# MM-Cubing: Computing Iceberg Cubes by Factorizing the Lattice Space[*]

Zheng Shao Jiawei Han Dong Xin
University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
zshao1@uiuc.edu hanj@cs.uiuc.edu dongxin@uiuc.edu

## Abstract

*The data cube and iceberg cube computation problem has been studied by many researchers. There are three major approaches developed in this direction: (1) top-down computation, represented by* MultiWay *array aggregation [22], which utilizes shared computation and performs well on dense data sets; (2) bottom-up computation, represented by* BUC *[6], which takes advantage of* Apriori *Pruning and performs well on sparse data sets; and (3) integrated top-down and bottom-up computation, represented by* Star-Cubing *[21], which takes advantages of both and has high performance in most cases. However, the performance of* Star-Cubing *degrades in very sparse data sets due to the additional cost introduced by the tree structure. None of the three approaches achieves* uniformly *high performance on all kinds of data sets.*

*In this paper, we present a new approach that compute Iceberg Cubes by factorizing the lattice space according to the frequency of values. This approach, different from all the previous dimension-based approaches where the importance of data distribution is not recognized, partitions the cube lattice into one dense subspace and several sparse subspaces. With this approach, a new method called* MM-Cubing *has been developed.* MM-Cubing *is highly adaptive to dense, sparse or skewed data sets. Our performance study shows that* MM-Cubing *is efficient and achieves high performance over all kinds of data distributions.*

## 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[8]. The previous studies on data cubing mainly fall into 5 categories: (1) computation of full or iceberg cubes with simple or complex measures [1, 22, 15, 6, 9], (2) approximate computation of compressed data cubes, such as quasi-cubes, wavelet cubes, etc. [4, 19, 16, 5], (3) closed cube computation with index structure, such as condensed, dwarf, or quotient cubes [13, 20, 18], (4) selective materialization of views [11, 3, 17], and (5) cubes computation in stream data for multi-dimensional regression analysis [7].

Among these categories, the first one, *efficient computation of full or iceberg cubes*, is a fundamental problem to the other categories. Any new method developed here may strongly influence the developments in the other categories.

Here is the definition of cube computation problem: Given a base relation $R$ and $n$ attributes, a cell $a = (a_1, a_2, \ldots, a_n, c)$ in an $n$-dimension data cube is a "GROUP BY" with measure $c$ and $n$ distinct attribute values $a_1, a_2, \ldots, a_n$. A cell $a$ is called an *$m$-dimensional cell*, if and only if there are exactly $m$ ($m \leq n$) values among $\{a_1, a_2, \ldots, a_n\}$ which are *not* $*$ ($*$ is a wildcard for all values). It is called a *base cell* if $m = n$. Given a base relation, our task is to compute an *iceberg cube*, i.e., the set of cells which satisfies an iceberg condition, or the *full cube* if there is no such condition. We study the case that the measure $c$ is the *count* of base cells, and *min_sup* (threshold) is the iceberg condition.

We observed that any dataset could be classified into one of the following categories: dense, sparse and skewed (where dense and sparse portions coexist in the same dataset). Previous studies on cube computation have developed three major approaches, each of which may work well on some kinds of dataset, but not all. The first approach, represented by MultiWay [22], utilizes shared computation on dense datasets. However, this approach is limited by the huge memory requirement and cannot be applied to sparse and skewed ones. The second approach, represented by BUC [6], facilitates Apriori pruning and has shown high performance on sparse datasets. However, this approach does not work well on dense and skewed datasets. The third approach, represented by Star-Cubing [21], tries to combine the advantages of the above two, and has shown high performance on skewed, dense, and not-so-sparse datasets. However, it suffers from the additional cost introduced by the tree structure in very sparse datasets. The performance

[*]The work was supported in part by National Science Foundation under grants No. IIS-02-09199/IIS-03-08215. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies.

vs. dataset property is summarized in Table 1.

| Algorithm | Dense | Sparse | Skewed |
|---|---|---|---|
| MultiWay | High | Low | Low |
| BUC | Low | High | Low |
| Star-Cubing | High | Medium | High |
| MM-Cubing | High | High | High |

**Table 1. Algorithm Performance vs. Dataset Property**

*Can we develop an algorithm which works uniformly well on all kinds of data sets?* In this paper, a new iceberg cubing method, MM-Cubing, is proposed. While all the previous algorithms are dimension-based approaches (the computation order follows dimension expansion or contraction in the lattice space), ignoring the diversity of data density in the same dimension (and further in the whole data set), MM-Cubing is a density-based method, where *MM* represents *major* and *minor* values. By distinguishing those values, the lattice space is factorized into one dense subspace and several sparse subspaces, and MM-Cubing applies specialized methods for each subspace. Our performance study shows that MM-Cubing is highly efficient and performs uniformly well in all kinds of data distributions.

The remaining of the paper is organized as follows. In Section 2, the three major algorithms in cube computation are re-examined. In Section 3, we introduce the concept of *factorization of the Lattice Space*. A new algorithm, MM-Cubing, based on such factorization, is developed in Section 4. Our performance study is presented in Section 5. Discussions on the potential extensions and limitations of the MM-Cubing algorithm is in Section 6, and we conclude our study in Section 7.

## 2. Overview of Existing Cubing Algorithms

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

### 2.1 MultiWay array aggregation

MultiWay array aggregation [22] is an array-based cubing algorithm. It uses a *compressed sparse array* structure to load the base cuboid and compute the cube. To save memory usage, the base cuboid is partitioned into *chunks*. It is unnecessary to keep all the chunks in memory since only parts of the group-by arrays are needed at any time. By carefully arranging the chunk computation order, multiple cuboids can be computed simultaneously in one pass.

The MultiWay algorithm is effective when the 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.

### 2.2 BUC

BUC [6] 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 as MultiWay does, which is very useful in computing dense datasets. Furthermore, BUC is very sensitive to the skew of the data. The performance of BUC degrades when the skew of data increases.

### 2.3 Star-Cubing

Star-Cubing [21] uses a hyper-tree structure, called Star-Tree, which is an extension of H-tree [9], to facilitate cube computation. Each level in the tree represents a dimension in the base cuboid. The algorithm takes the advantages of the above two models. On the global computation order, it uses the simultaneous aggregation similar to MultiWay. However, it has a sub-layer underneath based on the bottom-up model by exploring the notion of shared dimension. This integration allows the algorithm to aggregate on multiple dimensions, while it can still partition parent group-by's and use Apriori-based pruning on child group-by's.

Star-Cubing also introduced a special value "Star" for attributes whose count is less then *min_sup* (This is different from "*" in this paper, which is a wildcard), which has been proved useful for skewed data. It performs well on dense, skewed and not-so-sparse data. However, in very sparse data sets, it has to handle the additional traversal cost introduced by the tree structure.

## 3. Factorization of the Lattice Space

Mathematically, each (aggregation) cell is in the form of $(a, b, c, d, \ldots)$, where $a, b, c$, and $d$ is either a value in the corresponding dimension, or "*", representing all the values. These cells form the lattice space. The cardinality of each dimension in the lattice space is one (the "*" value) plus the cardinality of the corresponding dimension of the input tuples.

As a notational convention, the dimension of the input tuples is denoted by $D$, the cardinality of the $i_{th}$ dimension by $C_i$, and the number of input tuples by $T$.

Each tuple contributes to $2^D$ (base and aggregate) cells, however, there are altogether $\prod_{i=1}^{D}(C_i + 1)$ cells. Virtually, one can construct a huge *naive table*, with each tuple as a row, and each cell as a column. If the tuple contributes to the cell, the intersection of the row and the column is $1$, otherwise it is $0$. Then the sum of all the elements in a row is $2^D$, and the sum of all the elements in a column is the support of the cell. If $C_i$ or $D$ is big, the *naive table* will be very sparse. What is more important is that there are a lot of columns summing up to a support less than *min_sup*, and the computation of these columns are useless. This is one of the main reasons why it is hard to derive an efficient Iceberg Cubing algorithm.

The former three approaches try to solve this problem by a lattice tree. They first use a grid structure to represent the lattice space. Each non-$*$ value in one dimension is considered equally and merged to a single point, so the grid structure has only $2^D$ nodes, as shown in Figure 1 (1). Then they treat these $2^D$ nodes as vertices in a graph, add edges between adjacent nodes and compute them via a tree which is a subgraph of the graph. The tree structure can explore shared aggregation and Apriori pruning, which both can accelerate the computation. A possible tree based on the lattice structure is shown in Figure 1 (2).

Whatever the direction of computation (e.g., top-down, bottom-up, or both) that a previous algorithm will use, it overlooks the value density of the data. That is, for all the non-$*$ values in one dimension, there are possibly some values with a fairly high frequency, and some with a low frequency; but the grid structure sees no difference, and cells with different values are compressed into a single grid point. The computations of all three approaches are based on a fixed structure. Although they use different structures and some are more adaptive to the heterogeneity among different dimensions of data, the values in one dimension with different frequencies are always treated the same, since they are represented by a single cell in the grid structure.

In order to perform highly efficient and highly adaptive Iceberg Cubing, it is necessary to have different treatments of frequent values from infrequent ones, because frequent values tend to contribute to *frequent cells*. Let us denote frequent values as *major values*, and infrequent ones as *minor values*, and leave the problem of how frequent a *major value* should be to the next section.

Unlike the former grid structure, the factorization of the lattice space differentiates major values from minor ones, and these values are represented by separate points. Thus, the new grid structure has $3^D$ nodes. If we denote a major value by $M$, a minor value by $I$, and ALL by $*$, a 3-D lattice space can be written as:

$$(\{M, I, *\}, \{M, I, *\}, \{M, I, *\})$$

Note that a minor value should also occur at least *min_sup* times. A value that occurs less than *min_sup* times will never occur in an iceberg cell based on its definition.

Since a major value has a higher frequency, it is desirable to share its computation with *; but for a minor value, its further computation is likely to be pruned sooner. Thus, we factorize the lattice space into the following subspaces:

1. $(\quad \{M,*\}, \quad \{M,*\}, \quad \{M,*\} \quad)$
2. $(\quad I, \quad \{M,I,*\}, \quad \{M,I,*\} \quad)$
3. $(\quad \{M,*\}, \quad I, \quad \{M,I,*\} \quad)$
4. $(\quad \{M,*\}, \quad \{M,*\}, \quad I \quad)$

Note that these subspaces do not intersect, and the sum of these subspaces is just the original lattice space. (Add $4$ to 1 first, then add 3 and 2 in order.) See Figure 1 (3) for an illustration.

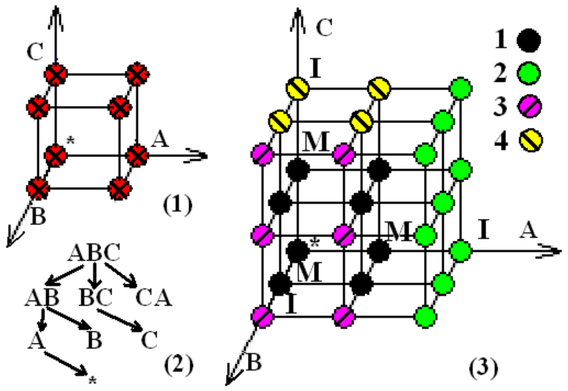

**Figure 1. Factorization of the Lattice Space**
(1) Traditional Lattice Space (2) Traditional Lattice Tree (3) Factorization of the New Lattice Space

1. $(\{M, *\}, \{M, *\}, \{M, *\})$. The value of each dimension in the first subspace is a major value or $*$. The cells in this subspace probably have a large support. This subspace is relatively dense, and we call it the *dense subspace*.
2. $(I, \{M, I, *\}, \{M, I, *\})$. All the values in the first dimension of the cells in the second subspace are minor values. The support of the cells will not be very large, since the frequency of a minor value is not very high.
3. $(\{M, *\}, I, \{M, I, *\})$. The third subspace is similar to the second one, and the support of the cells will not be very large. But there is a difference: the value on the first dimension can only be a major value or $*$. This is a strange requirement. However, without this requirement, this subspace would intersect with the second one.
4. $(\{M, *\}, \{M, *\}, I)$. This subspace is similar to the third one except that there are two dimensions in which a value must be a major value or $*$.

In contrast to the first subspace (the *dense subspace*), every other subspace is called a *sparse subspace*.

Note that although there is only one dense subspace in a factorization, which does not conform to real databases, recursive calls in the sparse subspaces will explore other dense subspaces inside them.

There are also some other factorization methods. For example, the following is also a valid factorization.

| | | | |
|---|---|---|---|
| 1. | ( $\{M,*\}$, | $\{M,*\}$, | $\{M,*\}$ ) |
| 2. | ( $I$, | $\{M,*\}$, | $\{M,I,*\}$ ) |
| 3. | ( $\{M,I,*\}$, | $I$, | $\{M,*\}$ ) |
| 4. | ( $\{M,*\}$, | $\{M,I,*\}$, | $I$ ) |
| 5. | ( $I$, | $I$, | $I$ ) |

To choose which factorization method depends on the algorithm. Generally, the first factorization is simple and can meet the need of efficient Iceberg Cube algorithms. However, the order of the dimension can be changed to optimize the speed. This will be discussed in Section 5.

## 4. MM-Cubing Algorithm

This section introduces the MM-Cubing algorithm, which first factorizes the lattice space by the method introduced above, and then computes (1) the dense subspace by simultaneous aggregation, and (2) the sparse subspaces by recursive calls to itself.

Before factorization, we first count the frequency of the values in each dimension and sort them in descending order, and then determine which values are major or minor ones. This step is crucial to the efficiency of the algorithm, however, there is no optimal way for this task since this requires knowledge about the correlation among multiple dimensions, which is too expensive to explore. We will present several heuristics here. Once the major and minor values are determined, the factorization step is done.

The dense subspace is computed by simultaneous aggregation. Unlike the simultaneous aggregation in MultiWay which computes the entire lattice space, we only consider the major and $*$ values. We will present our efficient method for computing this space.

The other subspaces are solved by recursive calls. However, the strange requirement mentioned in the last section makes the recursive calls more complex. The details of the recursive calls will also be shown in this section.

Notice that in some cases, the data set is uniformly dense, then all values are major values and no recursive calls will be made. On the other hand, in some other cases, the data set is uniformly sparse, then all values are minor values and the dense subspace will be a single cell $(*, *, \ldots, *)$.

**Algorithm 1 (MM-Cubing).** Compute iceberg cubes.

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

**Output**: The computed iceberg cube.

**Method**: The algorithm is described in Figure 2.

```
BEGIN
1.  for each dimension $i$
2.      count and sort the values in dimension $i$
3.  determine $major$ and $minor$ values in each dimension
4.  do simultaneous aggregation in the $dense\ subspace$
5.  for each $sparse\ subspace$
6.      for each $minor$ value of that subspace {
7.          recursive call
8.          replace the $minor$ value by the non-aggregatable value
9.      }
10. for each replaced value
11.     restore the non-aggregatable value by $minor$ values
END
```

**Figure 2. The MM-Cubing algorithm**

### 4.1. Count and Sort

We introduce the Count-and-Sort algorithm and related data structures, which computes the frequency of the values and provides a data structure that facilitates the major value selection and the computation of the subspaces.

Assume we have the following input data. The value row is the values of a specific dimension in all the tuples.

INPUT: (*min_sup* = 2)

| Tuple ID | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| Value | 1 | 2 | 1 | 1 | 2 | 4 | 3 | 2 | 2 |

OUTPUT:

| Count | Value | Tuples |
|---|---|---|
| 4 | 2 | 2, 5, 8, 9 |
| 3 | 1 | 1, 3, 4 |

Note that values that appeared less than *min_sup* times are pruned, and the resulting output is sorted in descending order of *Count*. We do this in two steps: (1) gather the same value together, and (2) sort the values according to *Count*. Step 1 can be done by the bucket-sort algorithm in linear time, provided that the cardinality is smaller than the number of tuples. For Step 2, we just use quick sort.

The data structure for this step is a key factor to the performance. The bucket-sort algorithm is implemented with two arrays, one with the size of cardinality and the other one of the same size as the input list, rather than implemented with pointers. The data structure for output can be an array of struct and an array of tuple IDs. We call this structure a *shared array*, as shown in Figure 3.

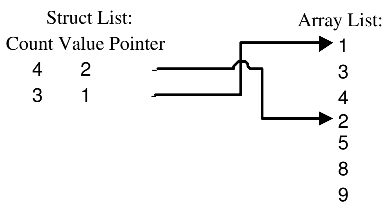

**Figure 3. The *Shared Array* Structure**

## 4.2. Determining Major Values

The optimal major value assignment should minimize the running time. However, the algorithm is a recursive one, and it is difficult to estimate the running time of the recursive calls, since the time depends on the major value assignment in those subspaces. Nevertheless, there is an obvious rule for determining major values: *All the major values in one dimension should appear more frequently than all the minor values in the same dimension*. That is why we sort the values according to their frequencies.

Here we use heuristics to help determine major values. We first define a quantity: *work_done* as the total amount of work done by the simultaneous aggregation:

$$\textit{work\_done} = T \times \prod_{i=1}^{D}(1 + sum[i]/T),$$

where $T$ is the number of tuples, $sum[i]$ is the number of tuples that have a major value in dimension $i$ and $D$ is the number of dimensions. The simultaneous aggregation table size can be calculated as: $\prod_i(1 + MC[i])$, where $MC[i]$ is the number of major values in the $i_{th}$ dimension.

We want to have a major-value assignment in which *work_done* is maximized while aggregation table size is no larger than a constant limit.

The definition of *work_done* is derived from a naïve algorithm. In this naïve algorithm, we keep a huge hash table storing all the aggregation cells. The key of the hash table is the aggregation cell itself $(a, b, \ldots)$ and the value is the support of this cell. This naïve algorithm will need $T \times 2^D$ steps to finish. Now we try to put as much *work_done* as possible in the dense subspace, since simultaneous aggregation will be used and it is efficient when data is dense. The difference of $T \times 2^D$ and *work_done* will be processed by recursive calls.

A greedy strategy is applied to determine major values. At first, all non-$*$ values are minor values. We repeatedly choose the most *promising* minor value from the most frequent value in each dimension and let it be a major value, until the predefined table size is reached.

Setting one value in dimension $i$ to be a major value can increase the value of $MC[i]$ by one, and cause the size of the aggregation table to increase by an additive factor of $1/(MC[i] + 1)$. After setting the value to be a major value, we need to add $oc[i]$ to $sum[i]$, where $oc[i]$ is the number of occurrences of the most frequent minor value in dimension $i$. The additive ratio of *work_done* increase is $oc[i]/(1 + sum[i]/T)$.

For the most frequent value in each dimension, we define a *weight* as the ratio of work increase divided by the ratio of space increase. The value with the highest weight is the most *promising* value which we will choose to be a major value.

Thanks to the *Shared Array* structure, it is easy to find out the most frequent value in each dimension. If there are multiple most frequent values in one dimension, we will only consider the one that comes first in the *Shared Array*. Others will be considers after the first one becomes a major value. The complexity for determining major values is $D \times \sum_{i=1}^{D}(MC[i])$. It can be further improved to $log(D) \times \sum_{i=1}^{D}(MC[i])$ if we apply a heap structure to store the candidates from all dimensions. $\prod_i(1+MC[i])$ is bounded by the aggregation table size, so $sum_{i=1}^{D}(MC[i])$ won't be large, especially in high-dimensional cases.

Here we use fixed aggregation table size and use the *work_done* as the target function, then we use greedy approach to solve the problem. However, there are many other ways to determine major values. It seems more appropriate to allow dynamic aggregation table size, and there might be better target function than the *work_done*. We will discuss such alternatives in Section 5.

## 4.3. Simultaneous Aggregation in Dense subspace

By now, the factorization of the lattice space is done. Now we will do aggregation in the dense subspace.

The aggregation here is similar to that in MultiWay, with two differences: (1) MultiWay considers a lot about memory usage and uses chunks to alleviate the problem, while MM-Cubing doesn't trade efficiency for memory space, since the dense subspace is small (predefined table size). (2) The aggregation in MultiWay is simpler than that in MM-Cubing. In MultiWay, values in the input tuples can directly be used as subscripts in the aggregation array. But in MM-Cubing, major values are generally not contiguous and a mapping from major values to the range of 1 and $MC$ (major value count) is needed.

All the aggregation values are put in a $D$-dimensional array, $A[b_1][b_2]\cdots[b_D]$, where $b_i$ is in the range of 0 to $MC[i]$. The subscript 0 corresponds to the $*$-value.

The simultaneous aggregation involves two steps: (1) find the base cell for each tuple, and (2) do the aggregation.

Each tuple corresponds to one base cell. It is hard to know which base cell that a tuple corresponds to, since there is no value to major-value-id table. However, we can calculate the corresponding base cells for all tuples at the same time efficiently with the help of the *shared array*.

We have a subscript array $B[tuple][dimension]$ to store the correspondent subscript in array $A$ for a tuple. We just need to go through the *shared array* once to get the subscript array. Once we get the subscript arrays $B$, we know which base cell a tuple corresponds to.

It is a crucial problem of how to treat non-major values. If we assign a new major-value-id to all non-major values, the aggregation table size will be $2^D$ even when there is no major value. This makes MM-Cubing impossible for $D > 30$. However, if we let non-major value share space with $*$-value, the main memory will be greatly saved. This approach needs a second array $C$ of the same size as $A$.

All the tuples contribute to the corresponding base cell of array $C$. Note that subscript 0 in array $C$ denotes all non-major values, but in array $A$ it denotes the $*$-value. So we have $A[b_1][b_2][0] = \sum_{j=0}^{MC[3]}(C[b_1][b_2][j])$ $(b_1, b_2 > 0)$.

The aggregation is a little tricky. For example, the value of $A[0][b_2][0]$ is $\sum_{i=0}^{MC[1]}(\sum_{j=0}^{MC[3]}(C[i][b_2][j]))$, or $\sum_{j=0}^{MC[3]}(C[0][b_2][j])+\sum_{i=1}^{MC[1]}(A[i][b_2][0])$ using dynamic programming. Obviously it is more efficient by using dynamic programming. Note that this approach actually has the same efficiency as assigning a new major-value-id to all non-major values, while saving a lot of memory.

Now the support of all the cells in the dense subspace is stored in array $A$, and all the cells with a support no less than *min_sup* can be output.

For the sake of memory, it is possible to use a single array for $A$ and $C$ by carefully arranging the order of aggregation. For the sake of efficiency, it is better to transform $A$ into a 1-d array in implementation, because we will then need only one subscript per tuple. In this way, array $B$ can be 1-d, too. This in turns saves a lot of memory.

### 4.4. Recursive Calls for Sparse subspaces

The sparse subspaces are computed by recursive calls. Consider the sparse subspace, $(I, \{M, I, *\}, \{M, I, *\})$. Decomposing the minor value, we get

$$\begin{array}{lcccl} & ( & I, & \{M,I,*\}, & \{M,I,*\} & ) \\ = & ( & i_1, & \{M,I,*\}, & \{M,I,*\} & ) \\ + & ( & i_2, & \{M,I,*\}, & \{M,I,*\} & ) \\ + & \cdots \end{array}$$

where the first minor value is denoted by $i_1$, the second by $i_2$, and so on.

For each of the subspaces on the right side of the equation, the value on the first dimension is a single value. So we can split all the tuples according to the value of the first dimension, and for each minor value in the first dimension, we recursively call the MM-Cubing algorithm on all the tuples with that minor value.

In the Count and Sort step, we have already gathered all the tuples with the same value in a specific dimension. We only need to pass the tuple IDs, instead of the tuples themselves, to the recursive calls. In this way, memory is saved and speed is improved.

All other sparse subspaces are computed almost the same as the first one, with one subtle difference. For example, the subspace $(\{M, *\}, I, \{M, I, *\})$ requires the aggregation value in the first dimension to be a major value or $*$, since $(I, I, \{M, I, *\})$ is already computed in the first sparse space. The solution to this problem is to set all the minor values in the first dimension to a special *non-aggregatable value*. After the computation of this subspace, the minor values are restored. Setting and restoring from the *non-aggregatable value* can be easily implemented by the *shared array* structure.

Note that BUC also use recursive calls. However, the recursive calls here are shallower and more efficient, since they are only on minor values and the number of tuples involved will shrink drastically. When the number of tuples is less than the *min_sup*, it can be pruned. In contrast, BUC does recursive calls on all values. If almost all tuples have the same value in one dimension, the recursive call of that value will involve almost all the tuples. Thus, the recursive calls in MM-Cubing are much shallower and provide a better performance.

## 5. Algorithm Improvements

So far, we have presented the basic MM-Cubing algorithm, *MM-Cubing V1.0*. Its efficiency is based on (1) calculating the dense subspace by simultaneous aggregation, which is efficient; and (2) calculating the sparse subspaces by recursive calls, which divide the sparse subspaces into even smaller subspaces to facilitate further pruning. Here we introduce a few performance improvement methods whose effectiveness has been verified by experiments.

### 5.1. Dimension Reordering

Suppose after determining major values, we find some of the dimensions have no major values. For example, $(\{M, I, *\}, \{I, *\}, \{M, I, *\})$. Then the four subspaces would be:

$$\begin{array}{llcccl} 1. & ( & \{M,*\}, & \{*\}, & \{M,*\} & ) \\ 2. & ( & I, & \{I,*\}, & \{M,I,*\} & ) \\ 3. & ( & \{M,*\}, & I, & \{M,I,*\} & ) \\ 4. & ( & \{M,*\}, & \{*\}, & I & ) \end{array}$$

Look at the 4-th subspace. Dimension 2 of that subspace is a single $*$, which means in fact it is a 2D subspace.

If we change the order of dimensions, we can get:

1. ( $\{M,*\}$, $\{*\}$, $\{M,*\}$ )
2. ( $\{M,I,*\}$, $I$, $\{M,I,*\}$ )
3. ( $I$, $\{*\}$, $\{M,I,*\}$ )
4. ( $\{M,*\}$, $\{*\}$, $I$ )

Subspace 3 and 4 are both 2-d subspaces now, better than the original dimension order. This is because fewer dimensions imply denser space and are more beneficial to the algorithm efficiency.

MM-Cubing V1.2 adopts the dimension reordering improvement. The implementation is straightforward: we only need to add an array parameter to the MM-Cubing function, storing the current dimension order.

### 5.2. Review of Determining Major Values

The dimension reordering improvement changes the running time in the recursive calls. Before doing dimension reordering, adding a major value will always decrease the running time in the recursive calls. But after doing dimension reordering, adding a major value might increase it, if the new major value is the first one in that dimension, since in this case, the number of dimensions of subspaces may increase. In another word, if we adopt the dimension reordering improvement, there will be a negative impact when we set the first value in one dimension to be a major value. Thus, it may be beneficial to keep some columns free of major values. There are three possible methods. The first is a simple modification in which we decrease the weight of the most frequent value in each dimension a little. The second is to use an alternative greedy strategy that first determines the number of columns that can contain major values, by looking at the number of tuples, *min_sup* and cardinalities, and then do the former greedy strategy. The third is to define a completely new target function, however, this is difficult since the simple *work_done* calculation does not reveal the impact of subspace dimensions.

MM-Cubing V1.21 adopts the first approach that decreases the weight of the first value by a factor of 2. MM-Cubing V1.22 adopts the new greedy strategy. It collects dimensions from the lowest (real) cardinality to the highest, and multiplies those cardinalities, till it reaches the aggregation table size. Only the collected dimensions will have major values. Other dimensions will have no major values.

### 5.3. Dynamic Aggregation Table Size

The maximum aggregation table size in the versions of MM-Cubing discussed so far is fixed at 64K. This allows simultaneous aggregation to use 2-byte unsigned short as index, which will gain 10% speedup compared to 4-byte integer. However, it is quite clear that dynamic aggregation table size (*AT size* in short) will make the algorithm more scalable to input data sets.

In order to maximize the performance, the AT size should be larger than a constant because of the overhead introduced to small aggregation table. (Anyway, the AT size could be one in some cases when the data is too sparse.) The AT size should not be too large because it may involve a lot of pruned cells.

There are two approaches to calculate the dynamic AT size: The first is to calculate before the major value selection step, and the second is to decide whether to stop during the major value selection step. The first approach is simple while the second is better, because it can make use of the target function value in decision making.

A simple approach in the first category is to use the number of tuples as the AT size. And we define the high and low limit for the AT size so that the aggregation table is not too large to fit in memory, nor is it too small compared to the overhead. The reason for using the number of tuples as the AT size is that we always need to scan the tuples in the counting step, thus it does not increase the complexity since the aggregation step did almost the same amount of work.

MM-Cubing V1.3 is a simple modification of V1.21, the most efficient version of all the previous ones, by adopting this approach.

## 6. Performance Study

To check the efficiency and adaptivity of the proposed algorithms, a comprehensive performance study[1] is conducted by testing MM-Cubing V1.3 against the best implementation we can achieve for the other two algorithms: BUC and Star-Cubing[2], based on the published literature. All the three algorithms are coded using C++ on a Pentium IV 2.4G system with 384MB of RAM. The system runs Windows XP Home Edition and VC 6.0 with Service Pack 5. All times recorded include both computation time and I/O time. Similar to other performance studies in cube computation [22, 6, 9, 21], all the tested data sets can fit in main memory. MultiWay is not tested, since it is limited to dense data sets with low cardinality and dimensionality, in which situation MM-Cubing runs simultaneous aggregation without recursive calls and has similar performance.

For the remaining of this section, $\mathcal{D}$ denotes the number of dimensions, $\mathcal{C}$ the cardinality of each dimension, $\mathcal{T}$ the number of tuples in the base cuboid, $\mathcal{M}$ the minimum support level, and $\mathcal{S}$ the skew or `zipf` of the data. When $\mathcal{S}$ equals 0, the data is uniform; as $\mathcal{S}$ increases, the data is more skewed. $\mathcal{S}$ is applied to all the dimensions. Values in different dimensions are independently generated, which is a neutral case for MM-Cubing. Please refer to Section

[1]All object codes and test data mentioned in this section can be downloaded from http://www.cs.uiuc.edu/∼hanj/software/mmcubing.htm

[2]We used an enhanced version of Star-Cubing, which runs faster than the original one in most cases. Due to space limit, we omit the details.

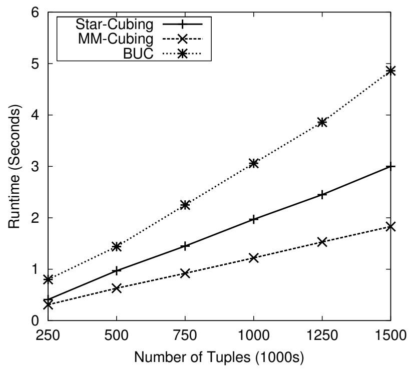

**Figure 4.** Full Cube Computation w.r.t. Number of Tuples, where $\mathcal{D} = 5,\ \mathcal{C} = 5,\ \mathcal{S} = 0,\ \mathcal{M} = 1$

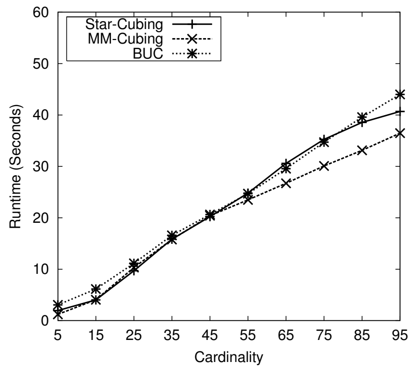

**Figure 5.** Full Cube Computation w.r.t. Cardinality, where $\mathcal{T} = 1M,\ \mathcal{D} = 5,\ \mathcal{S} = 0,\ \mathcal{M} = 1$

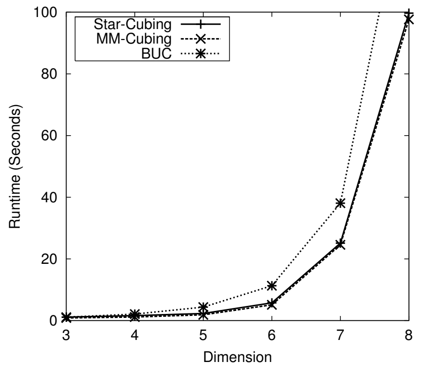

**Figure 6.** Full Cube Computation w.r.t. Dimension, where $\mathcal{T} = 1M,\ \mathcal{C} = 10,\ \mathcal{S} = 0,\ \mathcal{M} = 1$

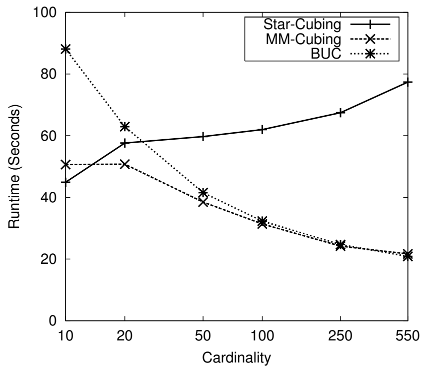

**Figure 7.** Iceberg Cube Computation w.r.t. Cardinality, where $\mathcal{T} = 1M,\ \mathcal{D} = 7,\ \mathcal{S} = 0,\ \mathcal{M} = 100$

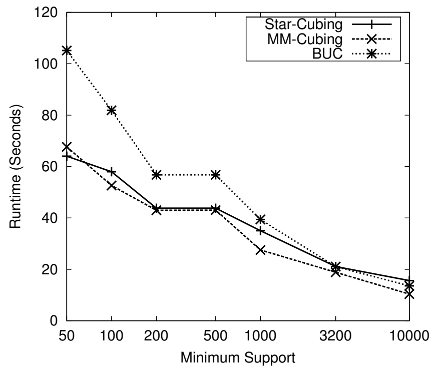

**Figure 8.** Iceberg Cube Computation w.r.t. Minsup, where $\mathcal{T} = 1M,\ \mathcal{D} = 10,\ \mathcal{C} = 10,\ \mathcal{S} = 0$

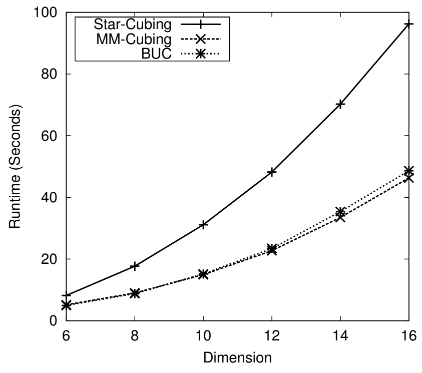

**Figure 9.** Iceburg Cube Computation w.r.t. Dimension, where $\mathcal{T} = 1M,\ \mathcal{C} = 100,\ \mathcal{S} = 1,\ \mathcal{M} = 1000$

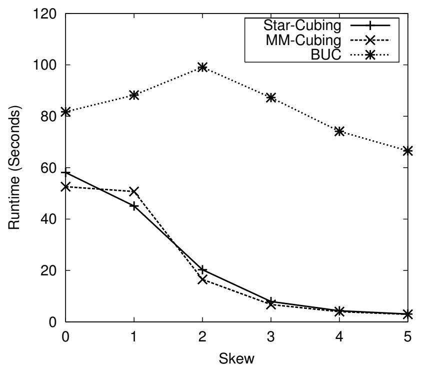

**Figure 10.** Data Skew, where $\mathcal{T} = 1M,\ \mathcal{D} = 10,\ \mathcal{C} = 10,\ \mathcal{M} = 100$

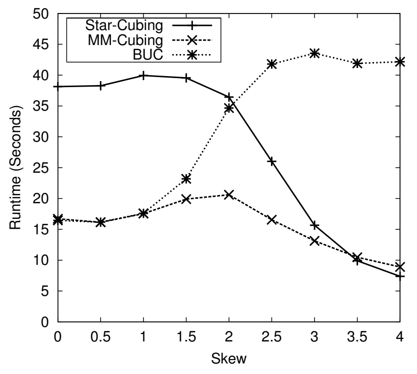

**Figure 11.** Data Skew, where $\mathcal{T} = 1M,\ \mathcal{D} = 9,\ \mathcal{C} = 25\,to\,400,\ \mathcal{M} = 100$

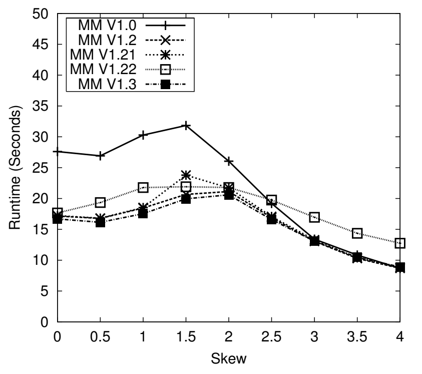

**Figure 12.** Different Versions of MM-Cubing, where $\mathcal{T} = 1M,\ \mathcal{D} = 9,\ \mathcal{C} = 25\,to\,400,\ \mathcal{M} = 100$

7 for more on how the dependence in different dimensions affects the efficiency of MM-Cubing.

**1. Full Cube Computation.** The experimental results for full cube computation are shown in Figures 4–6. We did not use more dimensions or greater cardinality because in high dimension or high cardinality datasets, the output gets extremely large and the I/O time dominates the cost of computation. This phenomenon is also observed in [6] and [15]. Moreover, the existing curves have clearly demonstrated the trends of the performance with the increase of dimensions and cardinality.

There are two main points that can be taken from these results. First, BUC is the worst in these test cases. This is because the cost of partition is quite high, and BUC needs to do all the partitions since there is no Apriori pruning in full cube computation.

Second, MM-Cubing is the best in Figure 4 and is almost the same as Star-Cubing in Figure 6. In Figure 4, MM-Cubing factorizes the lattice space into a single dense subspace. There is no recursive call and this enables MM-Cubing to achieve really high performance. In Figure 5, when the cardinality is large enough, Star-Cubing is inferior to MM-Cubing because of the cost introduced to a large number of child nodes in the visit of the star trees.

**2. Iceberg Cube Computation.** The second set of experiments compares the three algorithms for iceberg cube computation.

Figure 7 shows the performances with respect to cardinality. BUC performs better in sparse cases (when cardinality is large) and Star-Cubing performs better in dense cases (when cardinality is small). MM-Cubing performs uniformly well no matter it is sparse or dense.

Figure 8 shows the performances with respect to *min_sup*. When *min_sup* is $50$, BUC performs poorly because it relies more on the Apriori pruning, and small *min_sup* tends to prevent pruning until the chunk is very small. As the *min_sup* increases, all three algorithms speed up and the difference is shrinking. MM-Cubing and Star-Cubing perform almost the same in this figure.

In Figure 9, the running times of all the algorithms increase rapidly with respect to dimension, and Star-Cubing performs the worst due to high cardinality while MM-Cubing performs slightly better than BUC.

**3. Skewed Data Sets.** The third set of experiments tests the algorithms with respect to data skews. We use `zipf` to control the skew, varying from 0 to 5 (0 being uniform).

Figure 10 shows a 10-d data set with 1M tuples. The cardinality is 10 for all dimensions, and the *min_sup* is 100. BUC performs poorly compared to the other two: Although the data is not very dense, small *min_sup* prohibits BUC from early pruning. Note that BUC is the slowest when `zipf` is around 2. This is because data with small skew is almost uniform and BUC can start pruning before partitioning too many times, and data with large skew has a large portion of lattice cells that can be pruned immediately.

Figure 11 is the most representative data set for real data sets. In this Figure, the data set has $1$M tuples, $9$ dimensions. The cardinalities of each dimension vary from $25$ to $400$. The algorithms are tested against data skew. When the skew is $0$, Star-Cubing performs poorly due to the sparse property of the data. When the data is very skewed, BUC performs poorly due to the dense part of the data. In all cases, MM-Cubing is the best or very near to the best.

**4. Comparing Different Versions of MM-Cubing.** Figure 12 shows the relative efficiency of different versions of MM-Cubing, using the same data set as Figure 11. (There is no V1.1 simply because we don't have V1.1 in the series.) MM-Cubing V1.2 is better than V1.0 because it supports dimension reordering. V1.22 is inferior to V1.2 in most cases, because it cares too much about dimension reordering and does not choose an enough dense subspace. V1.21 is a little better than V1.2 since it takes into account the penalty of introducing the first major value in one dimension. MM-Cubing V1.3 is the best of all.

In summary, we have tested three cubing algorithms: BUC, Star-Cubing and MM-Cubing, with variations of density, *min_sup*, cardinality and skewness. For dense data, Star-Cubing is good and BUC is poor. For sparse data, BUC is good and Star-Cubing is poor. Both algorithms are poorer than MM-Cubing when the data is heterogenous (medium skewed, partly dense, and partly sparse). MM-Cubing performs uniformly well on all the data sets. Although there is no all-around clear-cut winner; however, in most cases, MM-Cubing performs better or substantially better than others.

## 7. Discussions

Although MM-Cubing performs well on an extensive set of data, there are still some problems that needs further improvements. One of these is memory consumption. Currently MM-Cubing requires roughly $3$ times of memory compared to the original table. The recursive calls of MM-Cubing require large memory to store the *shared array*, since recovering from the non-aggregatable value needs it. If the recursive call involves about half of all the tuples, the related memory usage is about twice of the original table, regardless of dimension and cardinality:

$$1 + 1/2 + 1/4 + ... = 2$$

The original tuples need to be in memory as well, so roughly $3$ times of memory is needed in total.

Generally, a recursive call will not involve so many tuples, since it is on a minor value. A value occurring in more than half of the tuples will almost always be a major value.

If the original table is extremely large and the memory is limited, the *shared array* can be stored on disk since only sequential access is needed. In this way, we need only two columns of the current *shared array* plus the original table: $(2 + D) \times T$, where $D$ is the number of dimensions and $T$ is the number of tuples. The original table size is $D \times T$.

MM-Cubing performs best when major values in different dimensions are correlated (appear in the same tuples), since in this case the dense subspace will be very dense and the simultaneous aggregation is extremely efficient. The experiments are all based on data with dimensional independence. The worst case for MM-Cubing will be that major values in one dimension are always correlated with minor values in other dimensions. However, although sometimes it may happen in real datasets, it is highly unlikely that it holds for all factorizations in the recursive calls. Even in this case, MM-Cubing won't perform much worse than BUC, since the computation time for the dense subspace is small compared to recursive calls.

## 8. Conclusions

In this paper, we present a new approach for cube computation, which factorizes the lattice space and is rather different from the previous lattice-based approaches. An efficient and adaptive algorithm, MM-Cubing, is developed based on this idea, which performs uniformly well on various data sets. A few optimization techniques are further explored, two of which are worth noting: (1) *dimension reordering* accelerates the computation by generating subspaces with lower dimensions; and (2) *different major value choosing strategies* may affect the efficiency. The simple greedy strategy performs well, but it is possible to gain more by introducing new major value choosing strategy.

Our performance study demonstrates that MM-Cubing is a promising one. For the uniform data distribution, MM-Cubing is almost the same as the better one of Star-Cubing and BUC and is significantly better than the worse one. When the data is skewed, MM-Cubing is better than both. Thus MM-Cubing is the only cubing algorithm so far that has uniformly high performance in all the data distributions.

For future work, it is interesting to extend the factorization methodology for efficient computation of condensed or quotient cubes, approximate cubes [4], cube-gradients [12], as well as the integration of cube analysis and clustering analysis.

## References

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

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

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

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

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

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

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

[8] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. Data Mining and Knowledge Discovery, 1:29–54, 1997.

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

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

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

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

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

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

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

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

[17] A. Shukla, P. M. Deshpande, and J. F. Naughton. Materialized view selection for multidimensional datasets. VLDB'98.

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

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

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

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

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