An Array-Based Algorithm for Simultaneous Multidimensional Aggregates *

Yihong Zhao
Computer Sciences Department
University of Wisconsin-Madison
zhao@cs.wisc.edu

Prasad M. Deshpande
Computer Sciences Department
University of Wisconsin-Madison
pmd@cs.wisc.edu

Jeffrey F. Naughton
Computer Sciences Department
University of Wisconsin-Madison
naughton@cs.wisc.edu

Abstract

Computing multiple related group-bys and aggregates is one of the core operations of On-Line Analytical Processing (OLAP) applications. Recently, Gray et al. [GBLP95] proposed the "Cube" operator, which computes group-by aggregations over all possible subsets of the specified dimensions. The rapid acceptance of the importance of this operator has led to a variant of the Cube being proposed for the SQL standard. Several efficient algorithms for Relational OLAP (ROLAP) have been developed to compute the Cube. However, to our knowledge there is nothing in the literature on how to compute the Cube for Multidimensional OLAP (MOLAP) systems, which store their data in sparse arrays rather than in tables. In this paper, we present a MOLAP algorithm to compute the Cube, and compare it to a leading ROLAP algorithm. The comparison between the two is interesting, since although they are computing the same function, one is value-based (the ROLAP algorithm) whereas the other is position-based (the MOLAP algorithm.) Our tests show that, given appropriate compression techniques, the MOLAP algorithm is significantly faster than the ROLAP algorithm. In fact, the difference is so pronounced that this MOLAP algorithm may be useful for ROLAP systems as well as MOLAP systems, since in many cases, instead of cubing a table directly, it is faster to first convert the table to an array, cube the array, then convert the result back to a table.

1 Introduction

Computing multiple related group-bys and aggregates is one of the core operations of On-Line Analytical Processing (OLAP) applications. Recently, Gray et al. [GBLP95] proposed the "Cube" operator, which computes group-by aggregations over all possible subsets of the specified dimensions. The rapid acceptance of the importance of this operator has led to a variant of the Cube being proposed for the SQL standard. Several efficient algorithms for Relational OLAP (ROLAP) have been developed to compute the Cube. However, to our knowledge there is nothing to date in the literature on how to compute the Cube for Multidimensional OLAP (MOLAP) systems.

For concreteness, consider a very simple multidimensional model, in which we have the dimensions *product*, *store*, *time*, and the "measure" (data value) sales. Then to compute the "cube" we will compute sales grouped by all subsets of these dimensions. That is, we will have sales by product, store, and date; sales by product and store; sales by product and date; sales by store and date; sales by product; sales by store; sales by date; and overall sales. In multidimensional applications, the system is often called upon to compute all of these aggregates (or at least a large subset of them), either in response to a user query, or as part of a "load process" that precomputes these aggregates to speed later queries. The challenge, of course, is to compute the cube with far more efficiency than the naive method of computing each component aggregate individually in succession.

MOLAP systems present a different sort of challenge in computing the cube than do ROLAP systems. The main reason for this is the fundamental difference in the data structures in which the two systems store their data. ROLAP systems (for example, MicroStrategy [MS], Informix's Metacube [MC], and Information Advantage [IA]) by definition use relational tables as their data structure. This means that a "cell" in a logically multidimensional space is represented in the system as a tuple, with some attributes that identify the location of the tuple in the multidimensional space, and other attributes that contain the data value corresponding to that data cell. Returning to our example, a cell of the array might be represented by the tuple `(shoes, WestTown, 3-July-96, $34.00)`. Computing the cube over such a table requires a generalization of standard relational aggregation operators [AADN96]. In prior work, three main ideas have been used to make ROLAP computation efficient:

1. Using some sort of grouping operation on the dimension attributes to bring together related tuples (e.g., sorting or hashing),
2. Using the grouping performed on behalf of one of the sub-aggregates as a partial grouping to speed the computation another sub-aggregate, and
3. To compute an aggregate from another aggregate, rather than from the (presumably much larger) base table.

By contrast, MOLAP systems (for example, Essbase from Arbor Software [CCS93, RJ, AS], Express from Oracle [OC],

*This work supported by NSF grant IRI-9157357, a grant under the IBM University Partnership Program, and ARPA contract DAAB07-91-C-Q518

and LightShip from Pilot [PSW]) store their data as sparse arrays. Returning to our running example, instead of storing the tuple `(shoes, WestTown, 3-July-1996, $34.00)`, a MOLAP system would just store the data value `$34.00`; the position within the sparse array would encode the fact that this is a sales volume for shoes in the West Town store on July 3, 1996. When we consider computing the cube on data stored in arrays, one can once again use the ROLAP trick of computing one aggregate from another. However, none of the other techniques that have been developed for ROLAP cube computations apply. Most importantly, there is no equivalent of "reordering to bring together related tuples" based upon their dimension values. The data values are already stored in fixed positions determined by those dimension values; the trick is to visit those values in the right order so that the computation is efficient. Similarly, there is no concept of using an order generated by one subaggregate in the computation of another; rather, the trick is to simultaneously compute spatially-delimited partial aggregates so that a cell does not have to be revisited for each sub-aggregate. To do so with minimal memory requires a great deal of care and attention to the size of the dimensions involved. Finally, all of this is made more complicated by the fact that in order to store arrays efficiently on disk, one must "chunk" them into small memory-sized pieces, and perform some sort of "compression" to avoid wasting space on cells that contain no valid data.

In this paper, we present a MOLAP algorithm incorporating all of these ideas. The algorithm succeeds in overlapping the computation of multiple subaggregates, and makes good use of available main memory. We prove a number of theorems about the algorithm, including a specification of the optimal ordering of dimensions of the cube for reading chunks of base array, and an upper bound on the memory requirement for a one-pass computation of the cube that it is in general much smaller than the size of the original base array.

We have implemented our algorithm and present performance results for a wide range of dimension sizes, data densities, and buffer-pool sizes. We show that the algorithm performs significantly faster than the naive algorithm of computing aggregates separately, even when the "naive" algorithm is smart about computing sub-aggregates from super-aggregates rather than from the base array. We also compared the algorithm with an implementation of a previously-proposed ROLAP cube algorithm, and found that the MOLAP algorithm was significantly faster.

Clearly, this MOLAP cube algorithm can be used by a multidimensional database system. However, we believe it may also have some applicability within relational database systems as part of support for multidimensional database applications, for two reasons. First, as relational database systems provide richer and richer type systems, it is becoming feasible to implement arrays as a storage device for RDBMS data. In another paper [ZTN], we explored the performance implications of such an approach for "consolidation" operations; the study in this paper adds more weight to the conclusion that including array storage in relational systems can significantly enhance RDBMS performance for certain workloads.

The second application of this algorithm to ROLAP systems came as a surprise to us, although in retrospect perhaps we should have foreseen this result. Simply put, one can always use our MOLAP algorithm in a relational system by the following three-step procedure:

1. Scan the table, and load it into an array.
2. Compute the cube on the resulting array.
3. Dump the resulting cubed array into tables.

The result is the same as directly cubing the table; what was surprising to us was that this three-step approach was actually faster than the direct approach of cubing the table. In such a three-step approach, the array is being used as an internal data structure, much like the hash table in a hash join in standard relational join processing.

The rest of the paper is organized as follows. In Section 2, we introduce the chunked array representation, and then discuss how we compressed these arrays and our algorithm for loading chunked, compressed arrays from tables. We then present a basic array based algorithm in Section 3. Our new algorithm, the *Multi-Way* Array method, is described in Section 4, along with some theorems that show how to predict and minimize the memory requirements for the algorithm. We present the performance results in Section 5, and we conclude in Section 6.

2 Array Storage Issues

In this section we discuss the basic techniques we used to load and store large, sparse arrays efficiently. There are three main issues to resolve. First, it is highly likely in a multidimensional application that the array itself is far too large to fit in memory. In this case, the array must be split up into "chunks", each of which is small enough to fit comfortably in memory. Second, even with this "chunking", it is likely that many of the cells in the array are empty, meaning that there is no data for that combination of coordinates. To efficiently store this sort of data we need to compress these chunks. Third, in many cases an array may need to be loaded from data that is not in array format (e.g., from a relational table or from an external load file.) We conclude this section with a description of an efficient algorithm for loading arrays in our compressed, chunked format.

2.1 Chunking Arrays

As we have mentioned, for high performance large arrays must be stored broken up into smaller chunks. The standard programming language technique of storing the array in a row major or column major order is not very efficient. Consider a row major representation of a two-dimensional array, with dimensions *Store* and *Date*, where *Store* forms the row and *Date* forms the column. Accessing the array in the row order (order of *Stores*) is efficient with this representation, since each disk page that we read will contain several *Stores*. However, accessing in the order of columns (*Dates*) is inefficient. If the *Store* dimension is big, each disk page read will only contain data for one *Date*. Thus to get data for the next *Date* will require another disk access; in fact there will be one disk access for each *Date* required. The simple row major layout creates an asymmetry among the dimensions, favoring one over the other. This is because data is accessed from disk in units of pages.

To have a uniform treatment for all the dimensions, we can chunk the array, as suggested by Sarawagi [SM94]. Chunking is a way to divide an n-dimensional array into small size n-dimensional chunks and store each chunk as one object on disk. Each array chunk has n dimensions and will correspond to the blocking size on the disk. We will be using chunks which have the same size on each dimension.

2.2 Compressing Sparse Arrays

For dense chunks, which we define as those in which more than 40% of the array cells have a valid value, we do not compress the array, simply storing all cells of the array as-is but assigning a null value to invalid array cells. Each chunk therefore has a fixed length. Note that storing a dense multidimensional data set in an array is already a significant compression over storing the data in a relational table, since we do not store the dimension values. For example, in our running example we do not store product, store, or date values in the array.

However, for a sparse chunk, that is one with data density less than 40%, storing the array without compression is wasteful, since most of the space is devoted to invalid cells. In this case we use what we call "chunk-offset compression." In chunk-offset compression, for each valid array entry, we store a pair, (offsetInChunk,data). The offsetInChunk integer can be computed as follows: consider the chunk as a normal (uncompressed) array. Each cell c in the chunk is defined by a set of indices; for example, if we are working with a three-dimensional chunk, a given cell will have an "address" (i, j, k) in the chunk. To access this cell in memory, we would convert the triple (i, j, k) into an offset from the start of the chunk, typically by assuming that the chunk is laid out in memory in some standard order. This offset is the "offsetInChunk" integer we store.

Since in this representation chunks will be of variable length, we use some meta data to hold the length of each chunk and store the meta data at the beginning of the data file.

We also experimented with compressing the array chunks using a lossless compression algorithm (LZW compression [Wel84]) but this was far less effective for a couple of reasons. First, the compression ratio itself was not as good as the "chunk-offset compression." Intuitively, this is because LZW compression uses no domain knowledge, whereas "chunk-offset compression" can use the fact that it is storing array cells to minimize storage. Second, and perhaps most important, using LZW compression it is necessary to materialize the (possibly very sparse) full chunk in memory before it can be operated on. By contrast, with chunk-offset compression we can operate directly on the compressed chunk.

2.3 Loading Arrays from Tables

We have designed and implemented a partition-based loading algorithm to convert a relational table or external load file to a (possibly compressed) chunked array. As input the algorithm takes the table, along with each dimension size and a predefined chunk size. Briefly, the algorithm works as follows.

Since we know the size of the full array and the chunk size, we know how many chunks are in the array to be loaded. If the available memory size is less than the size of the resulting array, we partition the set of chunks into partitions so that the data in each partition fits in memory. (This partitioning is logical at this phase. For example, if we have 8 chunks 0 - 7, and we need two partitions, we would put tuples corresponding to cells that map to chunks 0-3 in partition one, and those that map to chunks 4-7 in partition two.)

Once the partitions have been determined, the algorithm scans the table. For each tuple, the algorithm calculates the tuple's chunk number and the offset from the first element of its chunk. This is possible by examining the dimension values in the tuple. The algorithm then stores this chunk number and offset, along with the data element, into a tuple, and inserts the tuple into the buffer page of the corresponding partition. Once any buffer page for a partition is full, the page is written to the disk resident file for this partition. In the second pass, for each partition, the algorithm reads in each partition tuple and assigns it to a bucket in memory according to its chunk number. Each bucket corresponds to a unique chunk. Once we assign all tuples to buckets, the algorithm constructs array chunks for each bucket, compresses them if necessary using chunk-offset compression, and writes those chunks to disk. One optimization is to compute the chunks of the first partition in the first pass. After we allocate each partition a buffer page, we allocate the rest of available memory to the buckets for the first partition. This is similar to techniques used in the Hybrid Hash Join algorithm [DKOS84] to keep the "first bucket" in memory.

3 A Basic Array Cubing Algorithm

We first introduce an algorithm to compute the cube of a chunked array in multiple passes by using minimum memory. The algorithm makes no attempt to overlap any computation, computing each "group by" in a separate pass. In the next section, we modify this simple algorithm to minimize the I/O cost and to overlap the aggregation of related group-bys.

First consider how to compute a group-by from a simple non-chunked array. Suppose we have a three dimensional array, with dimensions A, B, and C. Suppose furthermore that we want to compute the aggregate AB, that is, we want to project out C and aggregate together all these values. This can be seen as projecting onto the AB plane; logically, this can be done by sweeping a plane through the C dimension, aggregating as we go, until the whole array has been swept.

Next suppose that this ABC array is stored in a number of chunks. Again the computation can be viewed as sweeping through the array, aggregating away the C dimension. But now instead of sweeping an entire plane of size $|A||B|$, where $|A|$ and $|B|$ are the sizes of the A and B dimensions, we do it on a chunk by chunk basis. Suppose that the A dimension in a chunk has size A_c, and the B dimension in a chunk has size B_c. If we think of orienting the array so that we are looking at the AB face of the array (with C going back into the paper) we can begin with the chunk in the upper left-hand portion of the array, and sweep a plane of size A_cB_c back through that chunk, aggregating away the C values as we go. Once we have finished this upper left-hand chunk, we continue to sweep this plane through the chunk immediately behind the one on the front of the array in the upper left corner. We continue in this fashion until we have swept all the way through the array. At this point we have computed the portion of the AB aggregate corresponding to the upper-left hand sub-plane of size A_cB_c. We can store this plane to disk as the first part of the AB aggregate, and move on to compute the sub-plane corresponding to another chunk, perhaps the one immediately to the right of the initial chunk.

Note that in this way each chunk is read only once, and that at the end of the computation the AB aggregate will be on disk as a collection of planes of size A_cB_c. The memory used by this computation is only enough to hold one chunk, plus enough to hold the A_cB_c plane as it is swept through the chunks. This generalization of this algorithm to higher dimensions is straight-forward; instead of sweeping planes through arrays, in higher dimensions, say k dimensional arrays, one sweeps $k-1$ dimensional subarrays through the array.

Up to now we have discussed only computing a single aggregate of an array. But, as we have mentioned in the introduction, to "cube" an array requires computing all aggregates of the array. For example, if the array has dimensions ABC, we need to compute AB, BC, AC, and A, B, C, as well as the overall total aggregate. The most naive approach would be to compute all of these aggregates from the initial ABC array. A moment's thought shows that this is a very bad idea; it is far more efficient to compute A from AB than it is to compute A from ABC. This idea has been explored in the ROLAP cube computation literature [AADN96]. If we look at an entire cube computation, the aggregates to be computed can be viewed as a lattice, with ABC as the root. ABC has children AB, BC, and AC; AC has children A and C, and so forth. To compute the cube efficiently we embed a tree in this lattice, and compute each aggregate from its parent in this tree.

One question that arises is which tree to use for this computation? For ROLAP cube computations this is a difficult question, since the sizes of the tables corresponding to the nodes in the lattice are not known until they are computed, so heuristics must be used. For our chunk-based array algorithm we are more fortunate, since by knowing the dimension sizes of the array and the size of the chunks used to store the array, we can compute exactly the size of the array corresponding to each node in the lattice, and also how much storage will be needed to use one of these arrays to compute a child. Hence we can define the "minimum size spanning tree" for the lattice. For each node n in the lattice, its parent in the minimum size spanning tree is the node n' which has the minimum size and from which n can be computed.

We can now state our basic array cubing algorithm. We first construct the minimum size spanning tree for the group-bys of the Cube. We compute any group-by $D_{i_1}D_{i_2}..D_{i_k}$ of a Cube from the "parent" $D_{i_1}D_{i_2}..D_{i_{k+1}}$, which has the minimum size. We read in each chunk of $D_{i_1}D_{i_2}..D_{i_{k+1}}$ along the dimension $D_{i_{k+1}}$ and aggregate each chunk to a chunk of $D_{i_1}D_{i_2}..D_{i_k}$. Once the chunk of $D_{i_1}D_{i_2}..D_{i_k}$ is complete, we output the chunk to disk and use the memory for the next chunk of $D_{i_1}D_{i_2}..D_{i_k}$. Note that we need to keep only one $D_{i_1}D_{i_2}..D_{i_k}$ chunk in memory at any time.

In this paper, we will use a three dimensional array as an example. The array ABC is a $16 \times 16 \times 16$ array with $4 \times 4 \times 4$ array chunks laid out in the dimension order ABC (see Figure 1). The order of layout is indicated by the chunk numbers shown in the figure. The chunks are numbered from 1 to 64. The Cube of the array consists of the group-bys AB, AC, BC, B, C, A, and ALL. For example, to compute the BC group-by, we read in the chunk number order from 1 to 64, aggregate each four ABC chunks to a BC chunk, output the BC chunk to disk, and reuse the memory for the next BC chunk.

While this algorithm is fairly careful about using a hierarchy of aggregates to compute the cube and using minimal memory for each step, it is somewhat naive in that it computes each subaggregate independently. In more detail, suppose we are computing AB, AC, and BC from ABC in our example. This basic algorithm will compute AB from ABC, then will re-scan ABC to compute AC, then will scan it a third time to compute BC. In the next few sections we discuss how to modify this algorithm to compute all the children of a parent in a single pass of the parent.

4 The Multi-Way Array Algorithm

We now present our multi-way array cubing algorithm. This algorithm overlaps the computations of the different group-bys, thus avoiding the multiple scans required by the naive algorithm. Recall that a data Cube for a n-dimensional array contains multiple related group-bys. Specifically, it consists of 2^n group-bys, one for each subset of the dimensions. Each of these group-bys will also be represented as arrays. Ideally, we need memory large enough to hold all these group-bys so that we can overlap the computation of all those group-bys and finish the Cube in one scan of the array. Unfortunately, the total size of the group-bys is usually much larger than the buffer pool size. Our algorithm tries to minimize the memory needed for each computation, so that we can achieve maximum overlap. We will describe our algorithm in two steps. Initially we will assume that there is sufficient memory to compute all the group-bys in one scan. Later we will extend it to the other case where memory is insufficient.

4.1 A Single-pass Multi-way Array Cubing Algorithm

As we showed in the naive algorithm, it is not necessary to keep the entire array in memory for any group-by — keeping only the relevant part of the array in memory at each step will suffice. Thus we will be reducing memory requirements by keeping only parts of the group-by arrays in memory. When computing multiple group-bys simultaneously, the total memory required depends critically on the order in which the input array is scanned. In order to reduce this total amount of memory our algorithm makes use of a special logical order called "dimension order".

4.1.1 Dimension Order

A dimension order of the array chunks is a row major order of the chunks with the n dimensions $D_1, D_2, .., D_n$ in some order $\mathcal{O} = (D_{j_1}, D_{j_2}, .., D_{j_n})$. Different dimension orders $\mathcal{O}'$ lead to different orders of reading the array chunks. Note that this logical order of reading is independent of the actual physical layout of the chunks on the disk. The chunks of array may be laid out on the disk in an order different from the dimension order. We will now see how the dimension order determines the amount of memory needed for the computation.

4.1.2 Memory Requirements

Assuming that we read in the array chunks in a dimension order, we can formulate a general rule to determine what chunks of each group-by of the cube need to stay in memory in order to avoid rescanning a chunk of the input array. We use the above 3-D array to illustrate the rule with an example.

The array chunks are read in the dimension order ABC, i.e., from chunk 1 to chunk 64. Suppose chunk 1 is read in. For group-by AB, this chunk is aggregated along the C dimension to get a chunk of AB. Similarly for AC and BC, this chunk is aggregated along B and A dimensions respectively. Thus the first chunk's AB group-by is aggregated to the chunk a_0b_0 of AB; the first chunk's AC is aggregated to the chunk a_0c_0 of AC; the first chunk's BC is aggregated to the chunk b_0c_0 of BC. As we read in new chunks, we aggregate the chunk's AB, AC and BC group-by to the corresponding chunks of group-bys AB, AC and BC. To compute each chunk of AB, AC, and BC group-by, we may

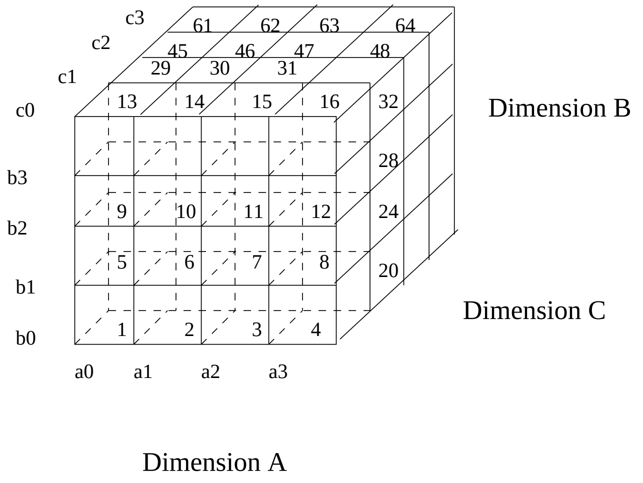

Figure 1: 3 D array

naively allocate memory to each chunk of those group-bys in memory. However, we can exploit the order in which each chunk is brought in memory to reduce the memory required by each group-by to the minimum so that we can compute the group-bys AC, AB, and BC in one scan of the array ABC.

Let us look into how we compute each chunk of those group-bys in detail. Notice that we read the chunks in dimension order (A, B, C) layout, which is a linear order from chunk 1 to chunk 64. For the chunk 1 to chunk 4, we complete the aggregation for the chunk b_0c_0 of BC after aggregating each chunk's BC group-by to the chunk b_0c_0 of BC. Once the b_0c_0 chunk is completed, we write out the chunk and reassign the chunk memory to the chunk b_1c_0, which is computed from the next 4 chunks of ABC i.e. the chunk 4 to chunk 8. So we allot only one chunk of BC in memory to compute the entire BC group-by. Similarly, we allocate memory to the chunks a_0c_0, a_1c_0, a_2c_0, and a_3c_0 of the AC group-by while scanning the first 16 chunks of ABC. To finish the aggregation for the chunk a_0c_0, we aggregate the AC of the chunks 1, 5, 9, and 13, to the chunk a_0c_0. After we aggregate the first 16 chunks of AC to those chunks of AC, the aggregation for those AC chunks are done. We output those AC chunks to disk in order of (A, C) and reassign those chunks' memory to the a_0c_1, a_1c_1,a_2c_1, and a_3c_1 of the AC group-by. To compute the AB group-by in one scan of the array ABC, we need to allocate memory to each of the 16 chunks of AB. For the first 16 chunks of ABC, we aggregate each chunk's AB to the corresponding AB chunks. The aggregation for those AB is not complete until we aggregate all 64 chunks' AB to those AB chunks. Once the aggregation for AB chunks is done, we output those chunks in (A, B) order.

Notice that we generate each BC chunk in the dimension order (B, C). So, before we write each BC chunk to disk, we use the BC chunks to compute the chunks of B or C as if we read in each BC chunk in the dimension order (B, C). Generally, the chunks of each group-bys of the Cube are generated in a proper dimension order. In fact, this is the key to apply our general memory requirement rule recursively to the nodes of the minimum memory spanning tree (MMST) and overlap computation for the Cube group-bys. We will explain this idea in detail when we discuss the MMST.

In this example, for computing BC we need memory to hold 1 chunk of BC, for AC we need memory to hold 4 chunks of AC and for AB we need memory to hold $4*4 = 16$ chunks of AB. Generalizing, we allocate $|B_c||C_c|u$ memory to BC group-by, $|A_d||C_c|u$ to AC group-by, and $|A_d||B_d|u$ to AB group-by, where $|X_d|$ stands for the size of dimension X, $|Y_c|$ stands for the chunk size of dimension Y, and u stands for the size of each chunk element. The size of the chunk element is same as the array element size which depends on the type of the array. For an integer array, each array element takes four bytes. There is a pattern for allocating memory to AB, AC, and BC group-bys for the dimension order (A, B, C). If XY contains a prefix of ABC with the length p, then we allocate $16^p \times 4^{2-p} \times u$ memory to XY group-bys. This is because each dimension is of size 16 and each chunk dimension has size 4. To generalize this for all group-bys of a n-dimensional array, we have the following rule.

Rule 1 *For a group-by $(D_{j_1}, .., D_{j_{n-1}})$ of the array $(D_1, .., D_n)$ read in the dimension order $\mathcal{O} = (D_1, .., D_n)$, if $(D_{j_1}, .., D_{j_{n-1}})$ contains a prefix of $(D_1, .., D_n)$ with length p, $0 \le p \le n-1$, we allocate $\prod_{i=1}^{p} |D_i| \times \prod_{i=p+1}^{n-1} |C_i|$ units of array element to $(D_{j_1}, .., D_{j_{n-1}})$ group-by, where $|D_i|$ is the size of dimension i and $|C_i|$ is the chunk size of dimension i.*

$|C_i|$ is much smaller than $|D_i|$ for most dimensions. Thus, according to the **Rule 1**, we allocate an amount of memory less than the size of the group-by for many of the group-bys. The benefit of reducing the memory allocated to each group-by is to compute more group-bys of the Cube simultaneously and overlap the computation of those group-bys to a higher degree. We need some kind of structure to coordinate the overlapped computation. A spanning tree on the lattice of group-bys can be used for this purpose. For a given dimension order, different spanning trees will require different amounts of memory. We define a minimum memory spanning tree in the next section.

4.1.3 Minimum Memory Spanning Tree

A MMST for a Cube $(D_1, .., D_n)$ in a dimension order $\mathcal{O} = (D_{j_1}, .., D_{j_n})$ has $n + 1$ levels with the root $(D_{j_1}, .., D_{j_n})$ at level n. Any tree node N at level i below the level n may be computed from those nodes at one level up whose dimensions contain the dimensions of node N. For any node N at level i, there may be more than one node at level $i+1$ from which it can be computed. We choose the node that makes the node N require the minimum memory according to the **Rule 1**. In other words, the prefix of the parent node contained in node N has the minimum length. So a MMST, for a given dimension order, is minimum in terms of the total memory requirement for that dimension order. If node N contains the minimum prefix for several upper level nodes, we use the size of those nodes to break the tie and choose the node with the minimum size as the parent of the node N.

Once we build the MMST for the Cube in a dimension order $\mathcal{O}$ we can overlap the computation of the MMST subtrees. We use the same example, the array ABC, to explain how to do it. Let us assume that we have enough memory to allocate each node's required memory. The MMST for the array ABC in a dimension order (A, B, C) is shown in Figure 2. As mentioned before, chunks of BC, AC, and AB are calculated in dimension orders (B, C), (A, C), and (A, B) in memory since we read ABC chunks in dimension order (A, B, C) to produce each chunk of BC, AC, and AB. To each node A, B,and C, this is equivalent to reading in chunks of group-by AB and AC in the dimension order (A, B) and

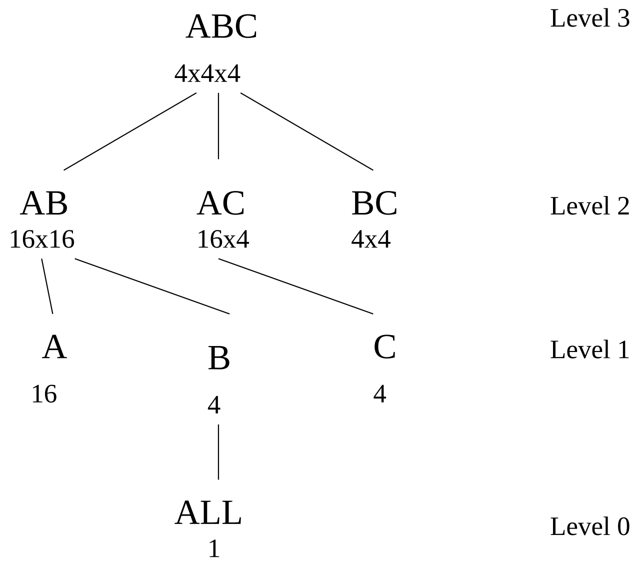

Figure 2: 3-D array MMST in dimension order (A, B, C)

(A, C). Similar to the nodes of the level 2, the chunks of the nodes A, B, and C are generated in the proper dimension orders. To generalize for any MMST, the nodes from the level n to the level 0, the chunks of each tree node are generated in a proper dimension order. Therefore, we can recursively apply the **Rule 1** to the nodes from the level n to the level 0 so that we allocate minimum number of chunks to each nodes instead of all chunks. Furthermore, we can compute the chunks of each tree node simultaneously. For example, we can aggregate the chunk a_0c_0 of AC along C dimension to compute the chunk c_0 of C after we aggregate the chunk 1, 5, 9, 13 of ABC to the chunk a_0c_0 and before we write the chunk a_0c_0 to disk. Generally, if we allocate each MMST node its required memory we can compute the chunks of the tree nodes from the top level to the level 0 simultaneously.

We now give a way of calculating the memory required for the MMST of any given dimension order $\mathcal{O} = (D_1, D_2, \ldots, D_n)$. We will assume that each array element takes u bytes. In addition, all the numbers used for the memory size in the following sections are in units of the array element size.

Memory requirements for the MMST

Let us assume that the chunk size is the same for each dimension, i.e., for all i, $|C_i| = c$. We can calculate the memory required by each tree node at each level of the MMST using **Rule 1**. We have the root of the MMST at the level n and allocate c^n to the root $D_1..D_n$. At the level $n-1$, which is one level down from the root $D_1..D_n$, we have the nodes: $D_1..D_{n-2}D_{n-1}$, $D_1..D_{n-2}D_n$, $\cdots$, and $D_2D_3..d_n$. Each node omits one dimension of the root dimensions $D_1..D_n$. So each node contains a prefix of the root $(D_1..D_n)$. The length of the prefix for each above node is $n-1$, $n-2$, $\cdots$, and 0. According to the **Rule 1**, the sum of memory required by those nodes is

$$\prod_{i=1}^{n-1}|D_i| + (\prod_{i=1}^{n-2}|D_i|)c + (\prod_{i=1}^{n-3}|D_i|)c^2 + \cdots + c^{n-1}.$$

At level $n-2$, we classify the tree nodes into the following types according to the length of the prefix of the root contained in those nodes: $D_1..D_{n-2}$, $D_1..D_{n-3}W_1$, $\cdots$, $D_1W_1W_2..W_{n-2}$, and $W_1W_2..W_{n-2}$. For the type $D_1..D_k$-$W_1..W_{n-2-k}$, the nodes start with the prefix $D_1..D_k$ of the root and followed by W_i, which are those dimensions not included in $D_1, D_2, \cdots, D_k$ and D_{k+1}. So there are $C(n-(k+1), n-2-k)$ nodes belonging to this type, i.e. we are choosing $n-2-k$ dimensions from $n-(k+1)$ dimensions. We use $n-(k+1)$ since we should not choose the dimension D_{k+1} for W_i. If we do so, the node will become the type of $D_1..D_{k+1}W_1..W_{n-2-(k+1)}$ instead of the type of $D_1..D_kW_1..W_{n-2-k}$. Hence the sum of the memory required by the nodes at this level is:

$$\prod_{i=1}^{n-2}|D_i| + C(2,1)(\prod_{i=1}^{n-3}|D_i|)c + C(3,2)(\prod_{i=1}^{n-4}|D_i|)c^2 + \cdots + C(n-1, n-2)c^{n-2}.$$

Similarly, we calculate the total memory required by the nodes at the level $n-3$. We have the sum:

$$\prod_{i=1}^{n-3}|D_i| + C(3,1)(\prod_{i=1}^{n-4}|D_i|)c + C(4,2)(\prod_{i=1}^{n-5}|D_i|)c^2 + \cdots + C(n-1, n-3)c^{n-3}.$$

In general we get the following rule.

Rule 2 *The total memory requirement for level j of the MMST for a dimension order $\mathcal{O} = (D_1, .., D_n)$ is given by :*

$$\prod_{i=1}^{n-j}|D_i| + C(j,1)(\prod_{i=1}^{n-j-1}|D_i|)c + C(j+1,2)(\prod_{i=1}^{n-j-2}|D_i|)c^2 + \cdots + C(n-1, n-j)c^{n-j}.$$

As a further example, the sum of the memory for level 1 nodes is $D_1 + C(n-1, 1)c$. At the level 0, there is one node "ALL" and it requires c amount of memory.

For different dimension orders of the array $(D_1, .., D_n)$, we may generate different MMSTs, which may have profoundly different memory requirements. To illustrate this, we use a four dimension array $ABCD$ which has 10x10x10x10 chunks. The sizes of dimensions A, B, C, and D are 10, 100, 1000, and 10000. The MMSTs for the dimension order (A, B, C, D) and for the dimension order (D, B, C, A) are shown in Figures 3 and 4. The number below each group-by node in the figures is the number of units of array element required by the node. Adding up those numbers for each MMST, we find the MMST for the order (D, B, C, A) requires approximately 4GB for a one-pass computation, whereas the tree for the order (A, B, C, D) requires only 4MB. On investigating the reason for this difference between the two trees, we find that switching the order of A and D changes the amount of memory required by each tree node. Clearly, it is important to determine which dimension order will require the least memory.

4.1.4 Optimal Dimension Order

The optimal dimension order is the dimension order whose MMST requires the least amount of memory. We prove that the optimal dimension order $\mathcal{O}$ is $(D_1, D_2, \ldots, D_n)$, where $|D_1| \leq |D_2| \leq \cdots \leq |D_n|$. Here, $|D_i|$ denotes size of the dimension D_i. So the dimensions are ordered incrementally in the dimension order $\mathcal{O}$.

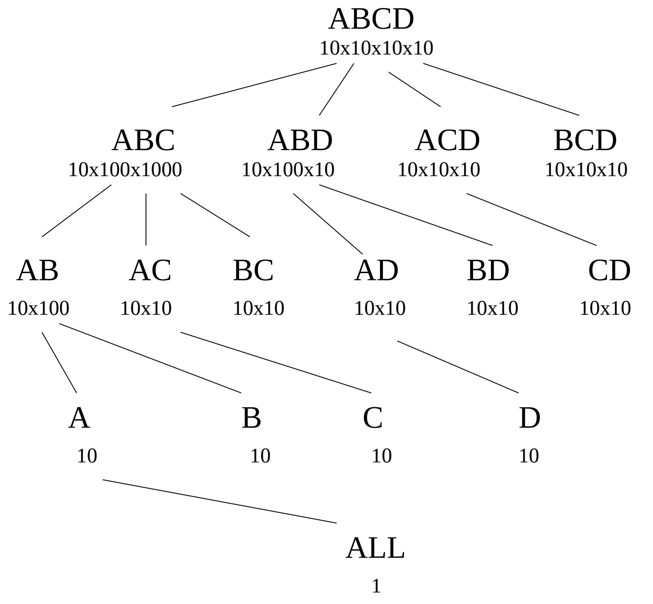

Figure 3: MMST for Dimension Order ABCD (Total Memory Required 4 MB)

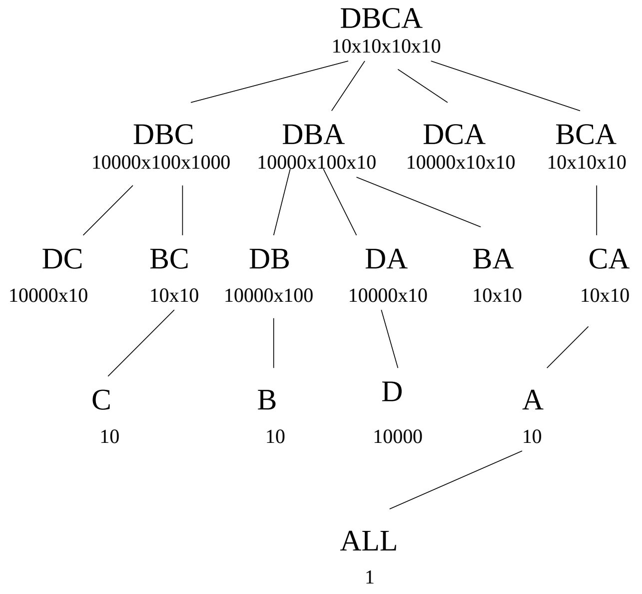

Figure 4: MMST for Dimension Order DBCA (Total Memory Required 4 GB)

Theorem 1 *Consider a chunked multidimensional array A of size* $\prod_{i=1}^{n} |D_i|$ *and having chunks of size* $\prod_{i=1}^{n} |C_i|$*, where* $|C_i| = c$ *for all* $i (1 \leq i \leq n)$*. If we read the chunks in logical order* $\mathcal{O}$ *, where* $\mathcal{O} = (D_1, D_2, , , D_n)$ *and* $|D_1| \leq |D_2| \leq |D_3| \cdots \leq |D_n|$*, the total amount of memory required to compute the Cube of the array in one scan of A is minimum.*

The question that naturally follows is "What is the upper bound for the total amount of memory required by MMST $\mathcal{T}_{\mathcal{O}}$?" The next theorem and corollary answer this question.

Theorem 2 *For a chunked multidimensional array A with the size* $\prod_{i=1}^{n} |D_i|$*, where* $|D_i| = d$ *for all i, and each array chunk has the size* $\prod_{i=1}^{n} |C_i|$*, where* $|C_i| = c$ *for all i, the total amount of memory to compute the Cube of the array in one scan of A is less than* $c^n + (d + 1 + c)^{n-1}$*.*

Corollary 1 *For a chunked multidimensional array with the size* $\prod_{i=1}^{n} |D_i|$*, where* $|D_1| \leq |D_2| \cdots \leq |D_n|$*, and each array chunk has the size* $\prod_{i=1}^{n} |C_i|$*, where* $|C_i| = c$ *for all i, the total amount of memory to compute the Cube of the array in one scan is less than* $c^n + (d + 1 + c)^{n-1}$*, where* $d = (\prod_{i=1}^{n-1} |D_i|)^{1/(n-1)}$*.*

Note that this indicates that the bound is independent of the size of the largest dimension D_n. The single-pass multi-way algorithm assumes that we have the memory required by the MMST of the optimal dimension order. If we have this memory, all the group-bys can be computed recursively in a single scan of the input array (as described previously in the example for ABC). But if the memory is insufficient we need multiple passes. We need a multi-pass algorithm to handle this case, as described in the next section.

4.2 Multi-pass Multi-way Array Algorithm

Let $\mathcal{T}$ be the MMST for the optimal dimension ordering $\mathcal{O}$ and $M_{\mathcal{T}}$ be the memory required for $\mathcal{T}$, calculated using

Rule 2. If $M \leq M_{\mathcal{T}}$, we cannot allocate the required memory for some of the subtrees of the MMST. We call these subtrees "incomplete subtrees." We need to use some extra steps to compute the group-bys included in the incomplete subtrees.

The problem of allocating memory optimally to the different subtrees is similar to the one described in [AADN96] and is likely to be NP-hard. We use a heuristic of allocating memory to subtrees of the root from the right to left order. For example, in Figure 1, the order in which the subtrees are considered is BC, AC and then AB. We use this heuristic since BC will be the largest array and we want to avoid computing it in multiple passes. The multi-pass algorithm is listed below:

```
(1)     Create the MMST T for a dimension order O
(2)     Add T to the Tobecomputed list.
(3)     For each tree T' in Tobecomputed list
        {
(3.1)       Create the working subtree W and
                incomplete subtrees Is
(3.2)       Allocate memory to the subtrees
(3.3)       Scan the array chunk of the root of T'
                in the order O
            {
(3.3.1)         aggregate each chunk to the groupbys
                    in W
(3.3.2)         generate intermediate results for Is
(3.3.3)         write complete chunks of W to disk
(3.3.4)         write intermediate results to the
                    partitions of Is
            }
(3.4)       For each I
            {
(3.4.1)         generate the chunks from the
                    partitions of I
(3.4.2)         write the completed chunks of I to disk
(3.4.3)         Add I to Tobecomputed
            }
        }
```

The incomplete subtrees Is exist in the the case where $M \leq M_{\mathcal{T}}$. To compute the Cube for this case, we need multiple passes. We divide $\mathcal{T}$ into a working subtree and a set of incomplete subtrees. We allocate each node of the working subtree the memory required by it and finish aggregation for the group-bys contained in the working subtree during the scan of the array. For each incomplete subtree $D_{j_1}, D_{j_2}, .., D_{j_{n-1}}$, we allocate memory equal to a chunk size of the group-by $D_{j_1}, D_{j_2}, .., D_{j_{n-1}}$, aggregate each input array chunk to the group-by $D_{j_1}, D_{j_2}, .., D_{j_{n-1}}$ and write the intermediate result to disk. Each intermediate result is aggregation of the $D_{j_1}, D_{j_2}, .., D_{j_{n-1}}$ group-by for each chunk of $D_1, D_2, .., D_n$. But, each of the intermediate result is incomplete since the intermediate results for different $D_1, D_2, .., D_n$ chunks map to the same chunk of the $D_{j_1}, D_{j_2}, .., D_{j_{n-1}}$ group-by.

We need to aggregate these different chunks to produce one chunk of the $D_{j_1}, D_{j_2}, .., D_{j_{n-1}}$ group-by. It is possible that the amount of memory required by the $D_{j_1}, D_{j_2}, .., D_{j_{n-1}}$ group-by is larger than M. Therefore, we have to divide the chunks of the $D_{j_1}, D_{j_2}, .., D_{j_{n-1}}$ group-by into partitions according to the dimension order so that the chunks in each partition fit in memory. When we output the intermediate chunks of $D_{j_1}, D_{j_2}, .., D_{j_{n-1}}$, we write them to the partition to which they belong to. For example, the partition may be decided by the values of $D_{j_{n-1}}$ in the chunk. Different ranges of values of $D_{j_{n-1}}$ will go to different partitions. In step (3.4.1), for each partition, we read each intermediate result and aggregate them to the corresponding chunk of the $D_{j_1}, D_{j_2}, .., D_{j_{n-1}}$ group-by. After we finish processing each intermediate result, each chunk of the $D_{j_1}, D_{j_2}, .., D_{j_{n-1}}$ group-by in memory is complete and we output them in the dimension order $D_{j_1}, D_{j_2}, .., D_{j_{n-1}}$. Once we are done for each partition, we complete the computation for the group-by $D_{j_1}, D_{j_2}, .., D_{j_{n-1}}$. To compute the subtrees of the $D_{j_1}, D_{j_2}, .., D_{j_{n-1}}$ node, we repeat loop 3 until we finish the aggregation for each node of the subtree.

5 Performance Results

In this section, we present the performance results of our MOLAP Cube algorithm and a previously published ROLAP algorithm. All experiments were run on a Sun SPARC 10 machine running SunOS 3.4. The workstation has a 32 MB memory and a 1 GB local disk with a sequential read speed 2.5 MB/second. The implementation uses the unix file system provided by the OS.

5.1 Data Sets

We used synthetic data sets to study the algorithms' performance. There are a number of factors that affect the performance of a cubing algorithm. These include:

- Number of valid data entries.

  That is, what fraction of the cells in a multidimensional space actually contain valid data? Note that the number of valid data entries is just the number of tuples in a ROLAP table implementing the multidimensional data set.

- Dimension size.

  That is, how many elements are there in each dimension? Note that for a MOLAP array implementation, the dimension size determines the size of the array. For a ROLAP implementation, the table size remains constant as we vary dimension size, but the range from which the values in the dimension attributes are drawn changes.

- Number of dimensions.

  This is obvious; here we just mention that by keeping the number of valid data cells constant, varying the number of dimensions impacts ROLAP and MOLAP implementations differently. Adding dimensions on MOLAP causes the shape of the array to change; adding dimensions in ROLAP adds or subtracts attributes from the tuples in the table.

Since the data density, number of the array dimensions, and the array size affect the algorithm performance, we designed three data sets.

Data Set 1: Keep the number of valid data elements constant, vary the dimension sizes. The data set consists of three 4-dimension arrays. For those arrays, three of the four dimensions sizes are fixed at 40, while the fourth dimension is either 40 (for the first array), or 100 (for the second), or 1000 (for the third). Every array has the 640000 valid elements. This results in the data density of the arrays (fraction of valid cells) ranging from 25%, to 10%, to 1%. The size of the input compressed array for the Array method turned out to be 5.1MB. The input table size for the ROLAP method was 12.85MB.

Data Set 2: Keep dimension sizes fixed, vary number of valid data elements.

All members of this data set are logically 4-dimensional arrays, with size 40x40x40x100. We varied the number of valid data elements so that the array data density ranges from 1% to 40%. The input compressed array size varied from 0.5MB, to 5.1MB, to 12.2MB, to 19.9MB. The corresponding table sizes for the ROLAP tables were 1.28MB, 12.8MB, 32.1MB, 51.2MB.

Data Set 3: this data set contains three arrays, with the number of dimensions ranging from 3, to 4, to 5. Our goal was to keep the density and number of valid cells constant throughout the data set, so the arrays have the following sizes: $40 \times 400 \times 4000$, $40 \times 40 \times 40 \times 1000$, and $10 \times 40 \times 40 \times 40 \times 100$. For each array, it has the same data density 1%. Hence, each array has 640000 valid array cells. The size of the input array was 5.1MB. The table size for ROLAP changed from 10.2MB, to 12.8MB, to 15.6MB, due to added attributes in the tuples.

We generated uniform data for all three data sets. Since these data sets are small, we used a proportionately small buffer pool, 0.5 MB, for most experiments. We will indicate the available memory size for those tests not using the same memory size.

5.2 Array-Based Cube Algorithms

In this section, we compare the naive and the *Multi-way* Array algorithms, study the effect of the compression algorithm to the performance of the *Multi-way* algorithm, investigate its behaviour as the buffer pool size decreases, and test its scale up as the number of dimensions increases.

5.2.1 Naive vs. Multi-way Array Algorithm

We ran the tests for the naive and the *Multi-way* Array algorithm on three 4-dimension arrays. Three of the four dimension sizes are fixed at 40, while the fourth dimension is varied from 100, to 200, to 300. Each array has the same data density 10%. In Figure 5, we see that the naive array algorithm is more than 40% slower than the *Multi-way* Array algorithm, due to multiple scans of the parent group-bys.

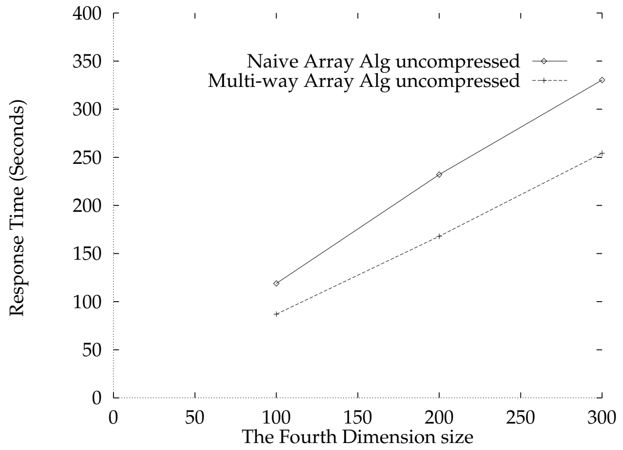

Figure 5: Naive vs. Multi-way Array Alg.

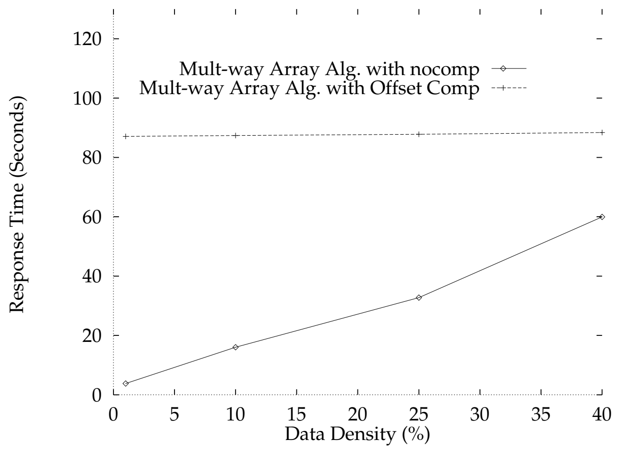

Figure 6: Two Compression Methods

5.2.2 Compression Performance

In Figure 6, we compare the array with no compression to the array with offset compression for **Data Set 2**. It shows that for data density less than 40% the *Multi-way* Array algorithm performed on the input array compressed by the offset algorithm is much faster than on uncompressed input array. There are two reasons for this. At lower densities, the compressed array size is much smaller. Hence, it reduces the I/O cost for reading the input array. The other is that the *Multi-way* Array algorithm only processes the valid array cells of the input array during computing the data Cube if the input array is compressed by the offset algorithm. For the uncompressed input array the *Multi-way* Array algorithm has to handle invalid array cells as well.

5.2.3 The Multi-way Array with Different Buffer Sizes

We ran experiments for **Data Set 2** at 10% density by varying the buffer pool size. In Figure 7, we see that the performance of *Multi-way* Array algorithm becomes a step function of the available memory size. In this test, we increased the available memory size from 52 KB to 0.5 MB. The first step on the right is caused by generating two incomplete subtrees in the first scan of the input array due to insufficient memory to hold the required chunks for the two subtrees. The algorithm goes through the second pass to produce each incomplete subtree and computes the group-bys contained in the two subtrees. As the available memory size increases to 300KB, only one incomplete subtree is generated, which causes the second step on the right. With the available memory more than 400 KB, the algorithm allocates memory to the entire MMST and computes the Cube in one scan of the input array. We flushed the OS cache before we process each working subtrees from their partitions. Theorem 2 predicts a bound of 570KB for the memory required for this data. The graph shows that above 420KB the entire MMST fits in memory. Thus the bound is quite close to the actual value.

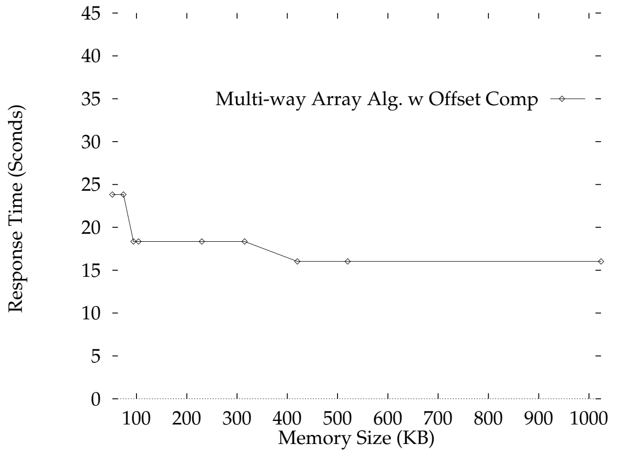

Figure 7: Multi-way Array Alg. with Various Memory Size

5.2.4 Varying Number of Dimensions

We discuss varying the number of dimensions when we compare the array algorithm with the ROLAP algorithm below.

5.3 The ROLAP vs. the Multi-Way Array Algorithms

In this section, we investigate the performance of our MOLAP algorithm with a previously published sort-based ROLAP algorithm in three cases. We used the *Overlap* method from [AADN96] as a benchmark for this comparison. In ROLAP the data is stored as tables. Computing the cube on a table produces a set of result tables representing the group-bys. On the other hand, in MOLAP data is stored as sparse multidimensional arrays. The cube of an array will produce a array for each of the group-bys. Since there are different formats (Table and Array) possible for the input and output data, there could be several ways of comparing the two methods. These are described in the following sections.

5.3.1 Tables vs. Arrays

One way to compare the array vs. table-based algorithms is to examine how they could be expected to perform in their "native" systems. That is, we consider how the multi-way array algorithm performs in a system that stores its data in array format, and how the table based algorithm performs in a system that stores its data in tables.

One might argue that arrays already order the data in such a way as to facilitate cube computation, whereas tables may not do so. Accordingly, in our tests we began with the table already sorted in the order desired by the table-based algorithm. This is perhaps slightly unfair to the array-based algorithm, since unless the table is stored in this specific order, the table based cube algorithm will begin with a large sort.

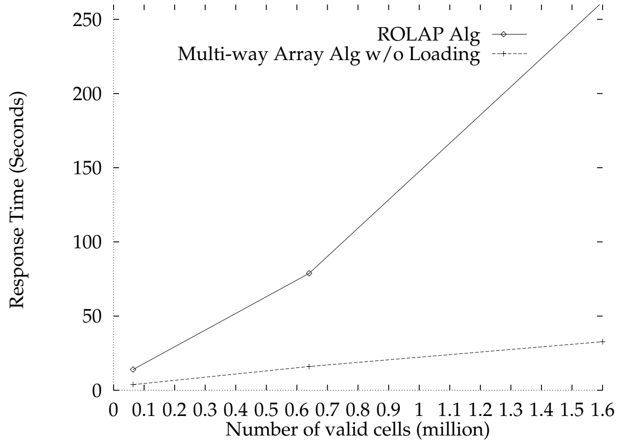

Figure 8: ROLAP vs. Multi-way Array for Data Set 2

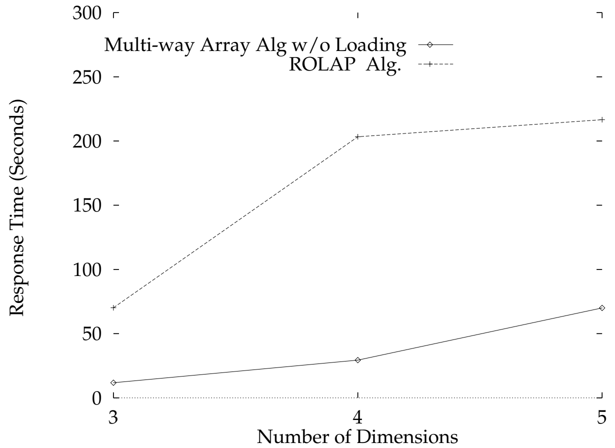

Figure 9: ROLAP vs. Multi-way Array for Data Set 3

The graphs in Figures 8, 9 and 10 compare the two methods for **Data Sets 2, 3** and **1**. For **Data Set 2**, as the density increases, the size of the input table increases. This also leads to bigger group-bys, i.e. the result table sizes also increase. The ROLAP method will need more memory due to this increase in size. Since the memory is kept constant at 0.5M, the ROLAP method has to do multiple passes and thus the performance becomes progressively worse. (As we shall see below, it is the growing CPU cost due to these multiple passes that dominates rather than the I/O cost.) For the array method, the array dimension sizes are not changing. Since the memory requirement for a single pass computation for the array method depends only on the dimension sizes, and not on the number of valid data cells, in

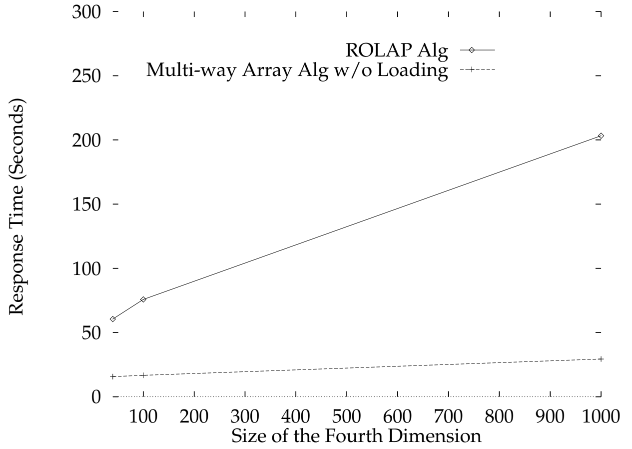

Figure 10: ROLAP vs. Multi-way Array w/o Loading for Data Set 1

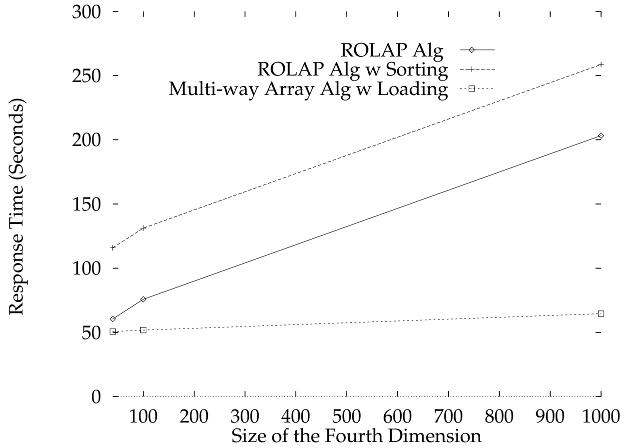

Figure 11: ROLAP vs. Multi-way Array with Loading for Data Set 1

all cases the array method can finish the computation in one pass. Thus we see a smaller increase in the time required for the array method.

Similarly for **Data Set 1**, as the size of the fourth dimension is increased, the sizes of the group-bys containing the fourth dimension in the ROLAP computation grow. Though the input table size is constant, the increase in the size of the group-bys leads to greater memory requirements. But due to the memory available being constant at 0.5MB, once again the ROLAP method reverts to multiple passes and its performance suffers. Turning to the array algorithm, the array sizes also increase due to increase in the size of the fourth dimension. But in the optimal dimension order, as given by Theorem 4.1.4, the biggest (fourth) dimension is kept last. Furthermore, by Corollary 4.1.4, the size of this last dimension does not affect the memory required fo ra single pass computation. Thus the memory requirements of the array algorithm remains constant at 0.5MB and it always computes everything in one pass. Thus the running time of the array method does not increase significantly.

In **Data Set 3**, we vary the number of dimensions from 3 to 5. The number of group-bys to be computed is exponential in the number of dimensions. Since both algorithms compute all of these group-bys, the running time of both the methods increases with the number of dimensions.

5.3.2 The MOLAP Algorithm for ROLAP Systems

Although it was designed for MOLAP systems, the array method could also be applied to any ROLAP system. Since the array method is much faster than the table method, it might be viable to convert the input table first into an array, cube the array, and then convert back the resulting arrays into tables. In this approach, rather than being used as a persistent storage structure, the array is used as a query evaluation data structure, like a hash table in a join. We did two experiments to study the performance of this approach.

In the first comparison, the *Multi-way* Array method loads data from an input table into an array as a first step. The ROLAP method just computes the cube from the input table as in the previous case. The input table is unsorted, so the ROLAP method has to specifically sort the input. The times for **Data Set 1** are shown in Figure 11. It can be seen that the array method with loading is much faster than the ROLAP method. We then repeated the experiments with a sorted input table for the ROLAP method, so that the initial sorting step can be avoided. The times are shown in the same graph. It turns out that even in this case, the *Multi-way* Array method turns out to be faster.

5.4 Drilling Down on Performance

In this section, we try to explain why the *Multi-way* Array method performs much better than the ROLAP method. Our experiments showed for the ROLAP method, about 70% of the time is spent on CPU computations and the remaining is I/O. The ROLAP method reads and writes data into tables. These table sizes are significantly bigger than the compressed arrays used by the *Multi-way* Array method. Thus the ROLAP method reads and writes more data, meaning that the 30% of the running time due to I/O dominates the I/O time used in the MOLAP algorithm.

Turning to the CPU usage, on profiling the code we found that a significant percentage of time (about 55-60%) is spent in sorting intermediate results while about (10-12%) time is spent in copying data. These sorts are costly, largely due to a large number of tuple comparisons. Tuple comparisons incur a lot of cost, since there are multiple fields to be compared. The copying arises because the ROLAP method has to copy data to generate the result tuples. This copying is also expensive since the tuples are bigger than the array cells used in the MOLAP algorithm.

On the other hand, the *Multi-way* Array method is a position based method. Different cells of the array are aggregated together based on their position, without incurring the cost of multiple sorts (the multidimensional nature of the array captures the relationships among all the dimensions.) Thus once an array has been built, computing different group-bys from it incurs very little cost. One potential problem with the array could be sparsity, since the array size will grow as the data becomes sparse. However, we found that the offset compression method is very effective. It not only compresses the array, but different compressed chunks can be directly aggregated without having to decompress them. This leads to much better performance for the array. It turns out that the *Multi-way* Array method is even more CPU intensive than the ROLAP algorithm (about 88% CPU time). Most of this time (about 70%) is spent in doing the aggregation, while 10% is spent in converting the offset to the index values while processing the compressed chunks.

6 Conclusion

In this paper we presented the Multi-Way Array based method for cube computation. This method overlaps the computation of different group-bys, while using minimal memory for each group-by. We have proven that the dimension order used by the algorithm minimizes the total memory requirement for the algorithm.

Our performance results show that that the Multi-Way Array method performs much better than previously published ROLAP algorithms. In fact, we found that the performance benefits of the Multi-Way Array method are so substantial that in our tests it was faster to load an array from a table, cube the array, then dump the cubed array into tables, than it was to cube the table directly. This suggests that this algorithm could be valuable in ROLAP as well as MOLAP systems — that is, it is not necessary that the system support arrays as a persistent storage type in order to obtain performance benefits from this algorithm.

References

[AADN96] S. Agarwal, R. Agrawal, P. Deshpande, J. Naughton, S. Sarawagi and R. Ramakrishnan. "On the Computation of Multidimensional Aggregates". In *Proceedings of the 22nd International Conference on Very Large Databases*, Mumbai (Bombay), 1996.

[AS] Arbor Software. "The Role of the Multidimensional Database in a Data Warehousing Solution". White Paper, Arbor Software. http://www.arborsoft.com/papers/wareTOC.html

[CCS93] E.F. Codd, S.B. Codd, and C.T. Salley. "Providing OLAP (On-line Analytical Processing) to User-Analysts: An IT Mandate", White Paper, E.F. Codd and Associates. http://www.arborsoft.com/papers/coddTOC.html

[DKOS84] D. Dewitt, R. Katz, G. Olken, L. Shapiro, M. Stonebraker, D. Wood. "Implementation Techniques for Main Memory Database Systems". In Proceedings of SIGMOD, Boston, 1984.

[GBLP95] J. Gray, A. Bosworth, A.Layman, and H.Pirahesh. "Data Cube: A relational aggregation operator generalizing group-by, cross-tabs and sub-totals. Technical Report MSR-TR-95-22, Microsoft Research, Advance Technology Division, Microsoft Corporation, Redmond, 1995.

[GC96] G. Colliad. "OLAP, Relational, and Multidimensional Database Systems". SIGMOD Record, Vol. 25. No. 3, September 1996.

[IA] Information Advantage. "OLAP - Scaling to the Masses". White Paper, Information Advantage. http://www.infoadvan.com/

[MC] Stanford Technology Group, Inc. "INFORMIX-MetaCube". Product Brochure. http://www.informix.com/informix/products/new_plo/stgbroch/brochure.html

[MS] MicroStrategy Incorporated. "The Case For Relational OLAP". White Paper, MicroStrategy Incorporated. http://www.strategy.com/dwf/wp_b_al.html

[OC] Oracle Corporation. "Oracle OLAP Products". White Paper, Oracle Corporation. http://www.oracle.com/products/collatrl/olapwp.pdf

[PSW] Pilot Software. "An Introduction to OLAP". White Paper, Pilot Software. http://www.pilotsw.com/r_and_t/whtpaper/olap/olap.htm

[RJ] Arbor Software Corporation, Robert J. Earle, U.S.Patent # 5359724

[SM94] Sunita Sarawagi, Michael Stonebraker, "Efficient Organization of Large Multidimensional Arrays". In *Proceedings of the Eleventh International Conference on Data Engineering*, Houston, TX, February 1994.

[Wel84] T. A. Welch. "A Technique for High-Performance Data Compression". IEEE Computer, 17(6), 1984.

[ZTN] Y.H. Zhao, K. Tufte, and J.F. Naughton. "On the Performance of an Array-Based ADT for OLAP Workloads". Technical Report CS-TR-96-1313, University of Wisconsin-Madison, CS Department, May 1996.