C-Cubing: Efficient Computation of Closed Cubes by Aggregation-Based Checking

Dong Xin Zheng Shao Jiawei Han Hongyan Liu
University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA

October 26, 2005

Abstract

It is well recognized that data cubing often produces huge outputs. Two popular efforts devoted to this problem are (1) *iceberg cube*, where only significant cells are kept, and (2) *closed cube*, where a group of cells which preserve roll-up/drill-down semantics are losslessly compressed to one cell. While numerous studies have been reported on the iceberg cube computation, there is only limited work on efficient computation of closed cubes.

Previous work on closedness pruning and checking have developed two different approaches, output-based vs. raw-data-bases. While these method either need to check output data or row data, it contribute a lot to the entire computation time.

In this paper, we propose an aggregation-based approach, named C-Cubing, to compute closed iceberg cube more efficiently. Using this method, neither the existing output nor the raw data is needed to be checked for closedness. Integrating this method with two successful iceberg cubing algorithms, MM-Cubing and Star-Cubing, we propose and implement three algorithms, C-Cubing(MM), C-Cubing(Star) and C-Cubing(StarArray). A through performance study is conducted and shows that this method runs almost one order of magnitude faster then those previous approaches. Finally, how the performance of these algorithms varies with the properties of data set is further demonstrated and analyzed.

1 Introduction

Data cube computation [5] is one of the most essential but expensive operations in data warehousing and OLAP. It is widely recognized that data cubing often produces huge outputs. A lot of efforts have been devoted to this problem. In general, interesting proposals can be classified into the following categories: (1) computation of iceberg cubes with simple or complex measures [3, 8, 14, 20]; (2) computation and indexing of lossless condensed cubes [9, 10, 16, 18]; (3) computation of compressed data cubes by approximation, such as quasi-cube, wavelet cube, and stream cube [2, 4, 13, 17]; and (4) computation of partial cubes with selective materialization [1, 6, 11, 15].

A	B	C	D
$a1$	$b1$	$c1$	$d1$
$a1$	$b1$	$c1$	$d3$
$a1$	$b2$	$c2$	$d2$

Table 1: Example of Closed Iceberg Cube

We are interested in a thorough study of efficient computation of *closed iceberg cubes*. Before formally defining the problem, we first introduce the following definitions.

Definition 1 *(Group-By Cell) In an n-dimension data cube, a cell* $c = (a_1, a_2, \ldots, a_n : m)$ *(where* m *is a measure) is called a* k-dimensional group-by cell *(i.e., a cell in a* k*-dimensional cuboid), if and only if there are exactly* k $(k \leq n)$ *values among* $\{a_1, a_2, \ldots, a_n\}$ *which are* not $*$*. We further denote* $M(c) = m$ *and* $V(c) = (a_1, a_2, \ldots, a_n)$*.*

Definition 2 *(Iceberg Cell) Given a threshold constraint on the measure, a cell is called* iceberg cell *if it satisfies the constraint. A popular iceberg constraint on measure count is* $M(c) \geq$ min_sup*, where* min_sup *is a user-given threshold.*

Definition 3 *(Closed Cell) Given two cells* $c = (a_1, a_2, \ldots, a_n, m)$ *and* $c' = (a'_1, a'_2, \ldots, a'_n, m')$*, we denote* $V(c) \leq V(c')$ *if for each* a_i $(i = 1, \ldots, n)$ *which is not* $*$*,* $a'_i = a_i$ *(the equality holds iff* $V(c) = V(c')$*). A cell* c *is said to be* covered *by another cell* c' *if* $\forall c''$ *such that* $V(c) \leq V(c'') < V(c')$*,* $M(c'') = M(c')$ *. A cell is called a* closed cell *if it is not covered by any other cells.*

Given the above definitions, the problem of closed iceberg computation is to compute all closed cells which satisfy the iceberg constraints. An example of the closed iceberg cube is given in Example 1. The closed cube has been shown as a lossless and effective compression of the original data cube in [9, 10].

Example 1 *(Closed Iceberg Cube) Table 1 shows a table(with four attributes) in relational database, assume the measure is count, the iceberg constraint is count* ≥ 2*. Then* $cell_1 = (a_1, b_1, c_1, * : 2)$*, and* $cell_2 = (a_1, *, *, * : 3)$ *are closed iceberg cells; but* $cell_3 = (a_1, *, c_1, * : 2)$ *and* $cell_4 = (a_1, b_2, c_2, d_2 : 1)$ *are not, because the former is covered by* $cell_1$*, whereas the latter does not satisfy the iceberg constraint.* ■

In this paper, we first assume that the measure m is *count*, and the iceberg condition is (*count* $\geq$ *min_sup*). Lemma 1 shows that *count* is the fundamental measure comparing with others. We will discuss the extension to other measures in Section 6.

Lemma 1 *If a cell is not closed on measure* count*, it cannot be closed with respect to any other measures.*

Rationale. If a cell c_1 is not closed, there must be another cell c_2 which covers c_1. Since both c_1 and c_2 are aggregated by the same group of tuples, they will get the same value for any other measures. One can easily verify that if the measure m is strictly monotonic or anti-monotonic, then closedness on measure *count* is equivalent to measure m. ■

While the iceberg cubing methods have been extensively studied, there are limited efforts on closed iceberg cubing algorithms. Previous studies on closed cubing or closed frequent pattern mining have developed two major approaches, *output-based checking* vs. *raw-data-based checking*. The former, represented by most closed pattern mining algorithms [19, 21], indexes the already found closed patterns by a tree or hash-table, to check the closedness of the later outputs. This method is not desirable in cubing problem, since normally the output of cubing can be very large, and maintaining the index structure would become the major bottleneck of the algorithm. The latter, represented by Quotient Cube approach [9, 10], needs to scan the raw data partition repeatedly to collect the closed information whenever there is an output, which also adds considerable overheads.

In this paper, we propose a new closed pruning and checking method, C-Cubing, which checks neither the output space nor the raw data. With a very small overhead, the closedness information of a cell can be aggregated as a measure, and the closedness checking can be simply done by testing this special measure. We call our approach as *aggregation-based checking*. The new method C-Cubing is implemented in two successful iceberg cubing algorithms: MM-Cubing and Star-Cubing. After running a comprehensive performance evaluation, we found although C-Cubing is superior over the previous approach, the alternatives compete each other on different kinds of data sets. We further study how the performance of these algorithms varies with respect to the properties of the data set and provide suggestions to choose the best-fit one in applications.

The remaining of the paper is organized as follows. In Section 2, we give an overview of the previous related works. In Section 3, we introduce the *aggregation-based closed checking* approach, and integrate it into MM-Cubing. Section 4 proposes an extended version of Star-Cubing and exploits more closed pruning techniques. Our performance study on both synthetic and real data sets is shown in Section 5. A discussion on potential extensions is in Section 6. We conclude our study in Section 7.

2 Algorithms Overview

In this section, we analyze the popular iceberg cubing and closed cubing related algorithms.

2.1 Iceberg Cubing Algorithms

2.1.1 BUC

BUC [3] employs a bottom-up computation by expanding dimensions. Cuboids with fewer dimensions are parents of those with more dimensions. BUC starts by reading the first

dimension and partitioning it based on its distinct values. For each partition, it recursively computes the remaining dimensions. The bottom-up computation order facilitates the Apriori-based pruning: The computation along a partition terminates if its count is less than *min_sup*.

Apriori pruning reduces lots of unnecessary computation and is effective when the dataset is sparse. However, BUC does not share the computations, but sharing is very useful in computing dense datasets. We will review a closed cube method [9, 10] based on BUC later in this section.

2.1.2 StarCubing

Star-Cubing [20] uses a hyper-tree structure, called star-tree, to facilitate cube computation. Each level in the tree represents a dimension in the base cuboid. The algorithm takes advantages of shared computation and Apriori pruning. In the global computation order, it uses simultaneous aggregation. However, it has a sub-layer underneath based on the bottom-up model by exploring the notion of shared dimension, which enables it to partition parent group-by's and use the Apriori-based pruning on child group-by's.

Star-Cubing performs well on dense, skewed and not-so-sparse data. However, in very sparse data sets, *e.g.*, the cardinalities of dimensions are large, the star tree gets wider. It requires more time in construction and traversal. In this paper, we first extend the original algorithm for the efficient computation in sparse data, then discuss the closed Star-Cubing method.

2.1.3 MM-Cubing

Different from BUC and Star-Cubing, where the computation order follows the order of dimensions, MM-Cubing [14] factorizes the lattice space into one dense subspace and several sparse subspaces, according to the frequency of values. Heuristics are designed to make the dense subspace reasonably small such that the MultiWay [22] array computation can be effectively applied on. The remaining sparse subspaces are recursively partitioned into dense and sparse parts.

MM-Cubing partitions the data dynamically, which makes it highly adaptive to dense, sparse, or skewed data sets. However, the dynamic partition also shatters the internal relationship between dimensions, as a result, the closed pruning and checking become more difficult.

2.2 Algorithms related to closed cubing

2.2.1 Quotient Cube

In Quotient Cube [9, 10], all the cube cells with identical aggregation value are grouped into the same class, preserving the roll-up/drill-down semantics. Each class is represented by an upper bound cell and several lower bound cells. According to our definition, the upper bound cell has the same meaning as the closed cell. Quotient Cube finds all upper bound

cells using a depth-first search algorithm (we refer as QC-DFS) derived from BUC [3]. To ensure that each output cell is an upper bound, it scans all the dimensions which are not within the group-by conditions, in the current data set partition. If a dimension is found where all tuples share the same value, the algorithm decides either to extend the current cell by fixing the value on the dimension, or to stop further computing the current partition since it is previously computed.

The major overhead that QC-DFS applies on BUC is the scanning. Although the scanning can be terminated earlier when the first discrepancy is found, the amount of the work is still considerably large. The algorithm will have to scan the whole partition if there does exist a common shared value on a dimension.

2.2.2 Closed frequent Pattern Mining

Closed frequent pattern mining does not fall into the category of closed iceberg cubing, but they have some common properties with respect to closed checking and pruning. Despite of the difference of vertical approach ([21]) or horizontal approach ([19]), they usually explore two kinds of strategies for efficiently mining of closed itemsets. First, internal closed pruning methods are developed to reduce the potential number of closed checking in output; Second, an external checking architecture such as a tree or hash-table is built on already found outputs, new output will have to check closedness with it to avoid redundant outputs.

The external checking method is not suitable for cubing. This is because the size of a data cube, even for the closed one, could be possibly much larger than the original data set. It is undesirable to maintain all the closed outputs in memory.

In summary, we believe that the current closed cubing algorithms are unsatisfactory. It is necessary to exploit more efficient methods. In the following sections, we propose an aggregation-based closed checking method, called C-Cubing, and integrate it with MM-Cubing and Star-Cubing. We choose these two algorithms because (1) they are successful iceberg cubing methods according to the literature [14, 20], and (2) they represent two different computational approaches (the former is subspace-based, whereas the latter dimension-based), which, as we will see, have different influences on closed extensions.

3 Closed MM-Cubing

MM-Cubing is one of the most adaptive iceberg cubing algorithm. Based on the recent report [14], it outperforms previous algorithms in most cases. Thus, we first develop C-Cubing(MM) by adapting MM-Cubing to closed cube computation.

In this section, we first analyze the challenges to check closedness in MM-Cubing, then introduce a closedness measure to solve the problem and finally discuss the implementation of C-Cubing(MM) and its efficiency.

3.1 Shared Computation vs. Closed Checking

In order to prune the non-closed cells, we need to check the actual tuple values on the $*$ dimensions of that cell. Since all $*$ dimensions eventually come out from the dense space (recursive call on sparse spaces does not produce $*$ dimensions), we only need to check that inside the dense subspace. Unlike QC-DFS [10] where the original tuples can be traced and scanned to check closedness, there are no such tuple lists available for each cell in MM-Cubing. Because MM-Cubing use MultiWay [22] aggregation inside the dense space, keeping the tuple ID list for each cell not only introduces computational overhead, but also has storage challenges. Thus, the problem turns out to be how to check closedness in MultiWay.

The major advantage of MultiWay in cube computation is that MultiWay adopts a simultaneous aggregation approach to share computation among different cuboids. Since we do not keep tuple ID list as in BUC, it is impossible to check the closedness in the last moment just before outputting the cell. Based on the computation order of MultiWay, naturally we get this idea: *Is it possible to compute the closedness information along with the support aggregation?* Given the multi-dimensional aggregation order, this question is equivalent to: *Instead of computing the closedness with a list of tuple IDs, is it possible to compute it incrementally by keeping a summary at each cell?* The answer to this question is yes, and the summary turns out to be a *closedness measure*.

3.2 Closedness Measure

A closedness measure is a value which indicates whether the cell is closed. Before getting into the details of the algorithm, let us first have a look at what kind of measures can be computed incrementally.

Definition 4 *(Distributive Measure) A measure is called* distributive*, if the measure of the whole data set can be computed solely based on the measures of the parts of that data set.*

Definition 5 *(Algebraic Measure) A measure is called* algebraic*, if the measure can be computed based on a bound number of measures of the parts of that data set.*

Example 2 *It is easy to see that min, count, and sum are distributive measures, e.g., $count(A \cup B) = count(A) + count(B)$, in which A, B denote different parts of the data set; while avg is an algebraic measure since $avg(A \cup B) = (sum(A) + sum(B))/(count(A) + count(B))$.*

Is *closedness* a distributive or an algebraic measure? First, it is not distributive. This can be shown with a simple example: In order to compute the closedness of $(*, *, 1)$, we check the closedness of $(*, 1, 1)$ and $(*, 2, 1)$. Suppose both $(*, 1, 1)$ and $(*, 2, 1)$ are not closed. It may imply several cases. One case is that we have two tuples $(1, 1, 1)$ and $(2, 2, 1)$. In this case, $(*, *, 1)$ is closed. Another case is that we have two tuples$(1, 1, 1)$ and $(1, 2, 1)$, in which case $(*, *, 1)$ is not closed. Thus, it is impossible to derive the closedness of a cell based on the closedness of the subcells. Second, closedness is an algebraic measure because it can be

computed based on a distributed measure *Representative Tuple ID* and an algebraic measure *Closed Mask*.

Definition 6 *(Representative Tuple ID) The* Representative Tuple ID *of a cell is the smallest ID of the tuples that aggregate to this cell. In the case the cell is empty (does not contain any tuple), the Representative Tuple ID is set to a special value NULL.*

Lemma 2 *Representative Tuple ID is a distributive measure.*

Rationale. In fact, Representative Tuple ID is the minimum in the set of all related Tuple IDs. We already know that *min* is a distributive measure. Thus, Representative Tuple ID is a distributive measure. ∎

Definition 7 *(Closed Mask) The* Closed Mask *of a cell contains D bits, where D is the number of dimensions in the original database. The bit is 1 if and only if all the tuples belonging to that cell have the same value in the corresponding dimension.*

Lemma 3 *Closed Mask is an algebraic measure.*

Proof. We show how to use the Closed Masks and the Representative Tuple IDs of subsets to compute the Closed Mask for the whole set.

We denote the subsets by $S_i,\ i \in \{1, 2, \ldots, k\}$, and the whole set by S, clearly, $S = \bigcup_{i=1}^{k} S_i$. Let $C(S_i, d)$ be the Closed Mask of set S_i on dimension d, $T(S_i)$ be the Representative Tuple ID of set S_i, and $V(t, d)$ be the value of Tuple t on dimension d.

By the definition of Representative Tuple ID, we have $T(S) = min_{i=1}^{k} T(S_i)$. From the definition of Closed Mask, the bit is 1 if and only if all the tuples belonging to that cell have the same value in the corresponding dimension, in which case the following two conditions hold: (1) all the Closed Masks of the subsets should have 1 on that bit, and (2) all the Representative Tuples of the subsets have the same value on that dimension. This leads to the following equation:

$$C(S, d) = \prod_{i=1}^{k} C(S_i, d) \times Eq(|\{V(T(S_i), d),\ 1 \le i \le k\}|, 1)$$

where $|\{V(T(S_i), d),\ 1 \le i \le k\}|$ means the number of distinct values in the set $\{V(T(S_i), d),\ 1 \le i \le k\}$, and $Eq(x, y)$ is 1 if x is equal to y, otherwise it evaluates to 0.

This equation defines how to compute the Closed Mask incrementally based on Closed Masks and Representative Tuple IDs of the subsets. Thus, Closed Mask is an algebraic measure. ∎

Please note in all these equations, multiplication can be replaced by a bit-and operation, which is more efficient. We may also notice that Representative Tuple ID can be the ID of any tuple related to the cell. We use the minimum intentionally to ease problem formulation and discussion.

Now we are able to define the closedness measure.

Definition 8 *(All Mask) The* All Mask *of a cell contains D bits, where D is the number of dimensions. The bit is 1 if and only if that cell has a value of $*$ in the corresponding dimension.*

Definition 9 *(Closedness Measure) Given a cell, whose closed mask is C and all mask is A, the* closedness measure *is defined as C&A, where & is bitwise-and operations.*

Example 3 *The All Mask of a cell $(*, *, 2, *, 1)$ is $(1, 1, 0, 1, 0)$. Please note All Mask is a property of the cell and can be computed directly. If the closed mask of this cell is $(1, 0, 1, 0, 0)$, then its closedness measure value is $(1, 0, 0, 0, 0)$.*

Since the *all mask* can be directly computed from the cell and the *closed mask* is an *algebraic measure*, we conclude that the *closedness* is also a *algebraic measure*.

Lemma 4 *Closedness is an algebraic measure.*

A cell is non-closed if and only if in at least one dimension of its closedness measure, the bit is 1. Otherwise, the cell is closed. The intuitive explanation is that the cell is non-closed if and only if (1) all tuples of the cell have the same value in a dimension, and (2) the value of the cell in that dimension is $*$.

Since closedness of a cell is an algebraic measure, we can compute it incrementally in the simultaneous aggregation steps of the dense subspace. Before the output step, we need to check the closedness of the cell. We will output the cell only if it is closed.

3.3 Implementation of C-Cubing(MM)

In this section, we will discuss how to implement the closedness measure in MM-Cubing. The new algorithm is called C-Cubing(MM).

As we explained earlier, MM-Cubing partitions the whole data into several subspaces according to the value frequency. The dense subspace will be computed directly by MultiWay array aggregation, while the sparse subspaces are recursively partitioned to get the next level subspaces until the size of the sparse subspace is less than *min_sup*. Since the subspaces are not mutually exclusive, they may contain values which are not within current computation interest. To avoid duplicate outputs, MM-Cubing will temporarily change these values to a special identifier, and switch back after the current subspace is finished. Although this method is efficient in the sense of space and computational requirement, it brings problems to closed checking. When closedness measure checks the values by retrieving the representative tuple, it may find the special identifier, instead of the original value. This is undesirable for closed checking. To solve this problem, C-Cubing(MM) uses a table of *Value Mask*, which keeps the information of whether a value is set to the special identifier. The original tuples are kept unchanged, and the closedness measure can be computed.

The integration of closedness measure into MM-Cubing is as follows. Whenever there is an aggregation of count, we aggregate closedness measure as well. When a cell is ready to output, the closedness measure is checked. Since we do not do any less computation (though

sometimes we do not output the cell, which means less I/O operations), we can expect that C-Cubing(MM) always has overheads over MM-Cubing.

However, we can also expect that C-Cubing(MM) will not degrade too much from MM-Cubing. For space efficiency, the additional data structures used are Value Mask, Closed Mask and Representative Tuple ID. The size of Value Mask is $\sum_{i=1}^{D} C_i$ bits, which is quite small compared to other data structures (where D is the number of dimensions and C_i is the cardinality for i^{th} dimension). The sizes of the Close Mask and the Representative Tuple ID are both proportional to the size of the aggregation table, which is generally limited to 4MB. For time efficiency, the Value Mask introduces a checking on every access to the tuple data. The cost is proportional to the number of data accesses. The Closed Mask and Representative Tuple ID are both aggregated in the same way as the support. Thus, the additional cost is proportional to the existing cost of aggregation. In a word, these modifications do not introduce large space or time overheads.

The technique we use in C-Cubing(MM) is called *Closed Checking*, which means that we check the closedness of a cell just before the output step. Correspondingly, when a new partition (or child tree in Star-Cubing) is to be computed, if we can identify in advance that all the cells which will be generated by this partition are not closed, then the whole partition can be simply bypassed. We call this task as *Closed Pruning*. Obviously, closed pruning is more promising. We will discuss how to achieve closed pruning with Star-Cubing in the next section.

4 Closed StarCubing

We select Star-Cubing for closed cubing study because we believe tree-like data structure is useful for closed pattern mining, since data are no longer individual tuples, but are organized by trees, and their relationships can be partially implied by the tree structure. In this section, we discuss closed iceberg cubing algorithm derived from Star-Cubing.

As seen in Section 2, Star-Cubing is inefficient in sparse data set due to the high cost in the tree operations. To lift this restriction, we extend the original Star-Cubing algorithm in two ways: First, we use a new data structure called *StarArray* to represent a cuboid tree; second, we exploit a new computational order to generate child trees. Closed pruning and checking methods are further applied on both the original Star-Cubing and extended StarArray algorithms.

Before we discuss the detail of the algorithm, it is necessary to first clarify the related terminology. In this paper, we use *father* and *son* when we refer to *node*, *parent* and *child* when we talk about *tree*. A tree is called *base tree* if it contains all tuples with full dimensions.

4.1 StarArray

Star-Cubing algorithm uses the star tree to completely represent the whole data set. However, it is quite clear that only upper-level nodes in the tree gain from the shared aggregation. To avoid high maintenance cost in the lower level part, we introduce a hybrid structure,

StarArray. A StarArray is constituted by a couple $\langle A, T \rangle$, where A is an array storing tuple IDs, and T is a partial tree whose leaf nodes pointing to a continuous subpool of A. Different from Star-Cubing where the nodes are constructed until the end of the dimension is reached, in StarArray, when we find a node whose aggregation value is less than *min_sup*, all the subbranches below this node are truncated. Instead, the corresponding tuple IDs are copied into A, and the leave node will have two pointers pointing to the start and end positions in A.

Example 4 *(Star Array) Given a data set (as in Fig. 1) and* $\min\_\sup(M = 3)$ *for iceberg cubing, the base StarArray and a child StarArray* CDE/a_1 *created from node* a_1 *are shown as Fig. 1, each leaf node points to a sub pool of the array. The tuple IDs in the array are partially ordered according to the remaining dimensions which were not shown in the tree. For example, the* c_1 *node in the base tree points to* t_1, t_2*, which are first ordered by dimension* D*, then ordered by dimension* E*. Similarly, for the* d_1 *node in* CDE/a_1 *tree,* t_4 *is placed before* t_1 *because of the ordering in dimension* E*.*

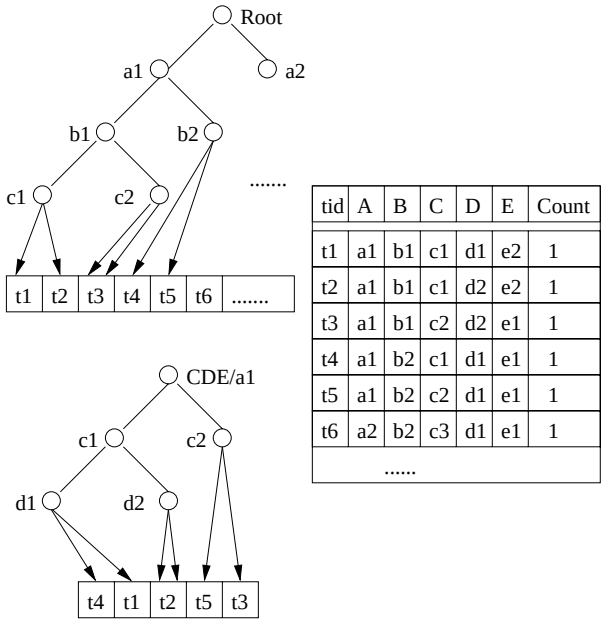

Figure 1: **StarArray: The base tree and a child tree**

4.2 MultiWay Traversal

Given the new StarArray data structure, we discuss how to efficiently create child trees from a parent tree. One of the major benefit of Star-Cubing is using *Multiway Aggregation*, that is, the parent tree only need to be traversed once to create all child trees. However, there is a hidden overhead on traversing child trees multiple times in the process of its creation.

We propose a new method called *Multiway Traversal*, where the parent tree is traversed multiple times(with one time for each child tree), but each child tree will be only traversed once during the process of creation. We first explain how the multiway traversal works, then show that in sparse data, the multiway traversal method is more efficient than multiway aggregation.

In multiway traversal, child trees are created and aggregated one by one. Take the child tree CDE/a_1 in Fig. 1 as an example, this child tree is created by collapsing dimension B under node a_1 in the parent tree. In order to traverse CDE/a_1 only once during its creation, we need to know the final aggregate value of each node of CDE/a_1 when they are first created. The way to achieve it is to simultaneously traverse all branches starting from the collapsed dimension in the parent tree. In Fig. 1, they are branches starting from nodes b_1 and b_2. These two depth-first traversals are synchronized so that they reach the nodes with the same value at the same time. For example, when b_1 traversal reaches c_1 node, b_2 traversal gets a pool of tuple ID. Since all the tuple ID are ordered, b_2 traversal is able to find the subpool of c_1 quickly by a simple sequential scan. The c_1 node of CDE/a_1 is constructed and the aggregate value can be filled. If the aggregate value is no less than *min_sup*, the depth-first traversals will continue to look for d_1. Again, the partial ordering property of the tuple IDs will facilitate to get the aggregate value of d_1 in b_2 traversal. After creating d_1 node in CDE/a_1 child tree, the algorithm notices that the aggregate value is less than the *min_sup*, and node d_1 in CDE/a_1 is a leaf node, the corresponding tuple IDs are copied to the tuple ID array of CDE/a_1 child tree. These tuple IDs are sorted based on the partial ordering on the dimension E. Because all these tuple ID pools are already partially ordered in the parent tree, we use a multiway merge sort to reorder these pools. To get the partial ordering on deeper dimensions, the merge sort is invoked recursively in a depth-first way. For our particular situation, the cost of merge sort on each dimension is linear to the number of tuples. Due to the size limitation, we do not discuss the detail of sorting in this paper.

We now compare the computational cost of *multiway aggregation* and *multiway traversal*. Intuitively, in sparse data set, the size of child trees are relatively large since there are not many shared values. The hidden cost to traversal the child tree multiple times would be relatively high. We show that this is true by a detail comparison of the computational costs of multiway aggregation and mutliway traversal.

For simplicity, we assume *min_sup* is 1, so that every node in the parent tree (except the nodes at the last two levels, which can be output directly) will create a child tree, and the difference between tree and array can be ignored (since when *min_sup* is 1, StarArray is identical to a star tree). Consider a parent tree with depth k, root is at level 0, and leaves are at level k. Assume that the number of nodes at level i are $N_i, i = (0, 1, \ldots, k)$, clearly, $N_0 = 1$. From the parent tree, we will construct N_0 child trees with depth $k-1$ by collapsing the first dimension; N_1 child trees with depth $k-2$ by collapsing the second dimension, and so on. We use the tree size to estimate the traversal cost. Let S_i^p be the average size of subtrees whose roots are at level i in the parent tree (*i.e.*, S_0^p is the full parent tree size, S_1^p is the average size of subtrees rooted at the first level nodes). For each node at level i of the parent tree, a child tree with average size S_{i+1}^c will be created (*i.e.*, S_1^c is the size of the child

tree created by the root of the parent tree). Note the changes on footprint implies that one dimension was collapsed.

In Star-Cubing, the multiple aggregation method traverses the parent tree once, the cost is S_0^p. For each $i \in \{0, 1, \ldots, k-2\}$, there will be N_i child trees. During the construction, each child tree will be traversed k times, where k is the number of the nodes on the collapsed dimension. On average, $k = \frac{N_{i+1}}{N_i}$. Since the child tree is expanding, and the traversal cost starts from S_{i+1}^p, and ends with S_{i+1}^c, we estimate the average cost as $\overline{S_{i+1}} = \frac{S_{i+1}^p + S_{i+1}^c}{2}$. The computational cost of multiway aggregation $C(MA)$ is:

$$S_0^p + \sum_{i=0}^{k-2} N_i \times \overline{S_{i+1}} \times \frac{N_{i+1}}{N_i} = S_0^p + \sum_{i=1}^{k-1} N_i \times \overline{S_i}$$

In StarArray, the multiple traversal method traverses the child tree once. For each child tree, the cost is $S_i^c, i \in \{1, 2, \ldots, k-1\}$, the traversal cost on the parent tree is S_{i-1}^p. Since there are N_{i-1} number of child trees with size S_i^c, the computational cost of multiway traversal $C(MT)$ is:

$$\sum_{i=1}^{k-1} (N_{i-1} \times (S_i^c + S_{i-1}^p))$$

The difference of these two, *i.e.*, $C(MT) - C(MA)$:

$$\sum_{i=1}^{k-1} (N_{i-1} \times (S_i^c + S_{i-1}^p) - N_i \times \overline{S_i}) - S_0^p$$

$$= \sum_{i=1}^{k-1} (N_i \times S_i^p + N_{i-1} \times S_i^c - N_i \times \overline{S_i}) - N_{k-1} \times S_{k-1}^p$$

$$\leq \sum_{i=1}^{k-1} (2 \times N_i \times S_i^p - N_i \times \overline{S_i}) - N_{k-1} \times S_{k-1}^p$$

Where the first equality is true because $N_0 = 1$, the second inequality is true because each child tree with size S_i^c is aggregated by $\frac{N_i}{N_{i-1}}$ number of subparent trees with size S_i^p, thus S_i^c is at most the sum of them (*i.e.*, $S_i^c \leq \frac{N_i}{N_{i-1}} \times S_i^p$).

In the case where the data set is sparse, the value S_i^p is much less than S_i^c (since there are not many shared values during the aggregation), thus $2 \times S_i^p \ll \overline{S_i}$. we conclude in sparse data set, the multiway traversal is more efficient. Note this does not imply that in dense data, the conclusion is true. In fact, later in the performance study, we will see that in dense data set, multiway aggregation is better.

4.3 Closed Pruning by Closedness Measure

In Section 3, we have seen the closedness measure can be efficiently integrated into the previous algorithms to get a closed version. Using the similar technique, we can aggregate

closedness measure in Star-Cubing. In this section, we first discuss how closedness measures are computed, we then exploit more efficient methods by closed pruning.

Star-Cubing uses a tree structure where each node keeps the count measure, which corresponds to the set of aggregated tuples. We add the closedness measure in each node. Initially, every node in the base tree holds a closedness measure, which includes a couple *Closed Mask* and *Representative Tuple ID*. Given a dimension order, each level of the tree corresponds to one specific dimension. Assume the node under investigation is on level i (*e.g.*, belong to dimension i), then the Closed Masks from 1 to i bits are set as "1", leaving the others as "0". For example, in Fig. 1, node c_1 in the base tree has Closed Mask $(1, 1, 1, 0, 0)$, which means all the tuples aggregated to the node c_1 (*i.e.*, tuples t_1 and t_2) share the same value on the first three dimensions. The representative tuple ID of c_1 is chosen as t_1. Note the Closed Masks in nodes are partial in the way that they only keep the closed information of the former dimensions, we adopt this approach for three reasons. First, the Star-Cubing algorithm only outputs the cells at the last two levels, where the Closed Masks are either complete (*i.e.*, at the leaves) or easy to extend to a complete one (*i.e.*, the last second level). Internal nodes will never output a cell. Thus it is safe to allow a partial closed information. Second, the closed information on the former dimensions can be got without additional work (just following the tree structure), while the closed information on the later dimensions need a tree traversal, which clearly has considerable overhead. Finally and most importantly, partial closed information is already enough for closed pruning.

We show how the closedness measure is aggregated on trees. Similar to the *All Mask* in Section 3.2, a *Tree Mask* is assigned on each tree. The tree mask contains D bits, where D is the number of dimensions. The tree mask of the base tree has "0" on every bits. Whenever there is a child tree created, it first inherits the tree mask of the parent tree, then switches the bit on the collapsed dimension, from "0" to "1". For example, in Fig. 1, the CDE/a_1 child tree has tree mask $(0, 1, 0, 0, 0)$, since the dimension B was collapsed. Assume node N is aggregated by nodes $N_i, i \in \{1, 2, \ldots, k\}$, we use the same notations as in lemma 3, that is, $C(N_i, d)$ is the closed mask on dimension d of node N_i, $T(N_i)$ is the representative tuple ID of node N_i, $V(t, d)$ is the value of tuple t on dimension d, and $Eq(x, y) = 1$, if $x = y$; otherwise, it is 0. We further define $TM(d)$ is the value of tree mask on dimension d. We have:

$$C(N, d) = \begin{cases} \prod_{i=1}^{k} C(N_i, d) & if \;\; TM(d) = 0 \\ \prod_{i=1}^{k} C(N_i, d) \times Eq(|\{V(T(N_i, d),\ 1 \le i \le k\}|, 1) & \\ & otherwise \end{cases}$$

The above equation means: If $TM(d) = 0$, then dimension d is not a (previously) collapsed dimension. We should reserve the partial closed mask to be consistent with tree structure. If $TM(d) = 1$, and further if the Closed Masks of all nodes agree on dimension d where they all have value "1", we check the number of distinct values on dimension d. If it is larger than 1, the corresponding bit is set to "0"; otherwise, the bit will be set as "1". As an example, the Closed Masks of node c_1 and d_2 of child tree CDE/a_1 are $(1, 0, 0, 0, 0)$ and $(1, 1, 1, 0, 0)$, respectively.

Based on the Closed Mask and Tree Mask, we have developed two lemmas to facilitate

the closed pruning.

Lemma 5 *(Closed Pruning by Closed Mask) In closed cube computation, assume the Closed Mask of a node is C, the Tree Mask is TM, if $C\&TM \neq 0$ (where & is a bitwise-and operation), then all the cells which are output by the branches under this node or by the child trees created from the branches under this node are non-closed. Particularly, if the node is at the last level of the tree, the output is pruned.*

Rationale. If $C\&TM \neq 0$, then there is at least one previously collapsed dimension (say d), on which all the tuples aggregated to the current node share the same value v_d. Since this dimension has been collapsed, all the cells that will be output by the branch under the current node or by child trees created from this branch (including current node) have value $*$ on the dimension d. However, for each such cell, there is at least a cell (*i.e.*, the cell whose value on the dimension d is v_d) covers it. That is, all the cells are not closed. Hence, it is safe to prune the unnecessary computation on the whole partition. Particularly, if the node is at the last level of the tree, then the output cell is not closed. ■

Before we discuss the second pruning rule, we first introduce a special subtree structure, *Single Path*. A *Single Path* is a branch of a tree where each node in the path has only one son.

Example 5 *(Single Path) In Fig. 2, a_1b_1 is a single path. c_1 does not belong to the single path, since c_1 has two sons d_1, d_2.*

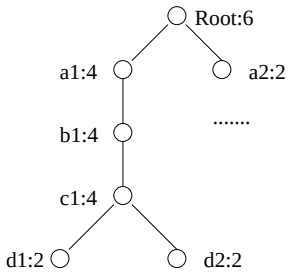

Figure 2: **Closed Pruning by Single Path**

Lemma 6 *In closed cube computation, if a node belongs to a single path, then all the cells which are output by the child tree created from this node are non-closed. Particularly, if the node is at the last second level, then the output is pruned.*

Rationale. New child tree created from a node F is obtained by collapsing its son nodes. If a node belongs to a single path, then this node has only one son S. Assume the corresponding dimension is d_s, the tree mask of the new child tree on bit d_s will be set as "1". On the other hand, all the nodes in the new child will share the same value on dimension d_s, that is, the d_s bit of their closed mask will be set as "1", according to Lemma 5, all the nodes in the child tree will be pruned. ■

Since we do closed pruning on all output levels, the closed checking is complete. The possible efficiency comes from the closed pruning of internal nodes. We have applied the above closed pruning method to both the original and the extended Star-Cubing. The two new algorithms are named as C-Cubing(Star) and C-Cubing(StarArray), respectively.

5 Performance Analysis

To check the efficiency and scalability of the proposed algorithm, a comprehensive performance study is conducted by testing our implementations of Closed Iceberg Cubing. We compare the new methods with QC-DFS, which, to our knowledge, is the best available method on closed cubing. C-Cubing(MM), C-Cubing(Star), and C-Cubing(StarArray) were coded using C++, the QC-DFS was provided by the author of [10]. The experiments were carried out on an Intel Pentium-4 3.2GHz system with 1G of RAM running windows XP. The times recorded include both the computation time and the I/O time. Similar to other performance studies in cube computation [22, 3, 8, 20, 14], all the tests used the data set that could fit in main memory.

We have conducted experiments on both synthetic and real datasets. In the synthetic experiment part, $\mathcal{D}$ 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.

In the real experiment part, we use the weather dataset SEP83L.DAT in [7], which has 1,002,752 tuples with selected 8 dimensions. The attributes (cardinalities) are as follows: year month day hour (238), latitude (5260), longitude (6187), station number (6515), present weather(100), change code (110), solar altitude (1535) and relative lunar illuminance (155).

5.1 Computing Full Closed Cube

The first set of experiments compare four algorithms, *i.e.*, C-Cubing(MM), C-Cubing(StarArray), C-Cubing(Star) and QC-DFS, for full closed cube computation (*i.e.*, $min\_sup = 1$). The performance of the four algorithms are compared with respect to tuple size (Fig. 3), dimension (Fig. 4), cardinality (Fig. 5), skewness (Fig. 6), and the weather data set (Fig. 7), where C-Cubing(MM)is denoted as $CC(MM)$, *etc.*

In the first experiment, we randomly generated data sets with 10 dimensions, varying the number of tuples from 200K to 1000K. In the second experiment, we varied the dimensions of the data from 6 to 10. The third experiment tests the performance while the cardinalities of each dimension are increased from 10 to 1000. We further tune the data skew from 0 to 3 in experiment 4. The tuple size for latter four datasets was 1000K. Finally, we test on weather data by selecting the first 5 to 8 dimensions.

The experimental results are shown in Figures 3–7. There are three main points that can be taken from these results. First, all the three new algorithms that use *aggregation-based closed checking* are consistently more efficient than QC-DFS, on both synthetic and

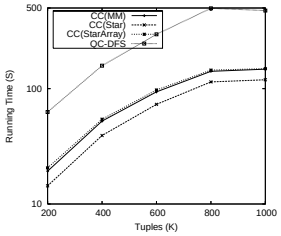

Figure 3: Closed Cube Computation w.r.t. Tuples, where $\mathcal{D} = 10,\ \mathcal{C} = 100,\ \mathcal{S} = 0,\ \mathcal{M} = 1$

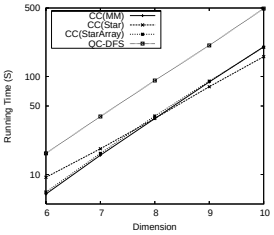

Figure 4: Closed Cube Computation w.r.t. Dimension, where $\mathcal{T} = 1000K,\ \mathcal{S} = 2,\ \mathcal{C} = 100,\ \mathcal{M} = 1$

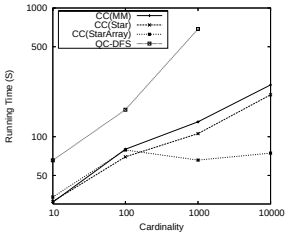

Figure 5: Closed Cube Computation w.r.t. Cardinality, where $\mathcal{T} = 1000K,\ \mathcal{D} = 8,\ \mathcal{S} = 1,\ \mathcal{M} = 1$

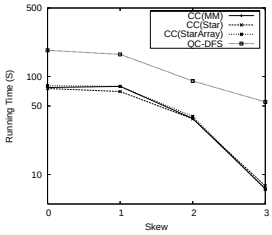

Figure 6: Closed Cube Computation w.r.t. Skew, where $\mathcal{T} = 1000K,\ \mathcal{C} = 100,\ \mathcal{D} = 8,\ \mathcal{M} = 1$

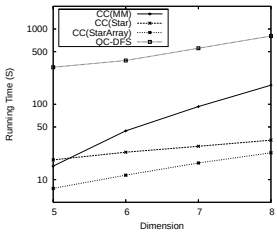

Figure 7: Closed Cube Computation w.r.t. Dimension, Weather Data Set, $\mathcal{M} = 1$

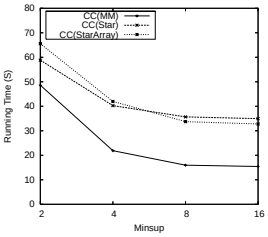

Figure 8: Closed Iceberg Cube Computation w.r.t. Minsup, where $\mathcal{T} = 1000K,\ \mathcal{C} = 100,\ \mathcal{S} = 0,\ \mathcal{D} = 8$

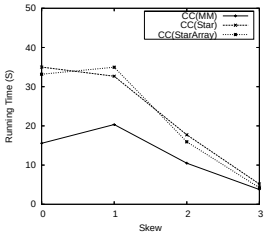

Figure 9: Closed Iceberg Cube Computation w.r.t. Skew, where $\mathcal{T} = 1000k,\ \mathcal{D} = 8,\ \mathcal{C} = 100,\ \mathcal{M} = 10$

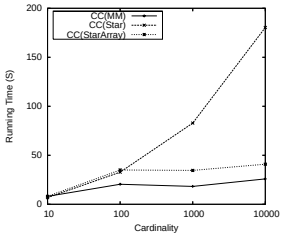

Figure 10: Closed Iceberg Cube Computation w.r.t. Cardinality, where $\mathcal{T} = 1000K,\ \mathcal{D} = 8,\ \mathcal{S} = 1,\ \mathcal{M} = 10$

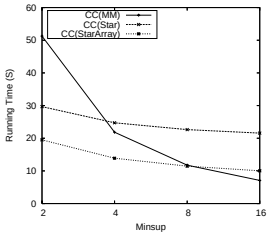

Figure 11: Closed Iceberg Cube Computation w.r.t. Minsup, Weather Data Set, $\mathcal{D} = 8$

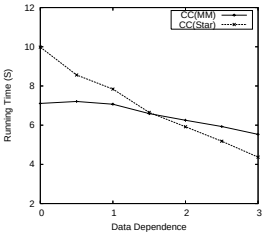

Figure 12: Cube Computation w.r.t. Data Dependence, where $\mathcal{T} = 400K, \mathcal{D} = 8,\ \mathcal{C} = 20,\ \mathcal{S} = 0,\ \mathcal{M} = 16$

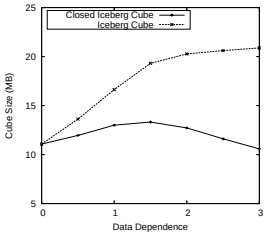

Figure 13: Cube Size w.r.t. Data Dependence, where $\mathcal{T} = 400k,\ \mathcal{D} = 8,\ \mathcal{C} = 20,\ \mathcal{S} = 0,\ \mathcal{M} = 16$

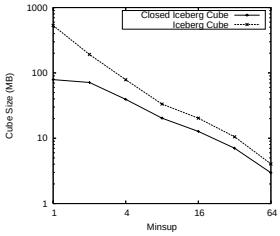

Figure 14: Cube Size w.r.t. Minsup, where $\mathcal{T} = 400k,\ \mathcal{D} = 8,\ \mathcal{C} = 20,\ \mathcal{S} = 0,\ \mathcal{R} = 2$

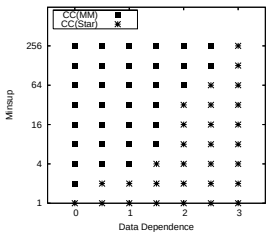

Figure 15: Best Algorithm, Varying Minsup and Dependence, where $\mathcal{T} = 400k$, $\mathcal{D} = 8$, $\mathcal{C} = 20$, $\mathcal{S} = 0$

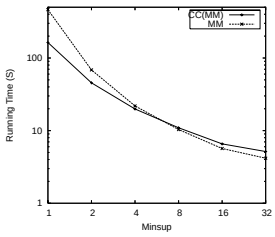

Figure 16: Overhead of Closed Checking (MMClose) w.r.t. Minsup, Weather Data Set, where $\mathcal{D} = 8$

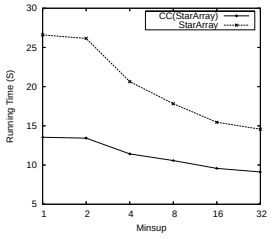

Figure 17: Benefits of Closed Pruning (StarArray) w.r.t. Minsup, Weather Data Set, Where $\mathcal{D} = 8$

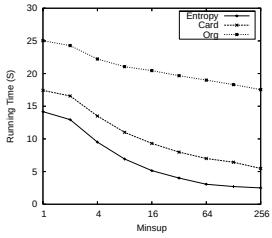

Figure 18: Cube Computation (StarArray) w.r.t. Dimension Order, where $\mathcal{T} = 400K, \mathcal{D} = 8$, $\mathcal{C} = 10, 1000$, $\mathcal{S} = 0, 1, 2, 3$

real datasets. C-Cubing(Star) and C-Cubing(StarArray) achieve better performance comparing with C-Cubing(MM) because the Star family algorithms exploit more closed pruning.

Second, when the cardinality is relatively low, C-Cubing(Star) performs better than C-Cubing(StarArray). However, when the cardinality increases, C-Cubing(StarArray) becomes better. This is consistent with our analysis in Section 4. In low cardinality, multiway aggregation in C-Cubing(Star) is promising, while in high cardinality, multiway traversal of C-Cubing(StarArray) is better. QC-DFS performs much worse in high cardinality because the counting sort costs more computation.

Third, all the algorithms get performance improvements as the skew increases. C-Cubing(MM) benefits from the nonuniform distribution where the dense and sparse spaces are easier to be identified. The two Star-Cubing-based algorithms, C-Cubing(Star) and C-Cubing(StarArray), are faster because the tree structure shares more computation in the dense parts of the data, and closed pruning is more effective in sparse parts.

5.2 Closed Iceberg Cubing

The second set of the experiments compare the performances of closed iceberg cubing (Fig. 8–11). We are not able to compare with QC-DFS in this section since QC-DFS did not implement iceberg cubing in their system. Moreover, the previous works [20, 14] already reported that, in iceberg cubing, MM-Cubing and Star-Cubing outperform BUC, from which QC-DFS is derived. The curves in the last subsection also clearly demonstrated our methods are more efficient while taking full closed cubing into consideration.

The performance is compared with respect to *min_sup*, data skew and cardinality in synthetic data. The data set used in Fig. 8 had 1M tuples with 8 dimensions and 0 skew. The *min_sup* is increased from 2 to 16. The parameter of data sets in Fig. 9 is the same as Fig. 8, except the skew is varied from 0 to 3. The cardinalities are increased from 10 to 10000 in Fig. 10.

We can see that C-Cubing(MM) performs better when *min_sup* increases. This is because the iceberg pruning gets more effective. The factor of closed pruning is decreasing since some

of the closed cells are ruled out by iceberg condition. Fig. 11 uses the same weather data with 8 dimensions. It is observed that the switching point between C-Cubing(MM) and the Star family algorithms in this experiment is higher than those in the synthetic datasets.

Comparing with the results in the last subsection, we have seen a totally different behavior of the algorithms. Generally speaking, Star family algorithms work well while *min_sup* is low. They are outperformed by C-Cubing(MM) as the *min_sup* increases. We are interested in where are the switching points and how the properties of the data set affect the switching points. We conducted another group of experiments for this purpose.

5.3 Closed Pruning vs. Iceberg Pruning

In this subsection, we first generalize the observations in the last two sections, which leads to an important data set property: data dependence. We then show how the data dependence will affect the performance of the algorithms.

Closed cells imply the dependence of the values on different dimensions. For example, if cell $cell_1 = (a_1, b_1, c_1)$ is closed and it covers cell $cell_2 = (a_1, b_1, *)$, then there exists a dependence rule: $(a_1, b_1) \rightarrow c_1$. Most data sets in our experiments are very sparse, that is, the feature space size (by multiplying all the cardinalities) is much larger than the number of tuples. Sparse data sets contain lots of trivial closed cells whose support is 1. Trivial closed cells play an important role when *min_sup* is 1, however, when *min_sup* increases, these trivial closed cells are already pruned by iceberg conditions. Generally, the closed pruning is more effective when the dependence in the data is higher, since more closed cells will "survive" from the iceberg pruning.

To verify this hypothesis, we incorporate data dependence into our synthetic data generation. The data dependence is modeled as *rules*. An example of rule is as discussed above: $(a_1, b_1) \rightarrow c_1$, which means that if a_1 appears on dimension A, and b_1 appears on dimension B, then the value on dimension C will be fixed to c_1. This kind of rules exists widely in real database. For example, in weather data, when a certain weather condition (a_1) appears at the same time of the day (b_1), there is always a unique value for solar altitude (c_1).

Each rule corresponds to a *pruning power*, which means how many cells will be pruned by this rule. Assume the original cube size is S, the portion which contains a_1 and b_1 has an estimated size $\frac{S}{Card(A) \times Card(B)}$. Further dividing the portion by the value on dimension C results in $Card(C) + 1$ classes, which have the form of $(a_1, b_1, *, \ldots)$, $(a_1, b_1, c_1, \ldots)$, $\ldots$, $(a_1, b_1, c_{Card(C)}, \ldots)$. After applying the rule $(a_1, b_1) \rightarrow c_1$, only cells $(a_1, b_1, c_1, \ldots)$ left. Thus the pruning power of this rule is estimated by $\frac{Card(C)}{Card(A) \times Card(B) \times (Card(C)+1)}$. The pruning power of a set of rules is evaluated by accumulating their individual pruning power. And the data dependence R is measured as $R = -\sum_{i=1}^{n} \log(1 - pruning\ power(rule_i))$. The larger the value of R is, the more dependent is the dataset.

To compare the closed pruning and iceberg pruning, we run a set of experiments by varying data dependence R and *min_sup*. Fig. 12 shows the performances of C-Cubing(MM) and C-Cubing(Star) with respect to the value of R. The data has 400K tuples with 8 dimensions, each having cardinality 20, and *min_sup* is 8. Fig. 13 is the corresponding sizes of iceberg

cubes and closed iceberg cubes. The larger the difference of iceberg cubes and closed iceberg cubes, the more significant the closed pruning. To compare the closed pruning and iceberg pruning with respect to *min_sup*, we fixed $R = 2$, varying *min_sup* from 1 to 64, the size of the cubes are shown in Fig. 14. In this case, we see that the iceberg pruning becomes the dominating factor when *min_sup* is high.

Fig. 15 shows a more comprehensive study on the algorithm performance w.r.t. R and *min_sup*, while the other parameters of the data are kept the same as above. The value of R is varied from 1 to 3, while the value of *min_sup* is increased from 1 to 512. For a given parameter set, the ID of the best algorithm is plotted. Since the test data sets have relatively low cardinality, C-Cubing(Star) performs better than C-Cubing(StarArray). We can expect that if the cardinality is large, C-Cubing(StarArray) will be better.

It is clear that the performance of C-Cubing(MM) and the Star family are correlated to the extent that closed pruning can be achieved with respect to the iceberg pruning . The Star family algorithms are promising when the closed pruning is comparatively large, and C-Cubing(MM) is better when the iceberg pruning dominates.

5.4 Overhead of Closed Checking

In this section, we examine the overhead of our closed checking method. There are additional costs to aggregate the closedness measure, but we will show that it is not a major expense in the context of computing the closed iceberg cube. Furthermore, the closed pruning in C-Cubing(Star) and C-Cubing(StarArray) actually reduces the overall computational cost.

We compare C-Cubing(MM) and C-Cubing(StarArray) with MM-Cubing and the non-closed version of StarArray, respectively. The output was disabled, so the difference on I/O is eliminated. We tested all the algorithms on the weather data. Fig. 16 shows the comparison on C-Cubing(MM) and MM-Cubing. To our surprise, C-Cubing(MM) performs better than MM-Cubing in low *min_sup*. This is because a simple optimization is exploited in C-Cubing(MM): When a subspace whose size is equal to the *min_sup* is found, C-Cubing(MM) will directly output the closed cell, while MM-Cubing will enumerate all the combinations (though they are not output). When the *min_sup* is high, MM-Cubing performs better, but the overhead of C-Cubing(MM) is within 10% of the total time.

Fig. 17 shows the performance of C-Cubing(StarArray), the closed version runs faster than the non-closed version, especially when *min_sup* is low, since at that time the closed pruning is more effective.

5.5 Dimension Ordering

Since C-Cubing(MM) does not follow a specific dimension order, it is not sensitive to dimension ordering. However, for C-Cubing(Star) and C-Cubing(StarArray), they obey the dimension order throughout the computation, it is worth to exploit a heuristic for efficient iceberg pruning.

The well-known strategy is to order the dimensions in cardinality descending order. We propose another ordering strategy based on data distribution. Think about 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:

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

where $|a_i|$ is the number of tuples whose value on dimension A is a_i. 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, the larger the entropy value, thus we order the dimension in measure E descending order. Fig. 18 shows the performance difference on a data set with 8 dimensions: four of them have cardinalities 10 with different skews $(0, 1, 2, 3)$, the other four have cardinalities 1000 with different skews $(0, 1, 2, 3)$. We vary *min_sup* from 10 to 1000. *Org* is the original dimension order, *Card* is cardinality descending order, and *Entropy* represents the order we discussed above. The results show that ordering by entropy gets considerable improvement.

In summary, we have systematically tested three closed iceberg cubing algorithms: C-Cubing(MM), C-Cubing(Star), and C-Cubing(StarArray), with the variations of cardinality, skew, *min_sup*, and data dependence. The Star family algorithms perform better when *min_sup* is low. C-Cubing(MM) is good when *min_sup* is high. The switching point of *min_sup* increases with the dependence in the data: High dependence incurs more closed pruning, thus it benefits the Star algorithms. Comparing C-Cubing(Star) and C-Cubing(StarArray), the former is better if the cardinality is low; otherwise, C-Cubing(StarArray) is better.

6 Discussions

In this section, we will discuss a few issues related to closed cubing and point out some research directions.

6.1 Closed Cube with Complex Measures

Throughout the paper, we have used *count* as the measure. For iceberg cubing, complex measures such as *average* can be easily incorporated into our algorithm, based on the technique proposed in [8].

For closed cubing, the closed pruning and checking method on *count* can play the basic role in the computation. Lemma 1 implies that closed cube on measure *count* is the baseline of any other measures. For example, given any complex measure, we attach *count* as the auxiliary measure, we then check closedness with respect to *count* instead of the given

measure. The result cube will not miss any closed cells (though may not be optimal). If the measure is strictly monotonic or anti-monotonic, the closed pruning and checking by measure *count* is complete.

6.2 Mining Closed Rules

Theoretically, the closed cube losslessly compresses the original cube. However, to build an efficient index structure of the cube (*i.e.*, QC-Tree [10]), additional information may be required. Instead of generating lower bounds for each closed cell as in [10], we recommend to compute the closed rules. A closed rule has the form of $a_{c1}, a_{c2}, \ldots, a_{ci} \rightarrow a_{t1}, a_{t2}, \ldots a_{tj}$, where a_{ck} ($k \in \{1, 2, \ldots, i\}$) and a_{tk} ($k \in \{1, 2, \ldots, j\}$) are non-$*$ values, $c1, c2, \ldots, ci$ are the condition dimensions, and $t1$, $t2$, $\ldots$, tj are the target dimensions. A closed rule means that if a cell having value a_{ck} on dimension ck ($k \in \{1, 2, \ldots, i\}$), then on dimension tk ($k \in \{1, 2, \ldots, j\}$), it must have value a_{tk}.

The benefit of rule generations is that it yields a more compact result comparing with the lower-bound approach, since there are many lower-bound and upper-bound pairs sharing the same closed rule. For example, in the 8-dimension weather data, we run the experiment with minsup 10. While there are 462k closed cells, we can get a number of 57k closed rules, which is less than 15% of the cube size.

6.3 Handling Large Databases

All the data sets used in our experiments can fit in main memory. One may wonder what may happen if the dataset cannot fit in memory. The issue of handling large database with Star-Cubing was discussed in [20]. We can follow the same approach in C-Cubing(Star) since the closedness checking does not introduce additional constraints. The idea is to first scan and partition the data set into separate smaller data files based on the values in one dimension, and then compute the partitions one by one. When it finishes computing the first partition, the released memory will be used to compute the next partition, and so on.

7 Conclusions

For efficient computation of closed (iceberg) cubes, we have proposed an aggregation-based closedness checking approach, C-Cubing. With this approach, we proposed and implemented three algorithms: C-Cubing(MM), C-Cubing(Star) and C-Cubing(StarArray). All the three algorithms outperform the previous approach. Among them, we have found C-Cubing(MM) is good when iceberg pruning dominates the computation, whereas the Star family algorithms perform better when closed pruning is significant.

References

[1] E. Baralis et al. Materialized View Selection in a Multidimensional Database. VLDB'97.

[2] D. Barbara et al. Quasi-cubes: Exploiting approximation in multidimensional database. SIGMOD Record, 26, 1997.

[3] K. Beyer and R. Ramakrishnan. Bottom-Up Computation of Sparse and Iceberg Cubes. SIGMOD'99.

[4] Y. Chen et al. Multi-Dimensional Regression Analysis of Time-Series Data Streams. VLDB'02.

[5] J. Gray et al. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. Data Mining and Knowledge Discovery, 1, 1997.

[6] H. Gupta et al. Index Selection of OLAP. ICDE'97.

[7] C.J. Hahn et al. 1994: Edited Synoptic Cloud Reports from Ships and Land Stations Over the Globe, 1982-1991. (http://cdiac.ornl.gov/ftp/ndp026b/)

[8] J. Han et al. Efficient Computation of Iceberg Cubes with Complex Measures. SIGMOD'01.

[9] L. Lakshmanan et al. Quotient Cubes: How to Summarize the Semantics of a Data Cube. VLDB'02.

[10] L. Lakshmanan et al. QC-Trees: An Efficient Summary Structure for Semantic OLAP. SIGMOD'03.

[11] X. Li et al. High-Dimensional OLAP: A Minimal Cubing Approach. VLDB'04.

[12] R. Ng et al. Exploratory Mining and Pruning Optimizations of Constrained Associations Rules, SIGMOD'98.

[13] J. Shanmugasundaram et al. Compressed Data Cubes for OLAP Aggregate Query Approximation on Continuous Dimension. KDD'99.

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

[15] A. Shukla et al. Materialized View Selection for Multidimensional Datasets. VLDB'99.

[16] Y. Sismanis et al. Dwarf: Shrinking the PetaCube. SIGMOD'02.

[17] J. Vitter et al. Data Cube approximation and Histograms via Wavelets. CIKM'98.

[18] W. Wang et al. Condensed Cube: An Effective Approach to Reducing Data Cube Size. ICDE'02.

[19] J. Wang et al. Closet+: Searching for the Best Strategies for Mining Frequent Closed Itemsets. KDD'03.

[20] D. Xin et al. Star-Cubing: Computing Iceberg Cubes by Top-Down and Bottom-Up Integration. VLDB'03.

[21] M. Zaki and C. Hsiao. Charm: An Effcient Algorithm for Closed Association Rule Mining. SDM'02.

[22] Y. Zhao et al. An Array-Based Algorithm for Simultaneous Multidimensional Aggregates. SIGMOD'97.