ordering should be determined by the skews. We use entropy, instead of cardinality, to order the dimension. The entropy for a dimension A is defined as:

$$Entropy(A) = -\sum_{i=1}^{Card(A)} \left(\frac{|a_i|}{|tuples|}\right) \times \log\left(\frac{|a_i|}{|tuples|}\right),$$

where a_i is the number of tuples whose value on dimension A is a_i, and $Card(A)$ is the cardinality of dimension A. Ignoring the constant items, we will compare a measure E:

$$E(A) = -\sum_{i=1}^{Card(A)} |a_i| \times \log(|a_i|)$$

for all the dimensions.

The more uniform the value distribution on the dimension is, the larger the entropy value is; thus, we order the dimension in measure E descending order.

4.5 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. The stat table is constructed by traversing the associated subtree and counting the frequencies of each attribute. Although the cost to build the star-table is nontrivial, 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 (in our implementation, we do not compute that star table as the aggregate value is larger than $min_sup \times c$, where c is the maximum cardinality among dimensions under this node).

4.6 Star-Cubing Algorithm

Based on previous discussions, the `Star-Cubing` algorithm is summarized in Algorithm 2. 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.

Algorithm 2 The `Star-Cubing` algorithm
Input: (1) A relational table R
(2) An iceberg condition, min_sup (taking *count* as the measure)
Output: The computed iceberg cube

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

`procedure` *starcubing*(T, *cnode*)// cnode: current node
1: `for each` nonnull *child* C of T in cuboids spanning tree {
2: `insert or aggregate` *cnode* to the corresponding
3: position or node in C's star-tree;
4: `if` (*cnode*.*count* $\geq min_sup$) {
5: `if` (cnode $\neq$ root)
6: `output` cnode.count;
7: `else` { // initiate a new child in cuboids spanning tree
8: `create` C_C as a child of T in cuboids spanning tree;
9: `let` T_C be C_C's star-tree;
10: T_C.*root*'s *count* = *cnode*.*count*; }}
11: `if` (*cnode* is not a leaf)
12: `call` *starcubing*(T, *cnode*.*first_child*);
13: `if` (C_C is not null) {
14: `call` *starcubing*(T_C, T_C.*root*);
15: `remove` C_C from T in cuboids spanning tree; }
16: `if` (*cnode* has sibling)
17: `call`*starcubing*(T, *cnode*.*sibling*);
18: `remove` T;

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 multiway tree aggregation. By scanning the 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 a tree shrink quickly. Thus, both computation time and memory requirement are reduced.

In terms of space requirement, we have the following lemma to justify that there are always limited number of cuboid trees in memory.

Lemma 3. *At any time, there are at most $n+1$ cuboid trees in memory, where n is the number of dimensions.*

Rationale. A cuboid tree can be released from memory if and only if all the child cuboid trees in the cuboids spanning tree have been computed and released. A cuboid tree will generate multiple child cuboid trees recursively. We use the notation $T(t, f)$ to represent a cuboid tree, where t is the total number of dimensions for the cuboids and f is the total number of free dimensions (i.e., total number of dimensions minus number of shared dimensions). The number of child cuboid trees of $T(t, f)$ is at most f. In our four-dimension cube example, the base cuboid is $T(4, 4)$, and it generates four child cuboid trees $T(3,0), T(3,1), T(3,2), T(3,3)$, representing `ABC/ABC`, `ABD/AB`, `ACD/A`, and `BCD`.

Suppose that at a particular time, the algorithm just finished the computation of a cuboid tree $T(t, f)$. There are f new children trees generated and the total number of cuboid trees in memory is $S(t, f)$. We show that $S(t, f) \leq n+1$. We prove the claim by induction. First, $S(n, n) = n+1$ because there are only one base cuboid tree and n children trees directly generated by the root. Assume $S(t+1, f') \leq n+1$ is true for all valid f'. According to our definition of $T(t, f)$, f' is valid only if $f' \leq t+1$. Consider $S(t, f)$, where $T(t, f)$ is generated by $T(t+1, f^*)$. We know that, when $T(t, f)$ is ready for generating new children trees, all the $T(t, p)$ $(p < f)$ must have been fully released. There are a total of $f-1$ trees which have been destroyed and $T(t, f)$ will generate at most $f-1$ new trees. We have

$$S(t, f) \leq S(t+1, f^*) - (f-1) + (f-1) = S(t+1, f^*) \leq n+1.$$

We conclude that the total number of cuboid trees is upbounded by $n+1$ for n dimension cube. □

4.7 Improve Performance by Indexing

The basic `Star-Cubing` algorithm works well in dense and not-so-sparse data sets. However, in a very sparse data set, where the cardinalities are large, the star tree gets very wide. Although a sparser cube enables `Star-Cubing` to prune earlier, the increase in tree size requires more time in construction, traversal, and aggregation, which negates the effects of pruning. On the other hand, while the tree gets wider, there are lots of single paths in the tree. A single path is a branch of the tree where no forks exist. For example, in Fig. 6, `a2b*c3d4` is a single path. Single paths have nice properties that can be used for optimization.

First, a single path enables child tree pruning. The combination in values of dimensions within the single path is simpler (the value on each dimension either chooses the value appears in the path or *all*, there is no other possibility). We do not create new child trees on the single path, instead, a simple enumeration will generate all possible output.

Second, single paths can be shared by different child trees. According to the star-cubing algorithm, nodes on the parent tree will be aggregated to the child trees. Most single paths on these child trees stay identical or partially the same to each other (i.e., `b1c1d*` on Base-Tree, `b1c1d*` on BCD-Tree and `c1d*` on ACD/A-Tree in Fig. 7). To avoid the cost of traversal and copying on the single paths, it is preferable to replace the single path with an index, pointing the original single path which was kept in a reference array.

To make the idea of indexing transparently embedded into the `Star-Cubing` algorithm, several techniques are developed as outlined below.

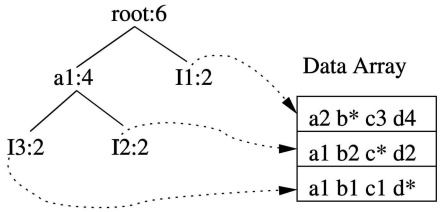


Fig. 10. Star tree with indexing.

4.7.1 Index Node

As presented in the last section, memory management is an important factor for the performance. The presupposition for repeatedly using the nodes is that they have the same data structure: (attribute, measure, child, sibling). We use the same data structure to represent an index by letting the attribute field be an index value. To differentiate from the previous notations, we call the nodes with attribute value as *attribute node*, nodes with index value as *index node*. An index node is always a leaf of the tree (see Fig. 10). If a leaf is found at the depth less than the depth it is supposed to be (i.e., the last depth of the tree), then it is an index node, and the difference of depth is the length of the single path. To avoid confusion, we never use an index node to point to a single path whose length is 1, that is, if the leaf node appears at the last depth of the tree, it must be an attribute node.

4.7.2 Aggregation with Index

In the process of multiway aggregation, when an index node is aggregated to a child cuboid tree, we use the corresponding attribute value in the single path to find the target node in the child cuboid tree. There are several possibilities: If the target node (i.e., nodes with the attribute value) does not exists, then a new node is created, and the index is assigned to the node. If the target node is an attribute node, then the measure of that node is updated, and the index node is aggregated to its children recursively. If the target node is an index node, then a comparison is done between these two single paths, a sequence of attribute nodes are generated until the attribute values on these two single paths are different. Two new index nodes are created as the children of the last attribute node.

4.7.3 Pruning Child Trees

All the single paths we mentioned above are branches whose end nodes are leaves of a cuboid tree. In order to fully exploit the single path pruning in the algorithm, we extend our definition of single path to a branch whose end node cannot necessarily be a leaf. These single paths are not necessary to be represented using index nodes. As shown in Fig. 11, `a1b1c1` is a single path. When a (extended) single path is met during the traversal, we delayed the creation of new child tree until the forking point is reached. In our example, child trees `a1CD/a1` and `a1b1D/a1b1` are pruned, only child tree `a1b1c1/a1b1c1` is created.

5 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 two algorithms: `MultiWay` and `BUC`, based on the published literature. We denote *Star1* as the `Star-Cubing` algorithm described in Section 4 and *Star2* as `Star-Cubing` with indexing. All the four algorithms were coded using C++ on an Intel Pentium-4 2.6GHz system with 1G of RAM. The times recorded include both the computation time and the I/O time. Similar to other performance studies in cube computation [30], [6], [13], all the tests used the data set that could fit in main memory.

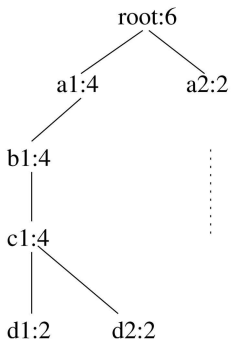


Fig. 11. Pruning child trees with single path.

We have conducted experiments on both synthetic data set and real data set. In synthetic data set, $\mathcal{D}$ 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 (`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. In real data set, we use the weather data set SEP83L.DAT[1] which has 1,002,752 tuples. We select 13 dimensions, and the attributes (cardinalities) are as follows: year month day hour (238), sky brightness indicator (2), station number (6,515), present weather (100), lower cloud amount (18), lower cloud base height (21), low cloud type (36), middle cloud type (95), high cloud type (93), middle cloud amount (79), change code (110), solar altitude (1,535), and relative lunar illuminance (155).

5.1 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 is compared with respect to tuple size (Fig. 12), cardinality (Fig. 13), and dimension (Fig. 14). In the first experiment, we randomly generated data sets with five dimensions, varying the number of tuples from 1,000K to 1,500K. In the second experiment, we varied the cardinalities for each dimension from 5 to 35. Finally, we increased dimension number from three to seven while keeping the cardinality of each dimension at 10. The tuple size for the latter two data sets was 1,000K. All the data were uniformly distributed, i.e., skew was 0.

The experimental results are shown in Fig. 12, Fig. 13, and Fig. 14. We did not use more dimensions and greater cardinality because in high dimension and high cardinality data sets, 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

1. http://cdiac.ornl.gov/ftp/ndp026b/.

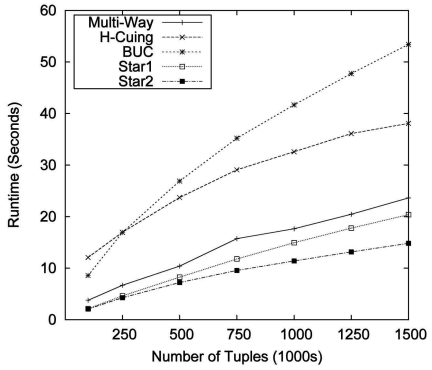


Fig. 12. Full cube computation with regard to number of tuples, where $\mathcal{D} = 5$, $\mathcal{C} = 25$, and $\mathcal{S} = 0, \mathcal{M} = 1$.

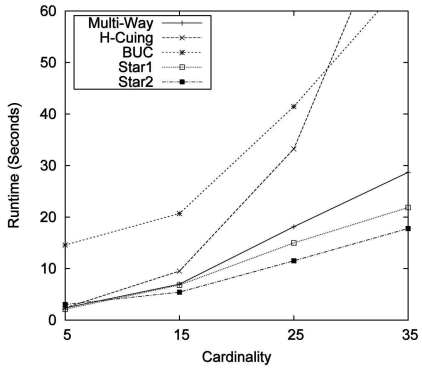


Fig. 13. Full cube computation with regard to cardinality, where $\mathcal{T} = 1M$, $\mathcal{D} = 5$, and $\mathcal{S} = 0, \mathcal{M} = 1$.

[6] and [21]. 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` degrades quickly when the dimension increases. Second, in those cases, `BUC` shows 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

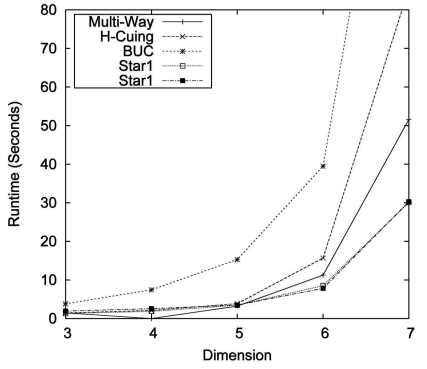


Fig. 14. Full cube computation with regard to dimension, where $\mathcal{T} = 1M$, $\mathcal{C} = 10$, and $\mathcal{S} = 0, \mathcal{M} = 1$.

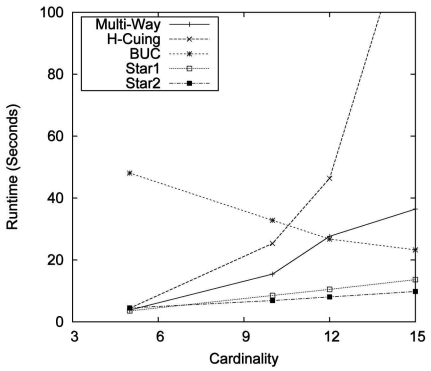


Fig. 15. Iceberg cube computation with regard to cardinality, where $\mathcal{T} = 1M$, $\mathcal{D} = 7$, $\mathcal{S} = 0$, and $\mathcal{M} = 1,000$.

increases. Finally, the two `Star-Cubing` algorithms show similar performance. Star2 takes advantage of single path when the cardinality increases; however, it also suffers from the overhead to manage attribute node and index node at the same time and visit attributes via index transforming.

5.2 Iceberg Cube Computation

The second set of experiments compares the four algorithms for iceberg cube computation. Except `MultiWay`, all the algorithms tested use some form of pruning that exploits the antimonotonicity of the `count` measure. As seen in the previous experiments, `MultiWay` does not perform well in high dimension and high cardinality data sets. We compared `BUC` and `Star-Cubing` only. The results are shown in Fig. 15, Fig. 16, and Fig. 17.

The data set used in Fig. 15 has 1,000K tuples with seven dimensions and zero skew. The min_sup is 1,000. The cardinality of each dimension is increased from 5 to 15. We can see that `BUC` and `Star-Cubing` performs better in sparse data. Star2 shows better performance than Star1 when the data gets sparser. We further compared these two algorithms with higher dimension and cardinality. In Fig. 16, the data set has 1,000K tuples with 10 dimensions, each with cardinality of 10. The skew of data is 0. At the point where min_sup is 1,000, `Star-Cubing` decreases the computation time more than 50 percent comparing with `BUC`. The improvements in performance get much higher when the min_sup level decreases. For example, when

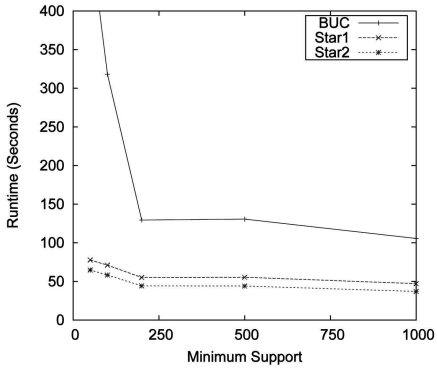


Fig. 16. Star-cubing versus BUC with regard to Minsup, where $\mathcal{T} = 1M$, $\mathcal{D} = 10$, $\mathcal{C} = 10$, and $\mathcal{S} = 0$.

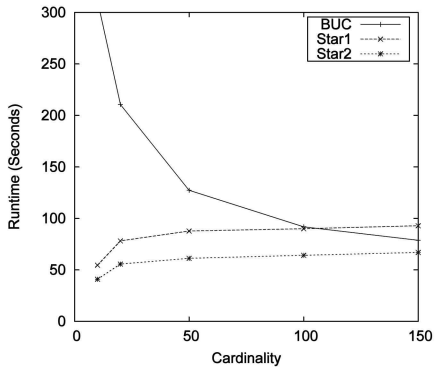


Fig. 17. Star-cubing versus BUC with regard to cardinality, where $\mathcal{T} = 1M$, $\mathcal{D} = 10$, $\mathcal{S} = 1$, and $\mathcal{M} = 100$.

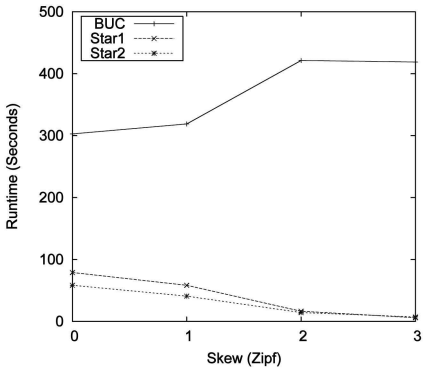


Fig. 19. Star-cubing versus BUC with regard to skew, where $\mathcal{T} = 1M$, $\mathcal{D} = 10$, $\mathcal{C} = 10$, and $\mathcal{M} = 100$.

min_sup is 50, `Star-Cubing` runs around five times faster than `BUC`.

Fig. 17 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. The *Star2* algorithm with the indexing techniques reduces the overhead and the experimental results shows that it works quite well. The simultaneous aggregation does not play an important role here because most of the outputs have a small number of nontrivial (not all) attributes (for example, in Fig. 17, when the cardinality is 150, 47.3 percent of the outputs have at most two nontrivial attributes, and 91.4 percent of the output have at most three nontrivial attributes). We believe this is a rare case in real-life applications since the data cube loses the power of multidimensional analysis.

Many real data sets have skews, correlations, and mixed dimensions with both high and low cardinalities. All these properties favor the star-cubing algorithm. For example, the skews and correlations in the data introduce more star nodes and, thus, the star tree will be dramatically compressed. Our performance studies on the real weather data (Fig. 24 and Fig. 25), which has cardinalities as high as 6,515, shows that Star-Cubing is significantly faster than BUC. We also observed that: 1) In the cases where the application have high cardinality and high minimum support, both Star-Cubing and BUC run very fast. 2) In other cases, both algorithms run slower due to higher complexity, and Star-Cubing can be significantly faster than BUC.

5.3 Data Skew

In this section, we show that skewness affects the performance of the algorithms. We use Zipf to control the skew of the data, and vary Zipf from 0 to 3 (0 being uniform). The input data had 1,000K tuples, 10 dimensions, and cardinality of 10 for each dimension. The min_sup was 100.

Fig. 18 shows the computation time for the four algorithms. Skewed data made `MultiWay` and `Star-Cubing` perform better. `BUC` is the only one that degrades. `MultiWay` improved because many chunks 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 of `MultiWay`.

We also compared `BUC` with `Star-Cubing` in a sparse data set in Fig. 19. The result is similar to Fig. 18: `BUC`'s performance degraded with increased skew while `Star-Cubing` improved.

Finally, Fig. 20 shows the memory usage o `Star-Cubing` (without indexing) comparing with the original data size.

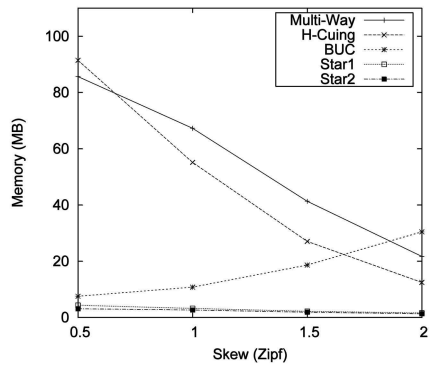


Fig. 18. Runtime with data skew, where $\mathcal{T} = 150K$, $\mathcal{D} = 10$, $\mathcal{C} = 8$, and $\mathcal{M} = 1,000$.

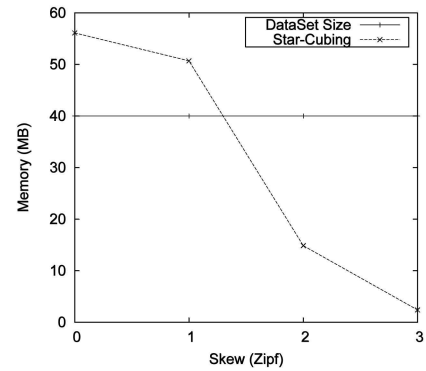


Fig. 20. Memory usage with regard to skew, where $\mathcal{T} = 1M$, $\mathcal{D} = 10$, $\mathcal{C} = 10$, and $\mathcal{M} = 100$.

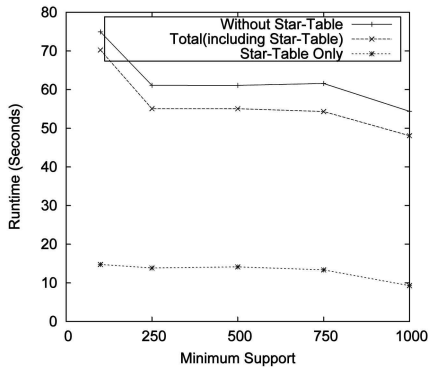


Fig. 21. Runtime with star-table effectiveness, $\mathcal{T} = 1M$, $\mathcal{D} = 10$, $\mathcal{C} = 10$, and $\mathcal{S} = 0$.

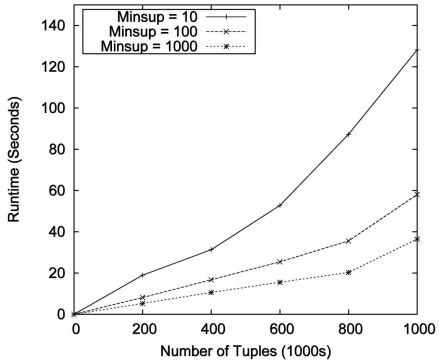


Fig. 22. Scalability with regard to the number of tuples, where $\mathcal{D} = 10$, $\mathcal{C} = 10$, and $\mathcal{S} = 0$.

5.4 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.

Fig. 21 shows the comparison of computation times between `Star-Cubing` (without indexing) 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 reducing the computation time. The proportion of time used in constructing the star-table over the total runtime is less than 30 percent.

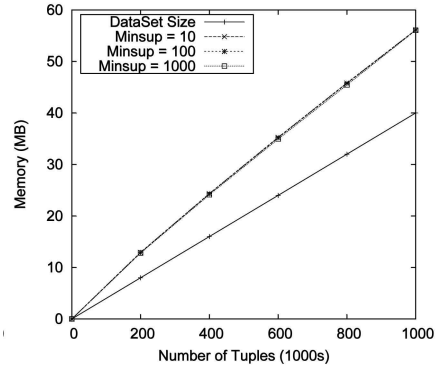


Fig. 23. Memory usage with regard to the number of tuples, $\mathcal{D} = 10$, $\mathcal{C} = 10$, and $\mathcal{S} = 0$.

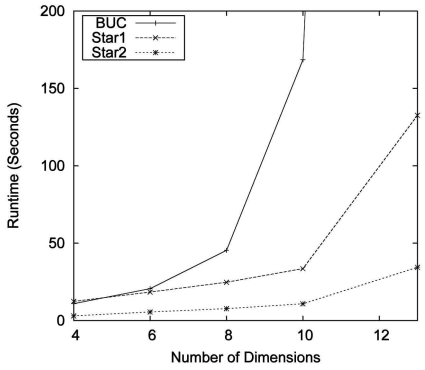


Fig. 24. Star-cubing versus BUC with regard to the number of dimension, real data set, and $\mathcal{M} = 100$.

5.5 Scalability

Using dimension of 10, cardinality of 10, skew of 0, and minimum support of 100, we generated several data sets with up to 1,000K tuples. Fig. 22 shows the scalability of `Star-Cubing` (with indexing) with respect to different min_sup level. The figure shows `Star-Cubing` is scalable with respect to database size. Fig. 23 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 subtrees are relatively small.

5.6 Real Data Set

The first experiment in real data set compares `Star-Cubing` and `BUC` for a different number of dimensions. We picked the first 4, 6, 8, 10, and 13 dimensions in the weather data set which we introduced in the beginning of this section, the min_sup is 100. Fig. 24 shows `Star-Cubing` is substantially better than `BUC`, especially in high dimension data. Fig. 25 shows the computational performance with regard to min_sup. The first 10 dimensions are used here. We

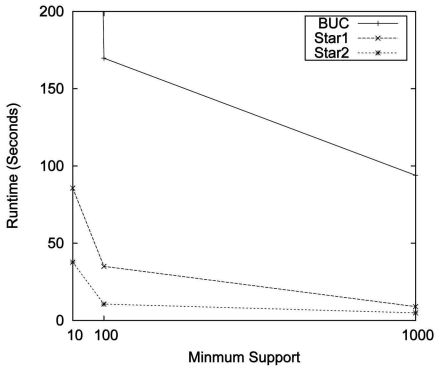


Fig. 25. Star-cubing versus BUC with regard to minsup and real data set.

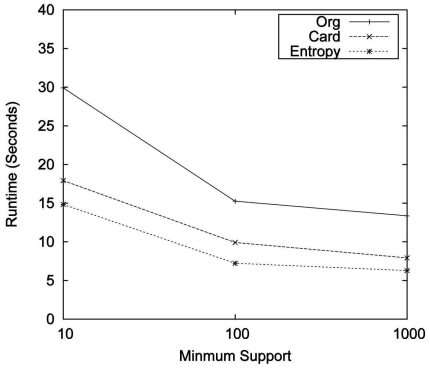


Fig. 26. Dimension order effectiveness with regard to Minsup, where $\mathcal{T} = 1M, \mathcal{D} = 8$.

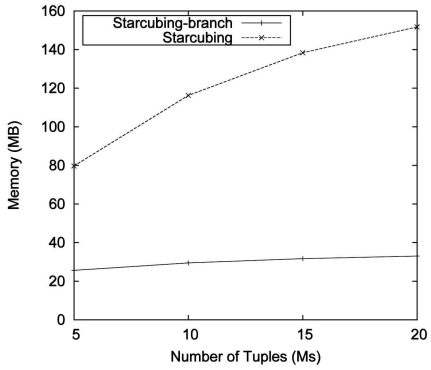


Fig. 27. Peak memory requirement, $\mathcal{D} = 7$, $\mathcal{C} = 10$, $\mathcal{S} = 0$, and $\mathcal{M} = 1,000$.

varied min_sup from 10 to 1,000. Similarly, Star-Cubing outperforms BUC, especially in low min_sup, where Star-Cubing gets more benefits from shared computation.

5.7 Dimension Order

Finally, we tested our dimension ordering strategy. We generate a data set with eight dimensions. Four of them have cardinalities 10 with different skews (0, 1, 2, and 3), the other four have cardinalities 100 with different skews (0, 1, 2, and 3). The min_sup is varied from 10 to 1,000. *Org* is the original dimension order, *Card* means the dimensions are in cardinality-descending order and *Entropy* represents the order we discussed in Section 4.4. Fig. 26 shows that *Card* and *Entropy* obviously outperform *Org*, while *Entropy* is slightly better than *Card*.

In summary, we have tested three cubing algorithms: MultiWay, BUC, and Star-Cubing, with the variations of density, min_sup, cardinality, and skewness. For dense data, Star-Cubing is always the best, MultiWay is 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. Although there is no clear winner for all cases, however, in most cases, Star-Cubing performs better than others.

6 Discussion

In this section, we will discuss a few issues related to Star-Cubing.

6.1 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 data set 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 Fig. 7, Fig. 8, and Fig. 9. 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. Note that the memory management method proposed in Section 4.3 has taken this factor into consideration by designing our own efficient but simple memory management routines.

One may also consider the case that even the much smaller, nonbase 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 [14] and do cubing for each projected database.

We have slightly modified the Star-Cubing algorithm to support large data sets. The new algorithm, we called StarCubing-branch, loads the initial base star-tree branch by branch. The method first scans the whole data set, loads the tuples with one specified value on a dimension, and partitions rest tuples in separated small data files. For example, tuples with value 1 on the first dimension are loaded and tuples with value i ($i \neq 1$) are saved in files $data_i$. When the first branch in the base tree finishes, the released memory can be used to load the second branch from $data_2$.

The experiments were conducted on a computer with 128M physical memory. We compared the performances of Star-Cubing and StarCubing-branch on four data sets with 5M, 10M, 15M, and 20M tuples, respectively. Fig. 27 shows the peak memory requirements. As we analyzed above, the StarCubing-branch consistently uses much less memory. Fig. 28 shows the runtime of the two algorithms. StarCubing-branch consumes more time than Star-Cubing in the cases where the whole data set can fit in the main memory, this is because StarCubing-branch has additional cost on partitioning the original data set into separated files. While the size of data set keeps increase, the Star-Cubing becomes unacceptable due to the limited memory.

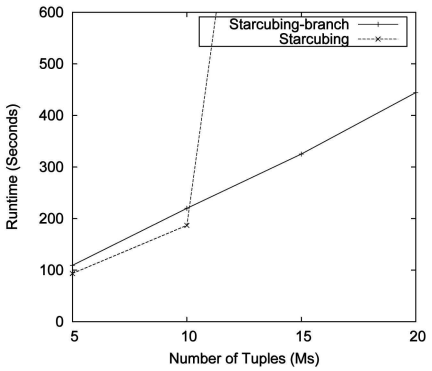


Fig. 28. Runtime, $\mathcal{D} = 7$, $\mathcal{C} = 10$, $\mathcal{S} = 0$, and $\mathcal{M} = 1,000$.

6.2 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 [13].

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-infos at each node of the tree and use the same technique as that proposed in [13], [17] to perform antimonotonicity testing to filter out those unpromising nodes during the cube computation process. Computing other complex measures may adopt the similar techniques suggested in [13].

It should be noted that for nonmonotonic measure such as `average()`, Star-Table does not have the benefit of compressing the tree size. The shared computation can always be achieved if the measure or its converted one is distributive [12].

7 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 antimonotonic property of the iceberg cube measure. No previous cubing method has fully explored both optimization methods in one algorithm.

Our performance study demonstrates that `Star-Cubing` is a promising method. For the full cube computation, if the data set is dense, its performance is comparable with `MultiWay`, and is much faster than `BUC`. If the data set is sparse, `Star-Cubing` is significantly faster than `MultiWay` 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 [16], and discovery-driven exploration of data cubes [22] using the `Star-Cubing` methodology are interesting issues for future research.

Acknowledgments

This work was supported in part by US National Science Foundation grants NSF IIS-03-08215/05-13678 and NSF BDI-05-15813. Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] S. Agarwal, R. Agrawal, P.M. Deshpande, A. Gupta, J.F. Naughton, R. Ramakrishnan, and S. Sarawagi, "On the Computation of Multidimensional Aggregates," *Proc. Int'l Conf. Very Large Data Bases (VLDB '96)*, pp. 506-521, Sept. 1996.

[2] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules," *Proc. Int'l Conf. Very Large Data Bases (VLDB '94)*, pp. 487-499, Sept. 1994.

[3] E. Baralis, S. Paraboschi, and E. Teniente, "Materialized View Selection in a Multidimensional Database," *Proc. Int'l Conf. Very Large Data Bases (VLDB '97)*, pp. 98-12, Aug. 1997.

[4] D. Barbara and M. Sullivan, "Quasi-Cubes: Exploiting Approximation in Multidimensional Databases," *SIGMOD Record*, vol. 26, pp. 12-17, 1997.

[5] D. Barbará and X. Wu, "Using Loglinear Models to Compress Datacube," *Proc. First Int'l Conf. Web-Age Information Management (WAIM '00)*, pp. 311-322, 2000.

[6] K. Beyer and R. Ramakrishnan, "Bottom-Up Computationn of Sparse and Iceberg Cubes," *Proc. ACM-SIGMOD Int'l Conf. Management of Data (SIGMOD '99)*, pp. 359-370, June 1999.

[7] Y. Chen, G. Dong, J. Han, B.W. Wah, and J. Wang, "Multi-dimensional Regression Analysis of Time-Series Data Streams," *Proc. 2002 Int'l Conf. Very Large Data Bases (VLDB '02)*, pp. 323-334, Aug. 2002.

[8] Z. Chen and V. Narasayya, "Efficient Computation of Multiple Group by Queries," *Proc. ACM-SIGMOD Int'l Conf. Management of Data (SIGMOD '05)*, pp. 263-274, June 2005.

[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 SubTotals," *Data Mining and Knowledge Discovery*, vol. 1, pp. 29-54, 1997.

[10] H. Gupta, "Selection of Views to Materialize in a Data Warehouse," *Proc. Seventh Int'l Conf. Database Theory (ICDT '97)*, pp. 98-112, Jan. 1997.

[11] H. Gupta, V. Harinarayan, A. Rajaraman, and J.D. Ullman, "Index Selection for OLAP," *Proc. Int'l Conf. Data Eng. (ICDE '97)*, pp. 208-219, Apr. 1997.

[12] J. Han and M. Kamber, *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2001.

[13] J. Han, J. Pei, G. Dong, and K. Wang, "Efficient Computation of Iceberg Cubes with Complex Measures," *Proc. ACM-SIGMOD Int'l Conf. Management of Data (SIGMOD '01)*, pp. 1-12, May 2001.

[14] J. Han, J. Pei, and Y. Yin, "Mining Frequent Patterns without Candidate Generation," *Proc. ACM-SIGMOD Int'l Conf. Management of Data (SIGMOD '00)*, pp. 1-12, May 2000.

[15] V. Harinarayan, A. Rajaraman, and J.D. Ullman, "Implementing Data Cubes Efficiently," *Proc. ACM-SIGMOD Int'l Conf. Management of Data (SIGMOD '96)*, pp. 205-216, June 1996.

[16] T. Imielinski, L. Khachiyan, and A. Abdulghani, "Cubegrades: Generalizing Association Rules," *Data Mining and Knowledge Discovery*, vol. 6, pp. 219-258, 2002.

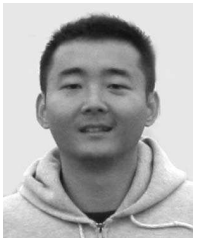
[17] L.V.S. Lakshmanan, J. Pei, and J. Han, "Quotient Cube: How to Summarize the Semantics of a Data Cube," *Proc. Int'l Conf. Very Large Data Bases (VLDB '02)*, pp. 778-789, Aug. 2002.

[18] L.V. S. Lakshmanan, J. Pei, and Y. Zhao, "QC-Trees: An Efficient Summary Structure for Semantic OLAP," *Proc. ACM-SIGMOD Int'l Conf. Management of Data (SIGMOD '03)*, pp. 64-75, June 2003.

[19] X. Li, J. Han, and H. Gonzalez, "High-Dimensional OLAP: A Minimal Cubing Approach," *Proc. Int'l Conf. Very Large Data Bases (VLDB '04)*, pp. 528-539, Aug. 2004.
[20] R. Ng, L.V.S. Lakshmanan, J. Han, and A. Pang, "Exploratory Mining and Pruning Optimizations of Constrained Associations Rules," *Proc. ACM-SIGMOD Int'l Conf. Management of Data (SIGMOD '98)*, pp. 13-24, June 1998.
[21] K. Ross and D. Srivastava, "Fast Computation of Sparse Datacubes," *Proc. Int'l Conf. Very Large Data Bases (VLDB '97)*, pp. 116-125, Aug. 1997.
[22] S. Sarawagi, R. Agrawal, and N. Megiddo, "Discovery-Driven Exploration of OLAP Data Cubes," *Proc. Int'l Conf. Extending Database Technology (EDBT '98)*, pp. 168-182, Mar. 1998.
[23] J. Shanmugasundaram, U.M. Fayyad, and P.S. Bradley, "Compressed Data Cubes for OLAP Aggregate Query Approximation on Continuous Dimensions," *Proc. Int'l Conf. Knowledge Discovery and Data Mining (KDD '99)*, pp. 223-232, Aug. 1999.
[24] A. Shukla, P.M. Deshpande, and J.F. Naughton, "Materialized View Selection for Multidimensional Datasets," *Proc. Int'l Conf. Very Large Data Bases (VLDB '98)*, pp. 488-499, Aug. 1998.
[25] Y. Sismanis and N. Roussopoulos, "The Complexity of Fully Materialized Coalesced Cubes," *Proc. Int'l Conf. Very Large Data Bases (VLDB '04)*, pp. 540-551, Aug. 2004.
[26] Y. Sismanis, N. Roussopoulos, A. Deligianannakis, and Y. Kotidis, "Dwarf: Shrinking the Petacube," *Proc. ACM-SIGMOD Int'l Conf. Management of Data (SIGMOD '02)*, pp. 464-475, June 2002.
[27] J.S. Vitter, M. Wang, and B.R. Iyer, "Data Cube Approximation and Histograms Via Wavelets," *Proc. Int'l Conf. Information and Knowledge Management (CIKM '98)*, pp. 96-104, Nov. 1998.
[28] W. Wang, H. Lu, J. Feng, and J.X. Yu, "Condensed Cube: An Effective Approach to Reducing Data Cube Size," *Proc. Int'l Conf. Data Eng. (ICDE '02)*, pp. 155-165, Apr. 2002.
[29] D. Xin, Z. Shao, J. Han, and H. Liu, "C-Cubing: Efficient Computation of Closed Cubes by Aggregation-Based Checking," *Proc. Int'l Conf. Data Eng. (ICDE '06)*, p. 4, Apr. 2006.
[30] Y. Zhao, P.M. Deshpande, and J.F. Naughton, "An Array-Based Algorithm for Simultaneous Multidimensional Aggregates," *Proc. ACM-SIGMOD Int'l Conf. Management of Data (SIGMOD '97)*, pp. 159-170, May 1997.

Dong Xin received the BS and MS degrees from the Department of Computer Science and Engineering at Zhejiang University in 1999 and 2002, respectively. He is currently a PhD candidate in the Department of Computer Science at the University of Illinois at Urbana-Champaign. His research interests include data mining, data warehousing, and database system. He is a student member of the IEEE and the IEEE Computer Society.

Jiawei Han is a professor in the Department of Computer Science at the University of Illinois at Urbana-Champaign. He has been working on research into data mining, data warehousing, stream data mining, spatiotemporal and multimedia data mining, biological data mining, social network analysis, text and Web mining, and software bug mining, with more than 300 conference and journal publications. He has chaired or served in many program committees of international conferences and workshops. He also served or is serving on the editorial boards for *Data Mining and Knowledge Discovery*, the *IEEE Transactions on Knowledge and Data Engineering*, the *Journal of Computer Science and Technology*, and the *Journal of Intelligent Information Systems*. He is currently serving as founding editor-in-chief of the *ACM Transactions on Knowledge Discovery from Data (TKDD)*, and on the board of directors for the executive committee of ACM Special Interest Group on Knowledge Discovery and Data Mining (SIGKDD). He is an ACM fellow. He has received many awards and recognitions, including ACM SIGKDD Innovation Award (2004) and IEEE Computer Society Technical Achievement Award (2005). He is a senior member of the IEEE.



Xiaolei Li received the BS degree in computer science with a minor in mathematics and the MS degree in computer science from the University of Illinois at Urbana-Champaign in 2002 and 2004, respectively. He is currently a PhD candidate in computer science at the University of Illinois. His research interests include data warehousing, anomaly detection, and spatio-temporal data mining.

Zheng Shao received the BEng degree from Tsinghua University, Beijing in 2003, the MS degree from the University of Illinois, Urbana-Champaign in 2005, and is now on leave of the PhD program at the University of Illinois, Urbana-Champaign. His research interests include data mining, Web mining, and large-scale distributed system. He is currently working at Yahoo! Web Search.

Benjamin W. Wah received the PhD degree in computer science from the University of California, Berkeley, in 1979. He is currently the Franklin W. Woeltge Endowed Professor of Electrical and Computer Engineering and professor of the Coordinated Science Laboratory at the University of Illinois at Urbana-Champaign, Urbana, IL. Previously, he served on the faculty of Purdue University (1979-1985), as a program director at the National Science Foundation (1988-1989), as Fujitsu Visiting Chair Professor of Intelligence Engineering, the University of Tokyo (1992), and McKay Visiting Professor of Electrical Engineering and Computer Science, the University of California, Berkeley (1994). In 1989, he was awarded a University Scholar of the University of Illinois; in 1998, he received the IEEE Computer Society Technical Achievement Award; in 2000, the IEEE Millennium Medal; in 2003, the Raymond T. Yeh Lifetime Achievement Award from the Society for Design and Process Science; and in 2006, the IEEE Computer Society W. Wallace-McDowell Award and the Pan Wen-Yuan Outstanding Research Award. His current research interests are in the areas of nonlinear search and optimization, multimedia signal processing, and computer networks. He cofounded the *IEEE Transactions on Knowledge and Data Engineering* in 1988 and served as its editor-in-chief between 1993 and 1996, and is the Honorary Editor-in-Chief of *Knowledge and Information Systems*. He currently serves on the editorial boards of *Information Sciences*, the *International Journal on Artificial Intelligence Tools*, the *Journal of VLSI Signal Processing*, *World Wide Web*, and *Neural Processing Letters*. He had chaired a number of international conferences, including the 2000 IFIP World Congress and the 2006 IEEE/WIC/ACM International Conferences on Data Mining and Intelligent Agent Technology. He has served the IEEE Computer Society in various capacities, including as Vice President for Publications (1998 and 1999) and President (2001). He is a fellow of the AAAS, ACM, and IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.