OLAP technology allows a user to extract and visualize information from a data set in an easy and intuitive way, from different perspectives. Usually, this technology is used in decision support system for report production on budgeting, previews, financial issues and similar goals. The ability to see data from different perspectives is due to the fact that OLAP operations use a multidimensional model to build an “image” of data, known as cube. The generation of cubes requires numerous multidimensional aggregates to be computed, which is a task with high needs of time and space. These high requisites motivated the development of various algorithms, like Multi-Way algorithm. This algorithm was one of the few developed for MOLAP systems and shows good performances when compared with algorithms that follow other approaches. The goal of this thesis is to study that algorithm and develop optimizations that would allow it to solve situations it is not able to hand as it was conceived, due to lack of memory.

1 Introduction

During the last decades, our ability to generate, collect and maintain data has significantly increased. With the growing research on information technologies area and database systems since the 70 years and the decreasing price of hardware resources, different organizations started keeping data on their events, activities and clients, such as inventories, sales records or market indicators. This data can be turned into a competitive advantage as long as it is carefully analyzed and mined for useful information. Soon the available data exceeded the human capacity of analysis, which led to a data rich but information poor situation [Inmon1996] because the decisions weren’t really based on the hidden information in the data. This situation motivated the creation of different techniques and tools to aid to individuals extract information from data.

Since the 80 years, the information technologies area registered great development due to the use of the relational model and the intensive research on data models and databases. The advent of Internet also stimulated this development because it introduced new research topics, like distribution, data sharing and heterogeneity in databases. Consequently, sophisticated decision support systems were created, allowing organizations to make the decision making process based on the available data more efficient. The expression data warehousing identifies a group of decision support technologies that allow a user to analyze hidden information in a data set. The operational databases differ from data warehouses because they are structured to optimize the daily use, while data warehouses maintain historical records that consolidate data from different sources. Data warehouses can be implemented over the relational database management systems, using SQL extensions. There is also another kind of data warehouses which implements the multidimensional data model.

In fact, data warehouses and the multidimensional model are essential to decision support tools as they influence its performance and the quality of its results. Cubes allow the user to analyze data in a flexible and intuitive way and there are different techniques to compute multidimensional aggregates, each more suited to a certain category of problems. The algorithms for hypercube computation must
deal not only with the complexity associated to the aggregate computation but also with the data characteristics, which are usually sparse in real life and can affect an algorithm’s performance. In this domain, implementation can make the difference.

Due to the fact that the Multi-Way algorithm shows promising behavior, this extended abstracts summarizes the results of a deep study done on this algorithm and a comparison with a variant of this algorithm used in DBMiner mining system, to identify its limitations and subsequently propose and evaluate optimization to improve its performance. This document is structured as follows: section 2 presents briefly Multi-Way algorithm and its variant used on DBMiner system; section 3 presents the proposed optimizations; section 4 shows the results of the tests with datasets with different characteristics; section 5 summarizes the conclusions of this work.

2 Multi-Way algorithm

In MOLAP systems, data is stored in arrays and techniques as sorting and hashing aren’t applicable. The fact of using another representation for data makes it necessary to deal with different issues, such as efficient loading and storing of large and sparse arrays. Multi-way array cubing was proposed by Zhao et al. and follows the principles of MOLAP paradigm. This algorithm’s goal is to scan the arrays cells in such a way that it won’t be necessary to repeat the scan to calculate each of the sub-aggregates. Due to performance issues, the arrays have to be stored divided in smaller arrays, using for that a strategy called chunking. Chunking is a way of dividing n-dimensional arrays in several smaller n-dimensional arrays (chunks) that can be stored in disk as distinct objects [Zhao1997]. Although, it’s not uncommon that some chunk cells are empty, which means that there is no data for that combination of coordinates. Zhao et al. [Zhao1997] say that a chunk is considered dense when more than 40% of the cells contain a valid value. When this situation doesn’t occur, the chunk is said sparse and a compression technique must be employed to assure no empty cells are stored. This technique rises the probability of having chunks with different sizes, which can be overcome keeping metadata in the beginning of the chunk, for example.

Using chunking ensures efficiency on loading and storing the values on the cube cells, while performance is assured by calculating the aggregates in the correct order. For that, the algorithm proposes the concepts of optimal dimension order and minimum memory spanning tree. The dimension order in a chunk is represented as $O = (D_{j1}, D_{j2}, ..., D_{jn})$, assuming that there are n dimensions $D_1, D_2, ..., D_n$. Different dimension ordering implies different chunk reading orders and determines the amount of memory needed. To prove the importance of the dimension order, we will consider a 3D data array with three dimensions (A, B and C), divided in 64 chunks as in figure 9. The cardinality of each dimension A, B and C is, respectively, 40, 400 and 4000.
Materializing this cube is equivalent to compute all the cuboids:

- The base cuboid (ABC), which corresponds to the 3D array in figure 1
- The 2D cuboids (AB, AC and BC), which correspond respectively to group-bys AB, AC and BC and have to be computed
- The 1D cuboids (A, B and C), which correspond to group-bys A, B and C and have to be computed
- The 0D cuboid (all), which corresponds to the group-by () and has to be computed

There are several possible orderings to read the chunks in memory. Assuming they are read in the order shown in figure 9, from chunk 1 to chunk 64, one of the larger chunks in the 2D plan (BC, which size is given by 400x4000=400000). That means b0c0 is completely aggregated after chunks 1 to 4 have been read, b1c0 is completely aggregated after chunks 5 to 8 have been read, and so on. On the other side, to calculate a chunk from the second larger 2D plan (AC, which size is given by 40x4000=40000) 13 chunks have to be read. For instance, a0c0 is only fully aggregated when chunks 1, 5, 9 and 13 have been read. Finally, computing a chunk on the smaller 2D plan (AB, which size is given by 40x400=4000) means 49 chunks have to be read; for instance, a0b0 is only completely aggregated after chunks 1, 17, 33 and 49 have been read. Using this dimension ordering, the smaller plan is the one which requires the larger number of read operations to be computed. The minimum amount of memory needed to calculate a cube in this way is 40x400 (AB plan) + 40x1000 (one line from AC plan) + 100x1000 (a chunk from BC plan) = 16000 + 40000 + 100000 = 156000.

Supposing that the chunks are read in such a way that aggregation is done first on AB plan, then in AC plan and finally in BC plan, the minimum memory requirements are higher than when using the previous dimension ordering. Reading chunks 1, 17, 33, 49, 5, 21, 37, 53 and so on, the minimum amount of memory needed is 400 x 400 (BC plan) + 40 x 1000 (one line from AC plan) + 10 x 100 (one chunk from AB plan) = 160000 + 40000 + 1000 = 1641000.

This shows that the order by which the dimensions are taken is decisive for the memory needs. The minimum memory spanning tree (MMST) is the minimum spanning tree that expresses the total memory needs for a determined dimension order. Formally, a minimum memory spanning tree for a cube (D_1, D_2, ..., D_n) which dimensions are ordered according to O = (D_1, D_2, ..., D_n) has n + 1 levels,
with root (D₁, D₂, ..., Dₙ) at level n. So, different dimensions orderings generate different spanning trees, with different memory requirements.

Zhao et al [4] proved that the optimal dimension ordering is the one that generates a MMST with the least total need of memory. Formally, the optimal dimension order is O = (D₁, D₂, ..., Dₙ) with |D₁| ≤ |D₂| ≤ ... ≤ |Dₙ| and |Dᵢ| representing the size of dimension Di.

After obtaining the minimum spanning tree for a cube with a dimension ordering O, it’s possible to increase the efficiency of the computation calculating the related group-bys when data is being read, assuming there is enough memory to allocate all the necessary tree nodes. Using the cube in figure 1, consider a 16x16x16 with 4x4x4 chunk, read according to O = (A, B, C). To calculate BC, it’s necessary to store 1 chunk of BC; to calculate AC, it’s necessary memory to store 4 chunks of AC; to calculate AB, it’s necessary memory to store 16 chunks of AB. Generically, one has to allocate |Bᵢ| |Cᵢ|u memory to calculate a BC group-by, |Aᵢ||Cᵢ|u to calculate a AC group-by and |Aᵢ||Bᵢ|u to calculate a AB group-by, having |Xᵢ| as the size of dimension X, |Yᵢ| as the size of a dimension Y chunk and u as the size of each chunk element [4]. Usually |Cᵢ| is inferior to |Dᵢ| in most of the dimensions, what leads to the conclusion that it is possible to calculate a cube allocating less memory than the one needed by a group-by and therefore calculate more group-bys simultaneously. For instance, to calculate a XY group-by that contains a prefix of ABC with size p, 16p x 42-p x u memory has to be allocated since each dimension has size 16 and each chunk has size 4.

Zhao et al [Zhao1997] formalized these calculations for group-bys of n-dimensional arrays in a rule. According to that rule, to calculate a group-by (Dᵢ₁, Dᵢ₂, ..., Dᵢₚ) from array (D₁, D₂, ..., Dₙ) read in order O = (D₁, D₂, ..., Dₙ) that contains a prefix of (D₁, D₂, ..., Dₚ) of size p (0 ≤ p ≤ n – 1),

\[
\prod_{i=1}^{p} |Dᵢ|^X \prod_{i=p+1}^{n} |Cᵢ|
\]

units of array have to be allocated, with |Dᵢ| being the dimension size and |Cᵢ| the chunk size for dimension i.

Although this algorithm is characteristic of MOLAP application, it can be used by ROLAP systems since it scans the data table, loads it to an array, computes the results over that array and transfers the results to the appropriate tables. This usage is explained by the high performance this algorithm shows and the efficient memory management it does, being even more efficient than the algorithms designed for ROLAP systems. This algorithm was implemented by Tam [Tam1998] during an investigation on algorithms for computing cubes in OLAP systems. Motivated by the CUBE operator [Gray1996] and the array-based algorithm [Zhao1997], Tam designed a chunk-based compressed algorithm using the MOLAP approach with some relational features and an aggregation computation algorithm (R-cubing). That algorithm was able to scan the relation only once, compute as many group-bys as possible together in memory from the smallest computed group-by and save the group-bys no longer needed to free memory for computation.
3 Multi-Way algorithm implementation

Multi-Way algorithm was proposed by Zhao et al [Zhao1997] to address the issue of the lack of algorithms specially designed for MOLAP systems. The characteristics of this kind of systems eliminate the need of reordering attributes and perform groupings such that the first aggregates that are computed can be used to compute other aggregates. The algorithm can achieve this running the dimensions values in the most efficient way possible and computing simultaneously the maximum possible number of partial aggregates. The main obstacle to this approach is the way the arrays are managed, as it is necessary to load and store arrays which size probably exceeds the amount of available memory. To address these issues, the authors proposed the use of chunking, a strategy that divides a n-dimensional array in smaller arrays that can be stored as a unique object.

The main principles of this algorithm – optimal order, chunking and MMST – were used by Tam to propose and implement another algorithm by Tam [Tam1998], which doesn't follow exactly what was proposed by Zhao et al but is still a valid example of a possible implementation based on those principles. Besides the good results obtained by both implementations, the algorithm still shows limitations as it can only compute small to medium cubes (with less than 10 dimensions) [Tam1998]. As there are no others works directed towards MOLAP systems besides the author’s work and Tam’s thesis and it shows good results, this work’s goal is to study the algorithm as proposed by Zhao et al and try to introduce in that implementation optimizations that make the algorithm more effective in a larger number of situations.

3.1 “Sub-treeing” optimization

The notion of minimum memory spanning tree (MMST) for a specific dimension order is central in the algorithm as it assures that only is allocated the minimum memory needed for that dimension order. The goal of this optimization is to improve the flow of the algorithm when it pretends to compute a specific aggregate, avoiding the need to recalculate every time an OLAP operation is applied. This way, when the objective is to compute a specific aggregate, there's no need to compute the aggregates in every tree node but only the ones really needed for our objective. This means that there's no need to compute the whole tree but only a part of it (sub-tree).

3.2 “Sub-chunking” optimization

One of the main obstacles to this algorithm is the amount of memory needed to compute the aggregates. When reporting the results of its algorithm, Tam points out the fact that her algorithm couldn't compute the cube for datasets with more than 240000 tuples with 6 dimensions [Tam1998]. This optimization proposes that, when the available memory limit is exceeded, the algorithm creates a smaller sub-chunk using the maximum common divisor (MDC) of the dimensions cardinalities. This allows the algorithm to transfer to persistent memory each of the sub-chunks to assure the computation isn't affected and to retrieve each sub-chunk when it is needed. Obviously, this strategy
increases the number of I/O operations and that leads to a decrease in performance, but this way the calculations can be performed. Using the MDC assures the creation of the larger possible block and minimizes the I/O operations needed. When this strategy isn’t enough to perform the calculation, a new sub-chunk whose size is equal to the result of dividing the maximum common divisor (MDC) and the minimum common divisor (mDC), is created. In worst case, each new sub-chunk will correspond to one memory block, once again assuring calculations can be done with a small cost on the algorithm’s performance.

4 Results

The datasets used in these tests were generated using Illimine project from Illinois University, supervised by Professor Jiawei Han. Tests were designed to evaluate the influence of the density and volume of data on the algorithm’s performance and I/O operations. In each test, the algorithm should compute the ALL aggregate.

<table>
<thead>
<tr>
<th>Test</th>
<th>Transactions</th>
<th>Dimensions</th>
<th>Cardinalities</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>60000</td>
<td>4</td>
<td>10 20 30 40</td>
</tr>
<tr>
<td>T2</td>
<td>120000</td>
<td>4</td>
<td>10 20 30 40</td>
</tr>
<tr>
<td>T3</td>
<td>180000</td>
<td>4</td>
<td>10 20 30 40</td>
</tr>
<tr>
<td>T4</td>
<td>240000</td>
<td>4</td>
<td>10 20 30 40</td>
</tr>
<tr>
<td>T5</td>
<td>5000</td>
<td>3</td>
<td>10 10 10</td>
</tr>
<tr>
<td>T6</td>
<td>5000</td>
<td>4</td>
<td>10 10 10 10</td>
</tr>
<tr>
<td>T7</td>
<td>5000</td>
<td>5</td>
<td>10 10 10 10 10</td>
</tr>
<tr>
<td>T8</td>
<td>5000</td>
<td>3</td>
<td>10 10 10</td>
</tr>
<tr>
<td>T9</td>
<td>5000</td>
<td>3</td>
<td>20 20 20</td>
</tr>
<tr>
<td>T10</td>
<td>5000</td>
<td>3</td>
<td>30 30 30</td>
</tr>
<tr>
<td>T11</td>
<td>5000</td>
<td>3</td>
<td>40 40 40</td>
</tr>
<tr>
<td>T12</td>
<td>5000</td>
<td>3</td>
<td>50 50 50</td>
</tr>
<tr>
<td>T13</td>
<td>5000</td>
<td>3</td>
<td>60 60 60</td>
</tr>
<tr>
<td>T14</td>
<td>5000</td>
<td>3</td>
<td>70 70 70</td>
</tr>
<tr>
<td>T15</td>
<td>5000</td>
<td>3</td>
<td>80 80 80</td>
</tr>
<tr>
<td>T16</td>
<td>5000</td>
<td>3</td>
<td>90 90 90</td>
</tr>
</tbody>
</table>

4.1 Influence of data density

The data density doesn’t influence the number of I/O operations, which may seem confusing but is just a consequence of the algorithm’s mechanics. The algorithm scans all dimensions, starting with the one with higher cardinality, and visits all positions because he is not aware of its content. That way, the algorithm does the same number of I/O operations for two datasets as long as they have the same number of dimensions and equal product of dimensions cardinalities.
4.2 Influence of data volume

Two datasets have different data volumes when they have different number of dimensions or, having the same number of dimensions, when the dimensions cardinalities are different.

4.2.1 Variation on the number of dimensions

As shown in figure 3, the number of I/O operations increases when the number of dimensions in data increase. The number of I/O operations is always lower for the implementation that uses sub-treeing (MW+ST) and higher for the implementation that uses sub-chunking (MW+SC), which can be explained by the subdivision of the arrays and subsequent higher number of intermediate read and write operations. When both optimizations are used, algorithm performs better than the original but slightly worse than the implementation that only uses sub-treeing.
Figure 3 – Effect of varying the number of dimensions on the number of I/O operations

Figure 4 shows the time spent by the different implementations to perform the computation and it mirrors the variations on the I/O operations on the algorithm.

4.2.2 Variation on the dimensions cardinalities

When the dataset comprises dimensions with higher cardinalities, the number of I/O operations also increases. Figure 5 shows this fact and enforces the previous conclusions, showing that the implementation that only uses sub-treeing performs less I/O operations than the implementation that only uses sub-chunking. Using both optimizations results in a number of I/O operations close to the one of the original implementation but, when the dimensions cardinalities increase, results in a higher number of I/O operations when in comparison with the other implementations.
These results lead to the question of the real usefulness of the proposed optimizations, considering that the first visible effect is the increase in I/O operations and the decrease in performance. The graphic in figure 6 answers this question, showing that for a dataset with 2560000 tuples and 4 dimensions only the implementation that uses sub-chunking (MW+SC) and the implementation that uses both optimizations (MW+ST+SC) can complete the computation.
5 Conclusions

This work’s goal was to study Multi-Way algorithm and evaluate the impact of the proposed optimizations. Unfortunately, there weren’t other implementations available as a way to make comparisons, so this work was entirely based on the implementation made following Zhao’s proposal and Tam’s variant, documented in her master thesis. Two optimizations were proposed: sub-treeing allows the algorithm to explore a lesser number of MMST nodes and increases the algorithm’s performance; sub-chunking is based on the chunking proposed by the authors and yields to a higher number of I/O operations and a decrease in performance. Using both optimizations doesn’t improve the algorithm’s performance but makes it possible to perform calculations in situations where the original implementation can’t obtain results due to lack of memory.

6 References


Figure 6 – Time spent by each implementation to compute a cube for a dataset with 2560000 tuples