Implementing Data Cube Construction Using a Cluster Middleware: Algorithms, Implementation Experience, and Performance Evaluation *

Ge Yang    Ruoming Jin    Gagan Agrawal
Department of Computer and Information Sciences
Ohio State University, Columbus OH 43210
{yangg,jinr,agrawal}@cis.ohio-state.edu

Abstract

With increases in the amount of data available for analysis in commercial settings, On Line Analytical Processing (OLAP) and decision support have become important applications for high performance computing. Implementing such applications on clusters requires a lot of expertise and effort, particularly because of the sizes of input and output datasets.

In this paper, we describe our experiences in developing one such application using a cluster middleware, called ADR. We focus on the problem of data cube construction, which commonly arises in multi-dimensional OLAP. We show how ADR, originally developed for scientific data intensive applications, can be used for carrying out an efficient and scalable data cube construction implementation. A particular issue with the use of ADR is tiling of output datasets. We present new algorithms that combine inter-processor communication and tiling within each processor. These algorithms preserve the important properties that are desirable from any parallel data cube construction algorithm.

We have carried out a detailed evaluation of our implementation. The main results from our experiments are as follows: 1) High speedups are achieved on both dense and sparse datasets, even though we have used simple algorithms that sequentialize a part of the computation, 2) The execution time depends only upon the amount of computation, and does not increase in a super-linear fashion as the dataset size or the number of tiles increases, and 3) As the datasets become more sparse, sequential performance degrades, but the parallel speedups are still quite good.

1 Introduction

Analysis on large datasets is increasingly guiding business decisions. Retail chains, insurance companies, and telecommunication companies have created very large datasets for their decision support systems. A system storing and managing such datasets is typically referred to as a data warehouse and the analysis performed is referred to as On Line Analytical Processing (OLAP).

Computing multiple related group-bys and aggregates is one of the core operations in OLAP applications. Jim Gray has proposed the cube operator, which computes group-by aggregations over all possible subsets of the specified dimensions [9]. When datasets are stored as (possibly sparse) arrays, data cube construction involves computing aggregates for all values across all possible subsets of dimensions. Developing sequential algorithms for constructing data cubes is a well-studied problem [2, 12].

Data cube construction is a compute and data intensive problem. Therefore, it is natural to use parallel computers for data cube construction. There is a limited body of work on parallel data cube construction [7, 8]. Because of the sizes of datasets involved, managing input and output data is critical for achieving high performance for data cube construction. Unfortunately, this significantly increases the complexity of a parallel implementation.

In this paper, we explore the use of a cluster middleware for carrying out a rapid, yet efficient and scalable, implementation of data cube construction. Specifically, we use Active Data Repository (ADR) [3, 4, 5], which was originally developed for scientific data-intensive applications, for implementing data cube construction algorithms. ADR integrates storage, retrieval, and processing of multidimensional datasets on a parallel platform. Because of ADR’s focus on multidimensional datasets and on applications that involve associative and commutative reduction operations, it appears like a good target for implementing data cube construction.

One of the key aspects of ADR is partitioning of output datasets into tiles. For each output dataset, only one tile is allocated in the main memory at a time. All the tiles allocated at any given time must fit in the main memory. This eliminates the use of virtual memory and achieves higher performance. To exploit this aspect of ADR, we have developed new parallel algorithms for data cube construction.

* This work was supported by NSF grant ACR-9982087, NSF CAREER award ACI-9733520, and NSF grant ACR-0130437.
These algorithms combine interprocessor communication and tiling on each processor. These algorithms preserve the important properties that are desirable from any parallel data cube construction algorithm, while matching the model of computation expected by ADR.

This paper describes these algorithms and our experience in implementing them using ADR. We have used a cluster of Sun workstations for our implementation. We have carried out a series of experiments to evaluate our implementation. The main results from our experiments are as follows: 1) High speedups are achieved on both dense and sparse datasets, even though we have used simple algorithms that sequentialize a part of the computation, 2) The execution time depends only upon the amount of computation, and does not increase in a super-linear fashion as the dataset size or the number of tiles increases, and 3) As the datasets become more sparse, sequential performance degrades, but the parallel speedups are still quite good.

The rest of this paper is organized as follows. The data cube construction problem is described in Section 2. Our parallel algorithms are described in Section 3. Our implementation is described in Section 4. We evaluate our implementation in Section 5. We compare our work with related research efforts in Section 6 and conclude in Section 7.

2 Data Cube Construction Problem

In this section, we describe the multidimensional data model for storing data in an organization. We then describe what a data cube is, and what the main issues are in constructing a data cube.

Organizations often find it convenient to express facts as elements of a (possibly sparse) multidimensional array. For example, a retail chain may store sales information using a three-dimensional dataset, with item, branch, and time being the three dimensions. An element of the array depicts the quantity of the particular item sold, at the particular branch, and during the particular time-period.

In data warehouses, typical queries can be viewed as group-by operations on a multidimensional dataset. For example, a user may be interested in finding sales of a particular item at a particular branch over a long duration of time, or all sales of all items at all branches for a given time-period. The former involves performing an aggregation along the time dimension, whereas the latter involves aggregations along the item and the branch dimensions.

To provide fast response to the users, a data warehouse computes aggregated values for all combination of values. If the original dataset is $n$ dimensional, this implies computing and storing $n(n-1)/2$ $n-2$ dimensional arrays, and so on. The resulting structure is called a data cube. The process of constructing such a structure is called data cube construction.

Data warehouses contain large datasets, involving both a large number of dimensions and a large number of attributes along each dimension. Thus, data cube construction is a compute and data intensive problem.

We now introduce some standard terminology associated with a data cube [12]. For simplicity, assume that the original dataset is three-dimensional. Let the three dimensions be $A$, $B$, and $C$. The sizes along these dimensions are $A_d$, $B_d$, $C_d$, respectively. Without loss of generality, we assume that $A_d \leq B_d \leq C_d$.

We denote the original array by ABC. Then, data cube construction involves computing arrays AB, BC, AC, A, B, C, and a scalar value all. The array AB has size $A_d \times B_d$. The arrays AB, BC, and AC need to be computed from ABC, by aggregating values along the dimensions C, A, and B, respectively. However, the array A can be computed from either AB or AC, by aggregating values along dimensions B or C. Because $B_d \leq C_d$, it requires less computation to compute A from AB.

A lattice can be used to denote the options available for computing each array within the cube. This lattice is shown in Figure 1. A data cube construction algorithm chooses a minimal spanning tree of the lattice shown in the figure. The overall computation involved in the construction of the cube
is minimized if each array is constructed from the smallest parent. Thus, the selection of the smallest parent for each node in the lattice is one of the important considerations in the design of a sequential algorithm. Another important consideration is cache and memory reuse for the input dataset. When a portion of the array ABC is brought into the cache or memory, we want to update appropriate portions of AB, BC, and AC. This avoids the need for reloading each portion of ABC several times.

3 Algorithms

In this section, we present parallel algorithms for data cube construction that combine interprocessor communication and tiling.

3.1 Background and Goals

In ADR’s model of computation, the input data is divided into chunks and brought into the main memory one at a time. The idea of chunks is consistent with what is used in sequential data cube construction algorithms [11, 12]. It is desirable that all computations associated with a chunk are performed when a chunk is read. This model alleviates the need for storing a large portion of input dataset in the main memory at any given time. The output datasets may also not fit in main memory. If this is the case, the output datasets are partitioned into tiles. Each tile must fit into main memory. Tiles are allocated one at a time and all computations on these tiles are performed before writing them back into disks. If an input chunk contributes to more than one tile, then it needs to be read more than once. To avoid the need for reading the same chunk multiple times, a portion of the output dataset may be allocated for the entire computation. However, the sum of the sizes of such a portion and each tile must not exceed the available main memory.

ADR’s management of input and output datasets reduces programmer’s burden significantly. However, algorithms for data-intensive computations must be designed to suit ADR’s model of computation. In summary, we have the following goals while designing parallel data cube construction algorithms:

- Each array must be computed from the smallest parent, as discussed in Section 2.
- Any input chunk brought into the memory should be used for all possible computations before being deallocated. In other words, as much cache and memory reuse should be done as possible. For the ADR’s model of computation, it also implies that tiling should be done so as to avoid the need for reading an input chunk more than once.
- The volume of interprocessor communication should be minimized. This is an important consideration in designing any parallel algorithm. A high ratio of communication to computation can easily limit the scalability of a parallel implementation.

- If any portion of the output space is in memory during the entire computation, its size should be minimized.
- The algorithm should easily fit into ADR’s model of computation, while meeting the above goals.

3.2 Data Cube Construction Algorithms

For simplicity in presenting our algorithms, we focus on the case when the original array is three dimensional. The three dimensions are denoted by $A$, $B$, and $C$. The sizes of these dimensions are denoted by $A_d$, $B_d$, and $C_d$, respectively. Without loss of generality, we assume that $A_d \leq B_d \leq C_d$.

When performing the data cube construction in parallel, the first step is to partition the input dataset between processors. For keeping our implementation simple, we partition along a single dimension only. The total communication volume is minimized when partitioning is done along the largest dimension. When the dataset is partitioned along the dimension $C$, the communication volume is $A_d \times B_d$, which is lower than $A_d \times C_d$ or $B_d \times C_d$. 

![Figure 2. Partitioning and tiling in case 1](image)

![Figure 3. Algorithm for the case $A_d \leq B_d \leq C_d$.

Tiling is done across dimension $C$.](image)
After partitioning along the dimension C, let the size of the dimension C on each processor be $C_d'$. We separately consider the following three cases:

1. $A_d \leq B_d \leq C_d'$.
2. $A_d \leq C_d' \leq B_d$.
3. $C_d' \leq A_d \leq B_d$.

Different strategies need to be used for each of the above three cases. The second and the third case, however, are very similar. Therefore, we present algorithms for the first and the second cases only.

We initially consider the first case. The first decision to be made is of tiling. For convenience, we tile along a single dimension. Studying potential benefits of tiling across multiple dimensions is a topic for future research. We want to tile along the dimension which minimizes the size of the dataset that needs to be in memory during the entire computation. Tiling along the dimension C implies that an array of size $A_d \times B_d$ needs to be in memory during the entire computation. Similarly, tiling along the dimensions A and B would imply that arrays of sizes $B_d \times C_d'$ and $A_d \times C_d'$, respectively will need to be in memory. We tile along the dimension C since $A_d \times B_d \leq A_d \times C_d' \leq B_d \times C_d'$. Let $C_d''$ be the size of the dimension C for each tile. Sufficient number of tiles are created so that $A_d \times B_d + A_d \times C_d'' + B_d \times C_d''$ does not exceed the available main memory for output datasets. This case is graphically shown in Figure 2. The figure shows the case when there are four processors and two tiles are created on each processor.

The algorithm for the first case is presented in Figure 3. One tile is processed at a time. Portions of arrays AC and BC corresponding to the extent of dimension C for the current tile and the particular processor are allocated with this tile. The input chunks whose extent of C dimension overlaps with the current tile need to be read while processing this tile.

As each chunk is brought into main memory, portions of AB, BC, and AC are updated. After processing all input chunks corresponding to the tile, final results for portions of AC and BC are available. AC is also used to compute the final result for the portion of C. The portions of AC, BC, and C are written back after the tile is processed. After the last tile has been processed, interprocessor communication is performed to obtain the final value for elements of AB. Each processor has an array of size $A_d \times B_d$ allocated for aggregating along the portions of C owned by the processor. The processor 0 performs a reduction operation to compute the final value for each element of the array AB. After AB has been computed, the processor 0 also computes A, B, and all. Note that this part of the computation is not parallelized but just performed on a single processor. We believe that this simplifies the implementation, without a significant change in the performance. We will further validate this claim when we experimentally evaluate our implementation.

This algorithm has the following properties:

- All arrays are computed from their smallest parents.
- Each input chunk needs to be brought in the main memory only once. An exception to this property will arise when extent of a chunk along C dimension may overlap with multiple tiles. However, the input dataset can be divided into chunks to avoid this problem.
- The interprocessor communication volume is minimal among any possible single dimensional partition.
- The portion of the output space kept in the main memory for the entire computation is minimal of all single dimensional tiling possibilities.

We will explore the suitability of implementation of this algorithm using ADR in the next section.

![Figure 2. Partitioning and tiling in case 2](image-url)
Implementation Using a Middleware

In this section, we give a brief overview of our implementation of the data cube construction algorithms using a cluster middleware called Active Data Repository (ADR) [3, 4].

ADR was designed for scientific data intensive applications. The main underlying observation behind the design of ADR was that data intensive applications from many important scientific domains, including satellite data processing, analysis of microscopy data, and simulation coupling, share important common properties. The basic computation in these application consists of 1) retrieving disk-resident input elements, 2) mapping the coordinates of the retrieved input items to the corresponding output items, and 3) aggregating, in some way, all the retrieved input items mapped to the same output data items. The computation of a particular output element is a reduction operation, i.e. the correctness of the output usually does not depend on the order in which the input data items are aggregated.

ADR offers an interface that simplifies the implementation of applications which have the above structure. An application programmer can develop an application using ADR by writing a number of C++ virtual functions. These applications are invoked by the ADR runtime support for performing the processing specific to the application. ADR’s runtime support includes task planning, which uses the user provided information about the application to generate a task schedule. A task schedule includes information about when a particular input chunk is to be read, and how I/O and communication are to be overlapped with computation.

The key functions in implementation of data cube construction using ADR are shown in Figure 6. The main function shows the ADR execution loop that invokes several functions written by the application programmer. Outline of the codes for local reduction, global reduction, and finalize are shown in this Figure.

ADR’s functionality simplifies the implementation task in several ways. The two critical factors are: 1) tiling and reading the input using asynchronous operations, and 2) simplifying the specification of interprocessor communication.

While processing each output strip, all input chunks that can update any element in the strip must be read. ADR decides which chunks are read for each time, the order in which these chunks are read, and when asynchronous I/O operations can be issued. ADR’s task planning phase uses two programmer provided function to make the above decisions. The first function is stripmine, which tells how the output dataset is tiled, and the second is project, which tells which output elements are updated using a given input element. Input chunks are read in an order so that disk seek-time is minimized. All input chunks are read using asynchronous I/O operations. Thus, use of ADR facilitates maximal memory reuse, minimal seek time, and overlap of I/O and computation, while requiring very small programming effort from the programmer.

Programming of interprocessor communication is also simplified by the use of ADR. The programmer needs to write code to copy partial results in a buffer, to perform the global reduction operation on one node, and also needs to specify the size of buffer. MPI calls are directly invoked by the ADR runtime support and are not the responsibility of the programmer.

5 Experimental Results

This section presents an experimental evaluation of our implementation.

In constructing data cubes, the initial multi-dimensional array can be stored in a dense format or a sparse format [12]. A dense format is typically used when 40% of array elements have a non-zero value. In this format, storage is used for all elements of the array, even if their value is zero. In a sparse format, only non-zero values are stored. However, additional space is required for determining the position of each non-zero element. We use chunk-offset compression,

---

Allocate AC and A
Foreach tile:
  Allocate AB and BC
  Foreach input chunk to be read:
    Update AB, AC, and BC
    Perform global reduction to obtain final AB
If (proc_id = 0)
  Compute B from AB
  Update A using AB
  Write-back AB, BC, and B
If (last tile)
  Finish AC
  Compute C from AC
If (proc_id = 0)
  Finish A
  Compute all from A

---

Figure 5. Algorithm for the case $A_d \leq C^d \leq B_d$. Tiling is done along dimension B.
Main() {
    StripMine();
    foreach (strip) {
        foreach chunk in strip
            LocalReduction();
        GlobalCombination();
    }
    Finalize();
}

LocalReduction(Chunk abc) {
    AB += abc.reductionToAB();
    AC += abc.reductionToAC();
    BC += abc.reductionToBC();
}

GlobalCombination() {
    if (Tiling along C) {
        output(AC);
        output(BC);
        C += AC.reductionToC();
        output(C);
        clear(AC,B,C);
    } else {
        output(BC);
        clear(BC);
        sendto(proc_0,AB);
        if (proc_id == 0) {
            AB += receivefrom(othernodes,AB);
            output(AB);
            B += AB.reductionToB();
            A += AB.reductionToB();
            output(B);
            clear(AB,B);
        }
    }
}

Finalize() {
    if (Tiling along C) {
        sendto(proc_0,AB);
        if (proc_id == 0) {
            AB += receivefrom(othernodes,AB);
            output(AB);
            A += AB.reductionToAB();
            B += AB.reductionToAB();
            output(A,B);
            clear(AB,A,B);
        }
    } else {
        output(AC);
        C += AC.reductionToC();
        output(C);
        clear(AC,C);
    }
}

Figure 6. Outline of the Middleware Specification for Implementing Data Cube Construction

used in other data cube construction efforts [12]. Along with each non-zero element, its offset within the chunk is also stored. After aggregation, all resulting arrays are always stored in the dense format. This is because the probability of having zero-valued elements is much smaller after aggregating along a dimension.

The only difference between processing dense and sparse datasets arises in the local reduction function. The extra processing required for sparse datasets is conceptually simple and not shown in Figure 6. However, the ratio of communication to computation increases as the sparsity of the initial array increases. This is because the computation involved in data cube construction primarily depends upon the number of non-zero elements in the array ABC, while the volume of communication depends upon the total size of the array AB.

Our implementation includes processing for both sparse and dense datasets, and for both the cases we described in Section 3. We have carried out experiments with the following goals:

- Evaluating the speedups obtained with both dense and sparse datasets
- Measuring the increase in processing time with increase in dataset sizes.
- Measuring the increase in processing time with increasing the number of tiles in which the output is partitioned.
- Evaluating the impact of sparsity on both uniprocessor and parallel performance.

Our experiments have been conducted on cluster of workstations. We used 8 Sun Microsystems Ultra Enterprise 450’s, with 250MHz Ultra-II processors. Each node has 1 GB of main memory which is 4-way interleaved. Each of the node have a 4 GB system disk and a 18 GB data disk. The data disks are Seagate-ST318275LC with 7200
rotations per minute and 6.9 milli-second seek time. The nodes are connected by a Myrinet switch with model number M2M-OCT-SW8.

Figure 7 shows the speedups achieved on dense datasets. The increase in execution time with increase in dataset sizes is also shown in this figure. Three datasets, with sizes $512 \times 512 \times 1024$, $512 \times 1024 \times 1024$, and $1024 \times 1024 \times 1024$ are used. The total memory requirements for these three cases are 1 GB, 2 GB, and 4 GB, respectively. Further, the ratio of communication to computation is in these three cases is identical. Performance on 1, 2, 4, and 8 nodes is shown for each of these datasets.

The speedups on 8 nodes are 7.78, 7.85, and 7.84, for the three datasets, respectively. In other words, the parallel efficiency is 97% or higher. Recall that in our algorithms, computation of A and B from AB is done sequentially. This was done to simplify our implementation using the middleware. Our experimental results show that executing this portion sequentially does not have a noticeable impact on performance.

As the dataset size is doubled and quadrupled, the execution time on 1 node increases by a factor of 1.98 and 3.94, respectively. The increase in execution time on 8 nodes is 1.97 and 3.91, respectively. These results show that as the dataset size increases, the computation time stays proportional to the dataset size. So, I/O costs are not dominant with increase in execution time.

Figure 8 shows the speedups achieved and scalability with dataset sizes for sparse datasets. We have experimented with three sparse datasets, where the sparsity (fraction of non-zero elements) was 25%. The array sizes are $512 \times 512 \times 1024$, $512 \times 1024 \times 1024$, and $1024 \times 1024 \times 1024$. The total storage requirements are .5 GB, 1 GB, and 2 GB, respectively. With sparse format, each non-zero element requires 8 bytes, because an offset also needs to be stored.

The speedups on 8 nodes are 6.89, 7.08, and 7.14 on .5 GB, 1 GB, and 2 GB datasets, respectively. The speedups obtained are lower than those on dense datasets. This is because of increased ratio of communication to computation. This ratio reduces as we go to larger datasets, which explains the increased speedups. The parallel efficiency is 86% or higher in all cases.

By doubling and quadrupling the dataset, the execution time on 1 node increases by a factor of 2.0 and 4.02, respectively. On 8 nodes, these factors are 1.94 and 3.86, respectively. This again shows that the execution times stay proportional to the amount of computation as the dataset size is increased.

All our experiments so far involved only a single tile per node. One interesting issue is the scalability of performance as larger number of tiles on each processor are used. In Figure 9, we study this effect. We initially consider a dense $512 \times 512 \times 1024$ dataset that uses 1 tile on each node. We experimented with 3 more datasets, with 2 times, 4 times, and 8 times as much data. We used 2, 4, and 8 tiles, respectively, for these 3 datasets. The execution times for all 4 cases, on 2 nodes, are shown in the Figure. The execution time increases by a factor of 2.04, 4.0, and 7.92, when there are 2, 4, and 8 tiles, respectively. This shows that the execution time is primarily dependent upon the amount of computation and not the number of tiles processed.

Our final experiment focuses on evaluating the impact of sparsity on uniprocessor and parallel performance. We ex-
experimented with 4 datasets where the fraction of non-zero elements was 25%, 10%, 5%, and 1%, respectively. The four datasets had the same number of non-zero elements. The array sizes were $1024 \times 1024 \times 1024$, $1024 \times 1280 \times 2048$, $1280 \times 2048 \times 2048$, and $1024 \times 5120 \times 5120$, respectively. Each dataset is 2 GB in size. The amount of computation involved in data cube construction for these four datasets is almost identical. This is because the dominant part of the computation comes in evaluating $AB$, $AC$, and $BC$ from $ABC$, which depends only upon the number of non-zero elements. The costs of computing $A$, $B$, $C$, and $all$ are dependent upon the sizes of dimensions, which are higher for a more sparse dataset with the same number of elements.

The execution times on these four datasets on 1, 2, 4, and 8 nodes are presented in Figure 10. Execution times on 1 node show an interesting trend. Compared to the execution time for the 25% sparse array, execution times for 10%, 5%, and 1% cases are higher by 6%, 16%, and 86%, respectively. This difference arises because as sparsity increases, each input updates a larger portion of arrays $AB$, $BC$, and $AC$. This has two consequences. First, there is relatively poor cache reuse when an input chunk updates larger arrays. The second reason is more specific to our implementation. We allocate and initialize temporary arrays corresponding to portions of $AB$, $AC$, and $BC$ that are updated after processing one chunk. After processing an input chunk, these temporary portions are merged with the main copy. The overhead associated with the extra initialization and merge increases with the size of temporary array, and therefore, with the sparsity of the datasets.

The speedups on 8 nodes for 25%, 10%, 5%, and 1% cases are 7.36, 7.19, 6.70, and 6.52, respectively. As the dataset becomes more sparse, the ratio of communication to computation increases, which results in lower speedups.

6 Related Work

The algorithms for data cube construction that we have presented here are quite different from the existing work on sequential and parallel data cube construction [2, 7, 8, 12]. In existing approaches, a portion of the output is written as soon as its final value has been computed. In contrast, we use the aggregate main memory to reduce the frequency of write operations. Moreover, we have focused on using a middleware for easing the implementation task.

Data cube construction is a significantly different application than any previous application implemented using ADR [1, 5, 4, 6, 10].

7 Conclusions and Future Work

In this paper, we have explored the use of a cluster middleware for implementing a parallel data cube construction algorithm. By developing suitable parallel algorithms, we are able to use Active Data Repository (ADR), a cluster middleware originally developed for scientific data-intensive problems. Our algorithms preserve the important properties that are desirable from any parallel data cube construction algorithm.

We have described our implementation experience and have argued how implementation of this compute and data intensive problem is simplified by the use of the middleware. We have conducted a series of experiments to evaluate our implementation. The main results from our experiments are as follows: 1) High speedups are achieved on both dense and sparse datasets, even though we have used simple
algorithms that sequentialize a part of the computation, 2) The execution time depends only upon the amount of computation, and does not increase in a super-linear fashion as the dataset size or the number of tiles increases, and 3) As the datasets become more sparse, sequential performance degrades, but the parallel speedups are still quite good.

We are currently working on implementing data cube construction for larger number of dimensions. We are exploring potential benefits of partitioning and tiling along multiple dimensions.

References


