Computing Multidimensional Aggregates in Parallel

Weifa Liang
Department of Computer Science
The Australian National University
Canberra, ACT 0200, Australia
E-mail: wliang@cs.anu.edu.au

Maria E. Orlowska
Distributed Systems Technology Centre
Department of Computer Science and Electrical Engineering
The University of Queensland
St. Lucia, QLD 4072, Australia
E-mail: maria@csee.uq.edu.au

Keywords: data cube, parallel algorithms, OLAP, aggregation computation, data warehousing

Edited by: Rudi Murn

Received: **Revised:** **Accepted:**

Computing multiple related group-bys and aggregates is one of the core operations of On-Line Analytical Processing (OLAP) applications. This kind of computation involves a huge volume of data operations (megabytes or treabytes). The response time for such applications is crucial, so, using parallel processing techniques to handle such computation is inevitable. In this paper we present several parallel algorithms for computing a collection of group-by aggregations based on a multiprocessor system with sharing disks. We focus on a special case of the aggregation problem–"Cube" operator which computes group-by aggregations over all possible combinations of a list of attributes. The proposed algorithms introduce a novel processor scheduling policy and a non-trivial decomposition approach for the problem in the parallel environment. Particularly, we believe the proposed hybrid algorithm has the best performance potential among the four proposed algorithms. All the proposed algorithms are scalable.

1 Introduction

Aggregation is a predominant operation in decision support database systems. On-Line Analytical Processing (OLAP) often needs to summarize data at various levels of detail and on various combinations of attributes. Recently [8] introduced the *data cube* operator for the convenient support of such aggregates in OLAP. In technical terms, the data cube is a redundant multidimensional projection of a relation on subsets of its schema. It computes all possible group-by SQL operators and aggregates their results into a n-dimensional space for answering OLAP queries. In general, if we consider an n-attribute relation R, there are 2^n group-by aggregations. In this paper we refer to the computation of the 2^n group-by aggregations as the *data cube* computation of R. [10] suggests a search lattice structure for computing the data cube. The lattice is a directed acyclic graph (DAG) $\mathcal{L} = (V, E)$, where each node $v \in V$ represents a group-by of the data cube. Each directed edge $\langle u, v\rangle \in E$ from group-by u to group-by v illustrates that v can be generated from u. u is called the *parent* of v, also v has exactly one attribute less than u. Thus, the outgoing degree of any node with d attributes is d. Let n be the number of attributes in a base relation R and $l \leq n$. Define *level* l of the search lattice as the level where all group-bys contain exactly $n - l + 1$ attributes. Each group-by v in the lattice is also called a *cuboid* in the data cube. The

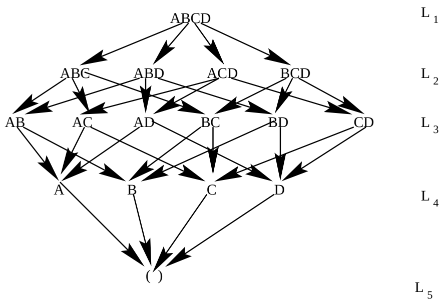

Figure 1: A lattice for the cube based on R.

size of a cuboid is the number of tuples in a relation defined by the corresponding projection. In this paper we will use the terms *node*, *group-by*, or *cuboid* interchangeably. Figure 1 is an example of a search lattice of a relation R with four attributes A, B, C and D, and each node is labeled by its attribute subset. There are $2^4 = 16$ different group-by aggregations for R, and each group-by corresponds to a table derived from R.

The response time is a primary concern in OLAP applications [3]. Interactive analysis (response time in seconds) is hard to achieve if each of the aggregates is computed "on the fly" at query execution time. Therefore, most solutions for the OLAP application precompute the aggregates and run interactive sessions against the precomputed data. It must be mentioned that efficiency is also important for the precomputation time, since this is a lower bound on the recency of the data available to analysts and decision makers. Furthermore, the cost of precomputation influences how frequently the precomputed data is brought up-to-date.

There are two fundamental problems related to the cube operator. One is related to the efficient computation of all dimensional aggregates (group-bys). Another focuses on the selection of a subset of aggregates for materialization in order to reduce the response time of the queries. This occurs when it is impossible to store all aggregates in a data warehouse, due to a limited storage space and other constraints. In this paper we shall focus on the efficiency issues by presenting parallel computational solutions. Following [1, 4, 14], we assume that the aggregating functions are *distributive* [8]. Examples of distributive functions include `max, min, count` and `sum`, etc.

1.1 Related work

Basically there are two approaches to compute a group-by: (1) the sort-based method and (2) the hash-based method [7]. Methods of computing single group-bys in both sequential and parallel environments have been well studied (see [7] for a survey, also see [15]). However, little work has been done on optimizing the total operations for a collection of related aggregates. The problem was first introduced by [8]. In their seminal paper they give some rules of thumb to be used for an efficient implementation of the data cube operator. These include the `smallest parent optimization` (a node is generated from a parent who has the minimum number of tuples), and the partitioning of data by attribute values. Since then, several works have been carried out along this direction. For example, [1, 14] embed five optimization strategies: `smallest-parent, cache-results, amortize-scans, share-sorts` and `share-partitions`, into a single algorithmic framework, and suggest algorithms `pipesort` and `pipehash` (based on sort-related and hash-related grouping methods) to implement the cube operator. [1, 4] give overlap algorithms for the data cube computation. Their algorithms seek to minimize the number of sorting steps required to compute many sub-aggregates that comprise a multidimensional aggregate, and make efficient use of the available memory to reduce the number of I/Os, i.e, by overlapping the computation of the various cuboids, this thereby reduces the number of sorting steps required. [13] considers the sparse relation by giving a recursive algorithm for the cube operator. Their solution is based on two fundamental ideas that have been successfully used for performing complex operations (e.g. sorting and joins) over large relations: (1) partition the large relations into fragments that fit in memory; and (2) perform the complex operation over each memory-sized fragment independently. [17] gives an algorithm for computing the cube by using a multidimensional array rather than a table and compression techniques. All of the above mentioned papers only consider computing all dimensional group-by aggregations of R, while [10, 9] present algorithms for deciding what group-bys to precompute and index if not all aggregates are needed. [12] presents efficient algorithms for the on-line maintenance of data cubes.

1.2 Contributions

So far all known algorithms for computing cubes are designed for the sequential uniprocessor computing environment. As it is well understood, the cube computation usually involves large data operations. Since the response time for such computation is crucial for the successful deployment of OLAPs, it seems that applying parallel processing techniques to this problem is interesting possibility. In [8], they mention the possibility of applying parallelism for the data cube computation. However, they do not exploit this concept. In this paper we shall propose several parallel algorithms to compute the data cube efficiently, based on a multiprocessor system with sharing disks. The proposed algorithms introduce a novel processor scheduling policy and a non-trivial decomposition approach for the problem in the parallel environment. We shall first introduce in detail a naive algorithm for the problem and then present an optimization algorithm for it based on the cost estimation using the minimum spanning tree (MST) method. We shall also give another optimization algorithm for the problem based on the cost estimation using the minimum weighted matching method. We shall finally propose a hybrid algorithm which makes use of the advantages of the previous two optimization algorithms. We believe that the hybrid algorithm has the best performance potential among the four algorithms introduced in this paper. All the proposed algorithms are scalable.

The rest of the paper is organized as follows. In Section 2 we shall introduce the parallel computational model, and then present some results for the single group-by aggregation in the parallel computational environment. In Section 3 we give three parallel algorithms for the cube operator, while in Section 4 we present a hybrid parallel algorithm for the problem by incorporating the two algorithms discussed in Section 3. In Section 5 we extend our result to a generalization in which we compute only some group-by aggregates in a data cube by optimizing the total operations on these aggregates. We conclude our discussion in Section 6.

2 Preliminaries

2.1 The parallel computational model

We assume that a multiprocessor system consists of p processors, and each processor has its own local main memory. All processors are assumed to be identical and connected with an array of disks which is used to store the initial data (the base relation table) and the final data (all dimensional group-by aggregations) through a high-speed interconnection network. We further assume that the sizes of the base relation table R and these group-bys are very large and cannot fit in the main memory.

2.2 Parallel algorithms for a single aggregate

There are a number of parallel algorithms for processing a single aggregate [2, 7, 15]. The generic framework for this kind of processing consists of two phases. In the first phase each processor executes aggregation on its local partition of the relation. In the second phase, all processors merge these partial results to produce the final result. For example, [2] discusses two sorting-based algorithms for aggregate processing on a shared disk cache architecture. [7] gives an optimization for dealing with bucket overflow for the two phase algorithm based on the hash-joining method. Recently, [15] considers a range of grouping selectivities by presenting an adaptive parallel algorithm for the problem.

Now let us consider a single aggregate v on a table R'. Assume that there are p' processors available for computing v. From the proposed algorithms above, we can derive that the time t_v used for computing v is inversely proportional to the number of processors used, i.e., the more processors used, the less time is needed. In general, t_v can be expressed as follows.

$$t_v \quad = \quad a \cdot |R'|/p' + b \cdot p' \tag{1}$$

where $|R'|$ is the number of tuples in R', a and b are the parameters dependent on the computing environment in which a is related to the time for read/write an I/O page and b is related to the communications overhead among the p' processors. In this paper we assume that the parallel algorithm for the single aggregate v is available [15], and refer to it as `Aggregate_multiprocessor(`v, p'`)`.

3 Parallel Algorithms

In this section we introduce three parallel algorithms for the efficient computation of data cubes, and provide an analysis of these algorithms in terms of the response time.

3.1 A naive parallel algorithm

Without loss of generality, let us assume that the base relation table R has n attributes and $|R|$ tuples. Assume that the multiprocessor system consists of p processors, and each processor P_i is responsible for a horizontal fragment R_i of R, where $R_i \cap R_j = \emptyset$, $\bigcup_{i=1}^{p} R_i = R$ and $|R_i| \approx |R|/p$, $i \neq j$ and $1 \leq i, j \leq p$.

The algorithm presented below is rather natural and consists of two phases. In the first phase, each processor computes the data cube of R_i, and stores the data cube back to the disks. Note that all processors execute their tasks independently. In the second stage, we merge the data cubes of the fragments of R to form the original data cube of R through a number of rounds of parallel merging. The detailed algorithm is described as follows.

Algorithm 1:

1. Horizontally partition R into p equal parts roughly. Let processor P_i be responsible for the fragment R_i of R.
2. **for** each processor P_i : $1 \leq i \leq p$ **pardo**
   compute the data cube of R_i and write
   the results back to the disks using
   an efficient sequential algorithm
   endfor;
3. **for** $k := 1$ **to** $\lceil \log p \rceil$ **do**
   Let G_i be a cluster of processors indexed from $2^k(i-1)+1$ to $i2^k$.
   for each cluster i : $0 \leq i \leq p/2^k$ **pardo**
      processors in G_i merge the data cubes
      obtained by them in the previous iteration
      and write the results back to the disks
   endfor

endfor.

Now, let us analyze the response time of the proposed algorithm. The time complexity T_p in Step 2 is approximately $1/p$ of the sequential time if carrying such a computation by a uniprocessor. This is because each processor only handles a fragment whose size is about $1/p$ of the size of the base relation table. Before running Step 3, there must be a synchronization at the end of Step 2. Now we consider the implementation of Step 3, assuming that every group-by has been sorted already during the first stage. Then, we merge these group-bys involved. So, the total of $\lceil \log p \rceil$ rounds is necessary. Since we assume that there is a high-speed interconnection network available, the read/write conflicts from and to the disks between different processors are negligible and ignored in our consideration afterwards.

`Algorithm 1` has some drawbacks when the data distribution on some dimensional aggregate is heavily skewed. The reason is that the synchronization at the end of Step 2 takes $\Delta = t' - t$ unit times, where $t = \min_{1 \leq i \leq p}\{t_i\}$ is the earliest finishing time by some processor and $t' = \max_{1 \leq i \leq p}\{t_i\}$ is the latest finishing time by some processor, t_i is the time of processor P_i for computing the data cube of R_i. As a result, the more skew the data distribution among processors is, the larger the Δ becomes. Meanwhile, the synchronization is also required between different rounds in the second stage.

In order to reduce the synchronization time for skewed data distribution, we propose the following approach. That is, when we compute a group-by aggregation or a merging (in the second stage), we first run a sub-routine to check whether the data distribution among the processors is skewed. If this is the case, we first re-distribute the data among the processors in order to make the processors involved are well load balanced. We then run through the steps of `Algorithm 1`. Certainly this will lead to a better response time for this case. However the load-balancing computation and data re-distribution in a multiprocessor system involve communications and data transfer among the processors which are also time consuming. Therefore, there must be a trade-off in terms of overheads between synchronization and load-balance.

3.2 An MST-based parallel algorithm

`Algorithm 1` gives some optimization for the cube operator in the total operations required for each fragment. But this local optimization is not necessarily the global optimization. In this section we introduce our second parallel algorithm `Algorithm 2` which in some sense presents global optimization in terms of the total operations for the cube operator of R. Actually, `Algorithm 2` is a parallelization of the smallest parent method [8].

Basically, the algorithm consists of two phases. In phase one we produce a scheduling for the cube computation by establishing an estimated cost tree which will be defined later. And in the second phase we assign processors to the nodes in the tree and implement the cube computation.

As mentioned in the introduction, the cube computation can be implemented through building a search lattice. Now we assign a weight for each directed edge in the lattice which represents the computation cost of a node from its parent. We then produce such a directed minimum weighted spanning tree (also called optimal branching) in the lattice that each node has in-degree one except the root with in-degree zero. Then, the weighted sum of the edges in the tree represents the minimum total operations for the cube computation.

However, before the actual computation of each node (a group-by aggregate in the data cube) in the tree, there is no way of knowing its size exactly. Therefore, in practice we can randomly withdraw a small fraction of pages from the disk, which stores the base relation R_i as random samples, and use them to estimate the size of every node in the cube by using some efficient estimating methods (e.g. uniform distribution estimation). Thus, in the first phase, in fact, we build an *estimated* directed minimum spanning tree T for the lattice since the size of every node has an estimated value. The detailed procedure for this phase is introduced as follows.

Phase one: compute an estimated directed MST.

1. **for**each processor P_i: $1 \leq i \leq p$ **pardo**
   Let P_i be responsible for fragment R_i.
   P_i builds the lattice $\mathcal{L}_i$ for R_i and assigns the weights to the edges of $\mathcal{L}_i$.
   The weights are obtained by randomly fetching a small fraction of the pages of R_i from the disks and estimating the size $S_{i,k}$ of each node cuboid$_{i,k}$ in $\mathcal{L}_i$ [8].
   endfor;
2. Construct the lattice $\mathcal{L}$ of R and assign weights to its edges.
   The estimated size S_k of a node cuboid$_k$ in $\mathcal{L}$ thus is $\lceil \sum_{i=1}^{p} S_{i,k}/p \rceil$. Assign all outgoing edges from cuboid$_k$ in $\mathcal{L}$ with weight S_k.
3. Find a directed minimum spanning tree T on $\mathcal{L}$.

Step 1 can be done easily because only a couple of pages are withdrawn from R_i to estimate the size of every node in $\mathcal{L}_i$. Step 2 involves the construction of $\mathcal{L}$ and the weight assignment of the edges in $\mathcal{L}$. Note that the size estimate of every node in $\mathcal{L}$ is now more accurate than it shows in Step 1 because p samples are used this time. Step 3 can be implemented as follows. For each node, except the root, it finds an incoming edge with the minimum weight, then another endpoint of the edge is the parent of the node in T. It is also straightforward to prove that the tree generated by this method is a directed minimum spanning tree for this special case.

In phase two we assign processors to the nodes in T such that the response time for computing all nodes in T is minimized, i.e., we aim to achieve the minimum response time for the data cube computation. However, how to assign processors to and how many processors are assigned to a node are very crucial to the entire response time. In this phase we present a processor allocation and deallocation scheduling procedure for the nodes in T.

Before we proceed, let us consider a node v in T. Assuming that each node in T has been assigned a weight which is the cost of generating the node. This assignment is carried out as follows. The root of T is assigned the cost for sorting its tuples by some order of all attributes. Now let $s(v)$ be the weight of the edge from the parent of v in T to v. We assign v by weight $s(v)$. T becomes a *node-weighted* tree afterwards.

Now we consider the processor allocation for T. Let W_T be the total estimated operations for the cube computation by T. Then,

$$W_T = \sum_{k=1}^{2^n} s(v_k). \qquad (2)$$

From equation 2, a lower bound of the response time for the cube computation based on T is obtained as follows.

$$T_{\min} = \lceil W/p \rceil. \qquad (3)$$

Basically our processor allocation algorithm for T is based on the estimation of the time lower bound $T_{\min}$. The basic idea behind the algorithm is that the number of processors allocated for each subtree is proportional to the total operations needed to compute all nodes in that subtree. Thus, all the subtrees whose roots share a parent can finish their computation in the same time approximately.

Recall that T is a node-weighted tree, and there are p processors available. In what follows, we first assign p processors to compute the root r of T. Let r have d children, $v_1, v_2, \ldots, v_d$, $d \leq n$, T_i the subtree rooted at v_i and W_{T_i} the weighted sum of the nodes in T_i, $1 \leq i \leq d$. It is obvious that v_i can be computed only after r (a table) is available, and the time lower bound for computing the nodes in these subtrees thus is

$$t_{\min} = \lceil \sum_{i=1}^{d} W_{T_i}/p \rceil. \qquad (4)$$

Now we allocate the p processors to the subtrees. Assume that $W_{T_1} \geq W_{T_2} \geq \ldots \geq W_{T_d}$. We scan the subtrees by this order. Let T_i be the subtree which we are considering. If $\lfloor \frac{W_{T_i}}{t_{\min}} \rfloor \neq 0$, we assign $\lfloor \frac{W_{T_i}}{t_{\min}} \rfloor$ processors to T_i; otherwise, we merge T_i with T_{i+1} and repeat the above procedure until all subtrees have been assigned processors. In the end we claim that the number of processors used for this allocation is no more than the actual available number of processors. In the following, we show that this claim is true. Let p' be the actual number of processors used for this assignment, then,

$$\begin{aligned} p' &= \sum_{i=1}^{d} \lfloor \frac{W_{T_i}}{t_{\min}} \rfloor \leq \frac{\sum_{i=1}^{d} W_{T_i}}{t_{\min}} \\ &= \frac{\sum_{i=1}^{d} W_{T_i}}{\lceil \sum_{i=1}^{d} W_{T_i}/p \rceil} \leq \frac{\sum_{i=1}^{d} W_{T_i}}{\sum_{i=1}^{d} W_{T_i}/p} = p \quad (5) \end{aligned}$$

We then turn to the processor allocation inside each subtree or a subset of subtrees by applying our allocation approach recursively. The detailed algorithm is outlined as follows.

Procedure `Processor_Allocation`(T, r, p)
/* T is a weighted-node tree, r is the root of T, */
/* and p is the number of processors. */
/* Compute r by assigning p processors to it,*/
/* e.g., [15] algorithm can be applied. */
`Aggregate_multiprocessor`(v, p);
$t_{\min} := \lceil \frac{W_T - s(r)}{p} \rceil$; /* the time lower bound. */
Let $v_1, v_2, \ldots, v_d$ be the children of r,
and T_i be a subtree rooted at v_i, $1 \leq i \leq d$.
$i := 0$;
repeat
  $i := i + 1$;
  $p' := \lfloor \frac{W_{T_i}}{t_{\min}} \rfloor$;
  if $p' \neq 0$ **then**
    allocate p' processors to T_i;
    if $p' > 1$ **then**
      `Processor_Allocation`(T_i, v_i, p');
    endif;
  else
    $i_0 := i$;
    $W' := W_{T_i}$;
    while $(p' = 0)$ & $(i \neq d)$
      $i := i + 1$;
      $W' := W' + W_{T_i}$;
      $p' := \lfloor \frac{W'}{t_{\min}} \rfloor$;
    endwhile;
    allocate a processor to a group $\mathcal{G}$ of subtrees $\{T_{i_0}, \ldots, T_i\}$; run an efficient sequential algorithm to compute all the nodes in $\mathcal{G}$;
  endif
until $i = d$.

Lemma 1 *If the* `while` *loop in procedure* `Processor_Allocation` *is executed, then either $p' = 1$ or $p' = 0$ after the loop termination, and $p' = 0$ only if $i = d$.*

Proof The proof for the case of $p' = 0$ is straightforward according to the `while` condition. Here we only show $p' = 1$ by considering $k = i - i_0 + 1$ subtrees are included after the `while` loop termination.

When $k = 1$, $p' = \lfloor W'/t_{\min} \rfloor = \lfloor (W_{T_{i_0}} + W_{T_{i_0+1}})/t_{\min} \rfloor \leq \lfloor 2W_{T_{i_0}}/t_{\min} \rfloor$ since $W_{T_{i_0}} \geq W_{T_{i_0+1}}$ by the initial assumption. Then we can derive that $0 < W'/t_{\min} < 2$ since $0 < W_{T_{i_0}}/t_{\min} < 1$. Therefore, $p' = \lfloor W'/t_{\min} \rfloor = 1$ because $p' \neq 0$.

Now consider the case of $k > 1$. Clearly $0 < \sum_{j=i_0}^{i-1} W_{T_j}/t_{\min} < 1$ by the `while` condition, and $W_{T_i} \leq W_{T_{i-1}}$. Then $0 < W'/t_{\min} = \sum_{j=i_0}^{i} W_{T_j}/t_{\min} < 2\sum_{j=i_0}^{i-1} W_{T_j}/t_{\min} < 2$, i.e., $0 \leq p' = \lfloor W'/t_{\min} \rfloor = 1$. Since $p' \neq 0$, so, $p' = 1$. $\Box$

Compared with `Algorithm 1`, in some sense `Algorithm 2` optimizes the total operations for the data cube computation globally. Moreover, this algorithm almost doesn't need any global synchronization (a synchronization by all processors involved), thereby reduces the entire response time. However, the processor allocation in this algorithm is more complicated. The efficiency of the algorithm is based on the size estimation for each node in $\mathcal{L}$, whereas the accuracy of the estimation depends on what fraction of the base table is used for samples, as well as which estimation method is used. If we only use a very small sample and the time used for this estimation takes a very small proportion of the entire computation, this may lead to an inaccurate estimation for the total operations for the data cube computation. As a result, the response time for the computation may be much worse than expected.

3.3 A matching-based parallel algorithm

In `Algorithm 2` we assumed that the size estimation of the nodes in $\mathcal{L}$ and the weight assignment of the edges in $\mathcal{L}$ are accurate. If not, the weighted sum of the directed minimum spanning tree T in $\mathcal{L}$ does not represent the real total optimal operations for the data cube computation. Therefore, the actual response time based on T is not the best one because the response time fully depends on the weight of T as well the processor allocation to the nodes in T. Actually `Algorithm 2` first gives a *static* estimate of the total operations for the data cube computation, then schedules processors to the nodes in T for implementing the computation. In the following we present an algorithm which gives a more accurate size estimation to each node in $\mathcal{L}$. The algorithm is inspired by the sequential algorithm due to [14].

Their algorithm proceeds level-by-level, starting from level $l = 1$ to level $l = n + 1$. For each level l it finds the best way of computing the nodes at level $l + 1$ from level l by reducing the problem to a minimum weighted matching problem on a weighted bipartite graph. Here we should stress that in the sequential environment, the subsequent computation for the nodes in the matching is easy after finding the matching. However, in the parallel environment the computation for the nodes is more complicated which reflects issues such as how to allocate the processors to the nodes in the matching, and how to determine the number of processors which should be assigned to a node, etc. In the following we introduce a processor allocation strategy for this purpose. We refer to the processor allocation and node computation as *the second phase* of the proposed algorithm. The detailed algorithm is outlined as follows.

Initially we sort the base relation table by some order of its attributes at level one (root). Now we assume that all group-by aggregations at level l are available and these group-by aggregations are sorted in some order by their attributes. Initially $l = 1$.

Following [14], a bipartite graph $G = (X, Y, E)$ is constructed as follows: X is the set of nodes at level l, and Y is the set of nodes at level $l + 1$. There is a directed edge from $x \in X$ to $y \in Y$ if y can be generated from x, and the weight $C_{scan}(x, y)$ associated with this edge is the size of x. We then construct another weighted bipartite graph $G' = (X' \cup X, Y, E' \cup E)$ from $G(X, Y, E)$ as follows: $X' = \{x_1, x_2, \dots, x_{n-l} \mid x \in X\}$, i.e., there are $n - 1$ new nodes in G' for each node $x \in X$. $\langle x_i, y\rangle \in E'$ if $\langle x, y\rangle \in E$, and it is assigned the cost $C_{sort}(x, y)$ of sorting x. The purpose of creating $n - 1$ corresponding edges of each edge (x, y) in G is that each node $y \in Y$ except x can be derived from the other $n - 1$ nodes.

Having the weighted bipartite graph G', find a minimum weighted matching $\mathcal{M}$ of G' which corresponds to a scheduling for computing the nodes in Y. An edge $\langle x_i, y\rangle$ in $\mathcal{M}$ means that y can be obtained by sorting x, and an edge $\langle x, y\rangle$ in $\mathcal{M}$ means that y can be obtained by scanning x without sorting x. Then the total operations for the node computation related to $\mathcal{M}$ is

$$W_{\mathcal{M}} = \sum_{\langle u,v\rangle \in \mathcal{M}, u \in X', \text{ and } v \in Y} C_{sort}(u, v) + \sum_{\langle u,v\rangle \in \mathcal{M}, u \in X, \text{ and } v \in Y} C_{scan}(u, v) \quad (6)$$

From $\mathcal{M}$, a weighted forest $\mathcal{F} = (V', E')$ of trees is constructed as follows. $V' \subseteq V$, $E' \subseteq E$ and $\mathcal{L} = (V, E)$. Let $T \in \mathcal{F}$, if $\langle x, y\rangle \in \mathcal{M}$, then x is the root of T and y is a son of x and the weight of this edge is $C_{scan}(x, y)$; if $\langle x_i, y\rangle \in \mathcal{M}$, then x is the root of a tree and y is a son of x and the weight of this edge is $C_{sort}(x, y)$.

In the following we consider the processor allocation for the node computation at level $l+1$ by $\mathcal{M}$. Let $t_{\mathcal{M}}$ be the time lower bound for computing all nodes at level $l+1$, then

$$t_{\mathcal{M}} \quad = \quad a \cdot W_{\mathcal{M}}/p + b \cdot p \qquad (7)$$

The remaining is to allocate processors to the trees in $\mathcal{F}$. We adapt the approach as following. We scan the elements in $\mathcal{F}$ one by one. Let $T \in \mathcal{F}$ be the current scanning element and let W_T be the weight of T. Then, the number of processors is assigned to T as follows: if $\lfloor W_T/t_{\mathcal{M}} \rfloor \neq 0$, we assign $\lfloor W_T/t_{\mathcal{M}} \rfloor$ processors to T, and remove T from $\mathcal{F}$; otherwise, we merge T with the next tree T', and repeat the above procedure until $\mathcal{F} = \emptyset$.

After finishing the processor allocation, the processors assigned to a tree or a subset of trees are dedicated to the computation of those (leaf) nodes in the tree or the subset of trees. In summary, the detailed algorithm is presented as follows.

Algorithm 3:
$l := 1$; /* Starting from level $l = 1$ */
repeat
1. Construct the bipartite graph $G(X, Y, E)$.
2. Construct the bipartite graph
   $G'(X' \cup X, Y, E' \cup E)$.
3. Find a minimum weighted matching $\mathcal{M}$ in G'.
4. Construct a weighted forest $\mathcal{F}$ from $\mathcal{M}$.
5. Allocate processors to the trees in $\mathcal{F}$.
6. Proceed group-by aggregations
   by each cluster of processors independently.
   $l := l + 1$;

until $l = n + 1$.

`Algorithm 3` can compute the size of every node accurately, but the cost incurred for this purpose means that a synchronization must be carried out at each level in order to obtain the computation cost for the nodes at the next level. Meanwhile, the estimated running time $t_{\mathcal{M}}$ for processor allocation is also important. A rough estimation of $t_{\mathcal{M}}$ will result in a considerable synchronization overhead. Finally, it should be mentioned that the local optimization carried out at each level does not mean global optimization with regard to the total operations for the data cube computation.

4 A Hybrid Parallel Algorithm

In this section we introduce a hybrid algorithm for the data cube computation. From `Algorithm 3` we can see that at the end of generating all group-by aggregations at each level, a synchronization is used to collect the size of each node at that level. As we know, if the estimated time $t_{\mathcal{M}}$ is correct, the overhead of the synchronization is not too much. Otherwise, we spend a lot of time for synchronization. In this section we combine `Algorithms 2` and `3` by presenting a hybrid parallel algorithm to overcome some disadvantages incurred by each of them individually.

The basic idea is that we partition the lattice $\mathcal{L}$ into several subgraphs horizontally. The synchronization is only carried out at the bottom levels of these subgraphs. Let H be such a subgraph with the starting level l and the ending level $l+h$. Assume that the computation for all nodes, as well their sizes at level l, has been done. Then for every node v at level l, we use its information and its size to estimate the sizes of those nodes in H which are reachable from v. We assign an estimated size for every node in H by combining all estimates to it, and as a result, H becomes a node-weighted graph. We then find a directed minimum spanning forest of a weighted graph H, and allocate the processors to the trees in the forest according to their weights. Finally we use `Algorithm 2` to compute the nodes in each subtree independently by using the processors allocated to it. In the following we illustrate these in detail.

Let N_l be the set of nodes at level l in $\mathcal{L}$. A subgraph H of $\mathcal{L}$ is an induced subgraph by the nodes from level l to level $l+h$, i.e., $H = (V_1, (V_1 \times V_1) \cap E)$ where $V_1 = \cup_{i=l}^{l+h} N_i$ and $\mathcal{L} = (V, E)$.

Define $den_H(v) = \{u \mid u \in V_1,\ \mathit{and\ u\ is\ reachable\ from\ v}\}$ for every $v \in N_l$; and $source_H(u) = \{v \mid v \in N_l,\ \mathit{and\ u\ is\ reachable\ from\ v}\}$ for every $u \in V_1 - N_l$. Now for every $v \in N_l$, it generates an estimated size $S_H(v, u)$ for every node $u \in den_H(v)$. As results, the estimated size of a node $u \in V_1 - N_l$ in H is

$$S_H(u) \quad = \quad \lceil \frac{\sum_{v \in N_i} S_H(v, u)}{|source_H(u)|} \rceil \qquad (8)$$

In order to reduce the time for estimating the sizes of nodes, the estimation is actually carried out during generating the nodes in N_l.

A directed weighted graph H' from H then is obtained where $H' = (V_1 \cup \{s\}, (V_1 \times V_1) \cap E \cup \{\langle s, v\rangle \mid v \in N_l\})$, where s is a virtual node which can reach every other node in V_1. The weight associated with each edge in H' is defined as follows. For every edge $\langle s, v\rangle$, it is assigned weight zero; for every edge $\langle x, y\rangle$ with $x \neq s$, it is assigned weight $S_H(x)$.

Having H', find a directed minimum spanning tree $T_{H'}$ rooted at s in H'. After that, a forest $\mathcal{F}$ of trees is obtained by removing s and all its adjacent edges from $T_{H'}$. Now the second phase of `Algorithm 2` can be applied to $\mathcal{F}$, i.e, allocate processors to each tree in $\mathcal{F}$ and implement the computation for those nodes which are not the tree roots in $\mathcal{F}$. For the sake of integrity, we present the entire algorithm below.

Algorithm 4:
$l := 1$; /* Starting from level $l = 1$ */
repeat
1. Construct a subgraph H induced by the nodes from level l to level $l + h$;
2. Estimate the size of every node in H by the information of the nodes in N_l;
3. Construct the subgraph H' from H;
4. Find a directed minimum spanning tree $T_{H'}$ rooted at s in H';
5. Construct a weighted forest $\mathcal{F}$ from $T_{H'}$;
6. Allocate processors to the trees in $\mathcal{F}$;
7. Proceed group-bys by each cluster of processors independently;

$l := l + h$;
until $l = n$.

Compared with the two previous algorithms `Algorithms 2` and `3`, `Algorithm 4` reduces the number of synchronizations in `Algorithm 3`, and therefore reduces the entire response time. However, the size estimate of each node is not as accurate as the result we achieved in `Algorithm 3`. As a result, the total number of operations is not optimal. However this algorithm gives a much more accurate estimation of total operations than `Algorithm 2` because we use the actual information and the sizes of the nodes at level l to estimate the sizes of the nodes in the subsequent h levels.

Here we give a variant of the hybrid parallel algorithm. Let P be a directed path from the root to a node v in the lattice $\mathcal{L}$, and the node sequence in P be $v_0, v_1, v_2 \ldots, v_k = v$. It is obvious that $s(v_0) \geq s(v_1) \geq \ldots \geq s(v_k)$ where $s(v_k)$ is the actual size of v. From this observation, we have the following variant of the proposed algorithm. That is, at the first l levels, we use `Algorithm 3` to compute the cuboids level by level. After that, we repeatedly use the hybrid algorithm for a subgraph consisting of the nodes by h_i levels ($h_i \geq 1$), and h_i is an ascending sequence, i.e, $h_j > h_i$ if $j > i$.

5 A Generalized Algorithm

In the previous two sections we have considered computing all dimensional group-by aggregations for a base relation table, or computing all nodes in the lattice. In practice, not all dimensional aggregates are interesting to analysts and decision makers. Instead only some of them are required quite often. Therefore, we only need to compute some nodes in the lattice.

Recall that R is an n-attribute relation and $\mathcal{L} = (V, E)$ is the lattice of R. Assume that each directed edge $\langle u, v\rangle \in E$ is assigned a cost which is the estimated operation cost of generating v from u.

We now construct a weighted directed graph $\mathcal{L}^* = (V^*, E^*)$ from $\mathcal{L}$ as follows: $V^* := V$. For an edge $\langle u, v\rangle \in E^*$, it is assigned the same weight as the edge $\langle u, v\rangle \in E$ in $\mathcal{L}$. If there is a directed path from u to v in $\mathcal{L}$, then $\langle u, v\rangle \in E^*$, and the weight associated with this edge is the cost of generating v from u directly. Actually $\mathcal{L}^*$ is the transitive closure of $\mathcal{L}$.

Let $S \subset V$ be a proper subset of the nodes in $\mathcal{L}$ in which we are interested. The problem is to compute all group-by aggregations in S with minimizing the total operations for these aggregates. In order to optimize the total operations, we need to find a directed tree T_S rooted at R in $\mathcal{L}^*$ which contains all nodes in S such that the weighted sum of the edges in T_S is minimized. Clearly, T_S is a directed Steiner tree in $\mathcal{L}^*$. It is well known that the directed Steiner tree problem on a general directed graph is NP-complete [6], and even for this special Steiner tree problem, it is also NP-complete [14]. It is also well known that the minimum set cover (MSC) problem can be reduced to a directed Steiner tree problem in polynomial time, whereas [5] has shown that there is no polynomial approximation algorithm for the MSC problem which delivers a solution better than $(1-\epsilon')\ln q$ times optimum unless there is an $O(n^{\log\log n})$ time algorithm for NP-complete problems for any fixed ϵ' with $0 < \epsilon' < 1$, where q is the problem size. Therefore, we have

Lemma 2 *Given a directed DAG $G(V,E)$ ($\mathcal{L}$ is a special DAG) with $|V| = N$, and each edge is assigned a weight, let $S \subset V - \{s\}$ be subset of the nodes in V where s is the unique source and every node $v \in S$ is reachable from s. Then there is a polynomial approximation algorithm for finding an approximate directed Steiner tree rooted at s including all nodes in S such that the weight of the tree is within $O(|S|^{\epsilon})$ times optimum if the running time is $O((N|S|)^{1/\epsilon}/N + N^3)$, where $0 < \epsilon \leq 1$.*

Proof By the result of [5], it is unlikely that there is a polynomial algorithm for the directed Steiner tree problem which gives an approximation solution better than $(1-\epsilon')\ln N$ times the optimum. However, by using the algorithm due to Zelikovsky [16], the claim in the lemma is straightforward.□

Basically `Algorithm 2` can be applied here with minor modifications, i.e, replace the subroutine for finding a minimum directed spanning tree by the subroutine for finding an approximately Steiner tree T_S. Now we sketch the steps for computing all aggregates in S as follows.
Step 1. Construct the lattice $\mathcal{L}$ of R and assign the edges in $\mathcal{L}$ with weights.
Step 2. Construct $\mathcal{L}^*$ from $\mathcal{L}$ and assign weights to the edges in $\mathcal{L}^*$.
Step 3. Find an approximation Steiner tree T_S in $\mathcal{L}^*$, using the directed Steiner tree algorithm [16].
Step 4. Allocate processors to the nodes in T_S and carry out the aggregates for all corresponding nodes using the processors allocated to them. This can be done by using the second phase of `Algorithm 2`.

Actually the approach above can be applied to any DAG and not just the lattice only.

6 Conclusions

In this paper we have developed four parallel algorithms to compute the data cube efficiently in the parallel computational environment. The proposed algorithms introduce a novel processor scheduling policy and a non-trivial decomposition approach for the problem. Particularly, the hybrid algorithm has the best performance potential among the four proposed algorithms. Partial contents of this paper also appeared in [11].

Appendix: Procedure for estimating the size of group-bys. As suggested in [8], we first convert the raw data by mapping each attribute value into a unique integer before starting to compute the cube. We can use this data conversion step to get the number of distinct values, D_i for each attribute, A_i of the cube. A starting estimate of the size of group-by $A_1A_2\ldots A_k$ consisting of attributes 1 through k can then be taken to be $\min\{D_1 \times D_2 \times \ldots \times D_k, N'\}$ where N' is the number of the tuples in the raw table.

These initial estimates are then refined in the following ways. (1) Once a group-by is computed, then the estimated size of all group-bys derivable from it can be strictly smaller than this group-by. (2) Once we have covered two levels we can get better estimates as follows.

$$\frac{|ABCD|}{|ABC|} \leq \frac{|ABD|}{|AB|} \tag{9}$$

therefore,

$$|AB| \leq \frac{|ABD||ABC|}{|ABCD|} \tag{10}$$

References

[1] Agawal S., Agrawal R., Deshpande P. M., Gupta A., Naughton J. F., Ramakrishnan R. & Sarawagi S. (1996) On the computation of multidimensional aggregates. *Proc. of the 22nd VLDB Conf.*, Mumbay, India, p. 506-521.

[2] Bitton D., Boral H., et al (1983) Parallel algorithms for the execution of relational database operations. *ACM Trans. on Database Systems*, 8(3), p. 324–253.

[3] Codd E. F. (1993) Providing OLAP: an IT mandate. *Unpublished manuscript*, E. F Codd and Associates.

[4] Deshpande P. M., Agarwal S., Naughton J. F., Ramakrishnan R. (1996) Computation of multidimensional aggregates. *Technical Report*, No: 1314, Dept. of CS, Univ. of Wisconsin-Madison.

[5] Feige U. (1996) A threshold of $\ln n$ for approximating set cover. *Proc. of 28th ACM Symp. on Theory of Computing*, p. 314–318.

[6] Garey M. R. & Johnson D. S. (1979) *Computers and Intractability.* W. H. Freeman, San Francisco.

[7] Graefe G. (1993) Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2), p. 73–170.

[8] Gray J., Bosworth A., Layman A. & Prahesh H. (1995) Data cube: a relational aggregation operator generalizing group-by, cross-tab, and sub-total. *Microsoft Technical Report*, MSR-TR-95-22, 1995.

[9] Gupta H., Harinarayan V., Rajaraman A. & Ullman J. D. (1997) Index selection for OLAP. *Proc. of Int'l Conf. on Data Engineering*, Birmingham, UK, p. 208-219.

[10] Harinarayan V., Rajaraman A. & Ullman J. D. (1996) Implementing data cubes efficiently. *Proc. of the 1996 ACM-SIGMOD Conf.*, p. 205–216.

[11] Liang W. & Orlowska M. E. (1998) Computing multidimensional aggregates in parallel. *Proc. of the 1998 Int'l Conf. on Parallel and Distributed Systems*, Taiwan, IEEE Computer Society Press, p. 92–99.

[12] Mumick I. S., Quass D. & Mumick B. S. (1997) Maintenance of data cubes and summary tables in a warehouse. *Proc. of the 1997 ACM-SIGMOD Conf.*, p. 100–111.

[13] Ross K. A. & Srivastava D. (1997) Fast computation of sparse datacubes. *Proc. of the 23rd VLDB Conf.*, Athens, Greece, p. 116-125.

[14] Sarawagi S., Agrawal R. & Gupta A. (1996) On the computing the data cube. *Research Report*, IBM Almaden Research Center, San Jose, CA.

[15] Shatdal A. & Naughton J. F. (1995) Adaptive parallel aggregation algorithms. *Proc. of the 1995 ACM-SIGMOD Conf.*, p. 104–114.

[16] Zelikovsky A. (1997) A series of approximation algorithms for the acyclic directed Steiner tree problem. *Algorithmica*, vol.18, p. 99-110.

[17] Zhao Y., Deshpande P. M. & Naughton J. F. (1997) An array-based algorithm for simultaneous multidimensional aggregates. *Proc. of the 1997 ACM-SIGMOD Conf.*, p. 159–170.