Computing Partial Data Cubes for Parallel Data Warehousing Applications

Frank Dehne[1], Todd Eavis[2], and Andrew Rau-Chaplin[3]

[1] School of Computer Science
Carleton University, Ottawa, Canada K1S 5B6
frank@dehne.net, www.dehne.net
[2] Faculty of Computer Science
Dalhousie University, Halifax, NS, Canada B3H 1W5
eavis@cs.dal.ca
[3] Faculty of Computer Science
Dalhousie University, Halifax, NS, Canada B3H 1W5
arc@cs.dal.ca, www.cs.dal.ca/~arc
Corresponding Author.

Abstract. In this paper, we focus on an approach to *On-Line Analytical Processing* (OLAP) that is based on a database operator and data structure called the datacube. The *datacube* is a relational operator that is used to construct all possible views of a given data set. Efficient algorithms for computing the entire datacube — both sequentially and in parallel — have recently been proposed. However, due to space and time constraints, the assumption that all 2^d (where d = dimensions) views should be computed is often not valid in practice. As a result, algorithms for computing partial datacubes are required. In this paper, we describe a parallel algorithm for computing partial datacubes and provide preliminary experimental results based on an implementation in C and MPI.

1 Introduction

As databases and data warehouses grow ever bigger there is an increasing need to explore the use of parallelism for storage, manipulation, querying, and visualization tasks. In this paper, we focus on an approach to On-Line Analytical Processing (OLAP) that is based on a database operator and data structure called the datacube [4]. Datacubes are sets of pre-computed views of selected data that are formed by aggregating values across attribute combinations (a *group-by* in database terminology) as illistrated in Figure 1. A generated datacube on d attribute values can either be complete, that is, contain all of the 2^d possible views formed by attribute combinations, or partial, that is, contain only a subset of the 2^d possible views. Although the generation of complete and partial views is related, the latter is a significantly more difficult problem. Despite this difficulty, in practice it is important to be able to generate such partial datacubes because, for high dimensional data sets (i.e., between four and

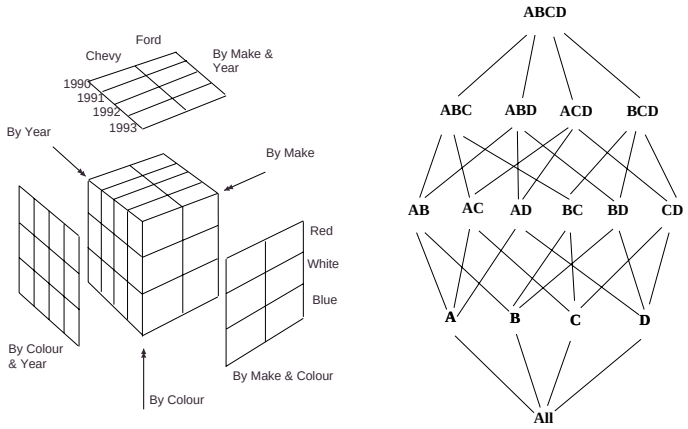

Fig. 1. An example 3 dimensional datacube and a 4 dimensional lattice. Lefthand side: An example three dimensional datacube concerning automobile data. Righthand side: The lattice corresponding to a four dimensional data cube with dimensions A, B, C and D. The lattice represents all possible attribute combinations and their relationships. The "all" node represents the aggregation of all records.

ten), a fully materialized datacube may be several hundred times larger than the original data set.

The datacube, which was introduced by Jim Gray et. al [4], has been extensively studied in the sequential setting [1, 2, 4–8] and has been shown to dramatically accelerate the visualization and query tasks associated with large information sets. To date the primary focus has been on algorithms for efficiently generating complete datacubes that reduce computation by sharing sort costs [1, 7], that minimize external memory sorting by partitioning the data into memory-size segments [2, 6], and that represent the views themselves as multi-dimensional arrays [4, 8]. The basis of most of these algorithms is the idea that it is cheaper to compute views from other views rather than from starting again with the original data set. For example, in Pipesort [7] the lattice is initially augmented with both estimates for the sizes of each view and cost values giving the cost of using a view to compute its children. Then a spanning tree of the lattice is computed by a level-by-level application of minimum bipartite matching. The resulting spanning tree represents an efficient "schedule" for building the actual datacube.

Relatively little work has been done on the more difficult problem of generating partial datacubes. Given a lattice and a set of selected views that are to be generated, the challenge is in deciding which view should be computed from which other view, in order to minimize the total cost of computing the datacube. In many cases computing intermediate views that are not in the selected set, but from which several views in the selected set can be computed cheaply,

will reduce the overall computation time. In [7], Sarawagi et al. suggest an approach based on augmenting the lattice with additional vertices (to represent all possible orderings of each view's attributes) and additional edges (to represent all relationships between views). Then a Minimum Steiner Tree approximation algorithm is run to identify some number of "intermediate" nodes (or so-called Steiner points) that can be added to the selected subset to "best" reduce the overall cost. An approximation algorithm is used here because the optimal Minimum Steiner Tree is NP-Complete. The intermediate nodes introduced by this method are, of course, to be drawn from the non-selected nodes in the original lattice. By adding these additional nodes, the cost of computing the selected nodes is actually reduced. Although theoretically neat this approach is not effective in practice. The problem is that the augmented lattice has far too many vertices and edges to be efficiently handled. For example, in a 6 dimensional datacube the number of vertices and edges in the augmented lattice increases by a factor of 326 and 8684 respectively, while for a 6 dimensional datacube the number of vertices and edges increase by a factor of 428 and 701,346 respectively. A 9 dimensional datacube has more than 2,000,000,000 edges. Another approach is clearly necessary.

In this paper we describe a new approach to efficiently generate partial datacubes based on a parallel version of Pipesort [3] and a new greedy algorithm to select intermediate views. We also present initial experimental results based on an implementation of our algorithm in C and MPI. The experimental results are encouraging in that they show an average reduction in computing a partial datacube of 82% over computation directly from the raw data. This reduction applies to both the sequential and parallel cases. Furthermore, the parallel version of our algorithm appears to achieve linear speedup in experiments on an eight node cluster.

2 Generating Partial Datacubes in Parallel

In the following we present a high-level outline of our coarse grained parallel partial datacube construction method. This method is based on sequential Pipesort [7] and a parallel version of Pipesort described in [3]. The key to going from these methods for computing *complete* datacubes to a method for computing *partial* datacubes is Step 2 of the following algorithm - the greedy method for computing an efficient schedule tree for the partial datacube generation problem.

A parallel algorithm for generating partial datacubes

1. **Build a Model:** Construct a lattice for all 2^d views and estimate the size of each of the views in the lattice. To determine the cost of using a given view to directly compute its children, use its estimated size to calculate (a) the cost of scanning the view and (b) the cost of sorting it.
2. **Compute a schedule tree using the model:** Using the bipartite matching technique presented in Pipesort [7], reduce the lattice to a spanning tree

that identifies the appropriate set of prefix-ordered sort paths. Prune the spanning tree to remove any nodes that cannot possibly be used to compute any of the selected nodes. Run a greedy algorithm using the pruned tree to identify useful intermediate nodes. The tree built by the greedy algorithm contains only selected nodes and intermediate nodes and is called the schedule tree as it describes which views are best computed from which other views.

3. **Load balance and distribute the work:** Partition the schedule tree into $s \times p$ sub-trees (s = oversampling ratio). Distribute the sub-trees over the p compute nodes. On each node use the sequential Pipesort algorithm to build the set of local views.

Given that finding the optimal schedule tree is NP-Complete[4], we need to find a method that takes a manageable amount of time to find a reasonable schedule. In computing the schedule tree we propose starting from the spanning tree that is derived from Pipesort. Clearly there are many other approaches that could be taken. We chose this approach for our initial try at generating partial cubes because the Pipesort tree has proven to be effective in the generation of complete datacubes and therefore appears to be a good starting point for a schedule for partial datacubes. This choice is indeed supported by our experimental findings.

In the following sections we will describe exactly how the Pipesort tree is pruned, as well as the greedy algorithm for selecting intermediate nodes/views. For a description of how the model is built and the details of the load balancing algorithm see [3].

The Pruning Algorithm Before passing the Pipesort tree to the greedy algorithm, we want to ensure that is has been pruned of any unnecessary nodes. Quite simply, we remove any node from the tree whose attributes are not a superset of at least one selected node. The pseudo code can be written as follows:

```
Input: Spanning tree T and Subset S
Output: Pruned (spanning) tree T

for every node i in T - S
    for all nodes j of S
        if there is no node j whose attributes are a
           subset of the attributes of i
            delete node i from T
```

The operation of this simple quadratic time algorithm is illustrated in Figure 2.

The Greedy Algorithm The greedy algorithm takes as input a spanning tree T of the lattice that has been pruned and a set S of selected nodes representing those views to be materialized as part of the partial datacube. The algorithm

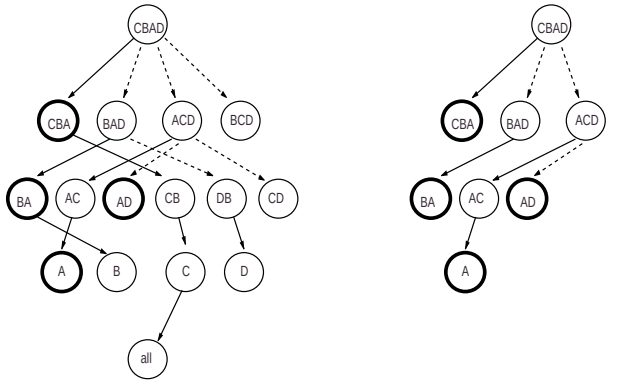

Fig. 2. Graph Pruning. Lefthand side: Spanning tree of the lattice as created by Pipesort with selected nodes in bold. Righthand side: Pruned tree.

begins by assuming that its output, the schedule tree T', will consist only of the selected nodes organized into a tree based on their relative positions in T. In other words, if a selected node a is a descendant of a selected node b in T, the same relationship is true in the initial schedule tree T'. The algorithm then repetitively looks for intermediate nodes that reduce the total cost of the schedule. At each step, the node from T that most reduces the total cost of T' is added. This process continues until there are no nodes which provide a cost improvement. Figure 3 shows an example of an initial schedule tree T' for a five dimensional cube with attributes A, B, C,D, and E and selected views A, AB, BC, CD, DE, and DAE, as well as a possible final schedule tree, T'.

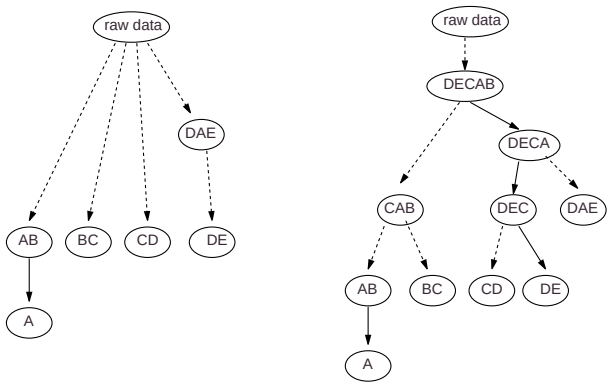

Fig. 3. The Schedule tree. Lefthand side: The initial schedule tree T' containing only selected nodes. Righthand side: An example final schedule tree T' containing both selected and intermediate nodes.

The pseudo code for the greedy algorithm is as follows:

```
Input: Spanning tree T and Subset S
Output: Schedule tree T

Initialize T with nodes of S
    each node of S is connected to its immediate predecessor in S
    edges are weighted accordingly

while global_benefit >= 0
    for each node i in T - S
        /* compute the benefit of including node i */
        for every node j in S
              if attributes of j are a prefix of i
                local_benefit += current_cost of j  - cost of scanning i
             else if attributes are a subset of i
                local_benefit += current_cost of j  - cost of resorting i
        local_benefit -= cost of building i
        if local_benefit > global_benefit
              global_benefit = local_benefit
              best node = i

    if global_benefit > 0
        add node i to T
```

3 Experimental Evaluation

Our preliminary results indicate that our new greedy approach significantly reduces the time required to compute a partial datacube in parallel. In this section we examine experimentally two aspects of our algorithm: 1) how well does the greedy algorithm reduce the total cost of building the partial datacube, and 2) what speedups are observed in practice.

We first examine the reduction in total cost. Table 1 provides the results for arbitrarily selected partial datacubes in five, seven, and nine dimensions. Each result represents the average over five experiments. In each row we see three graph costs. The first is the cost of computing each view in the partial datacube directly from the raw data set. The second cost is that obtained by our algorithm without the addition of intermediate nodes. Finally, the third column shows the cost obtained by our algorithm when utilizing intermediate nodes. Columns four and five show the percentage reductions in cost our algorithm obtains. Note that our algorithm for generating schedules for the computation of partial datacubes reduces the cost of such computation by between 77% and 85% over a range of test cases. It appears that the algorithm works best when the number of selected views is not too small. This is what one might expect given that when there are only a small number of selected views, there is little to be gained by introducing

Dim.	Partial datacube	(1) Base Cost	(2) Our Cost with no intermediate nodes	(3) Our Cost with intermediate nodes	1 vs 2	1 vs 3
5	8/32	35,898,852	23,935,211	6,203,648	44%	83%
7	15/128	95,759,942	36,353,947	14,139,249	62%	85%
9	25/512	119,813,710	49,372,844	28,372,844	59%	77%

Table 1. Cost reductions in five, seven, and nine dimensions.

intermediate nodes. More extensive experimental results will be reported on in the final version of this paper.

Figure 3 provides a graphical illustration of the algorithm's benefit. The image depicts a "before and after" scenario for a five dimensional lattice and associated partial datacube (this was an actual test case). On the left we see a spanning tree containing only the selected nodes (constructed during the initialization process in the greedy algorithm). On the right we have the final result - a new spanning tree with four additional nodes. In this case, the tree was reduced in size from 32,354,450 to 5,567,920 for an 83% reduction in total cost.

The following experiments were carried out on a very modest parallel hardware platform, consisting of a front-end machine plus 8 compute processors in a cluster. These processors were 166 MHZ Pentiums with 2G IDE hard drives and 32 MB of RAM. The processors were running LINUX and were connected via a 100 Mbit Fast Ethernet switch with full wire speed on all ports.

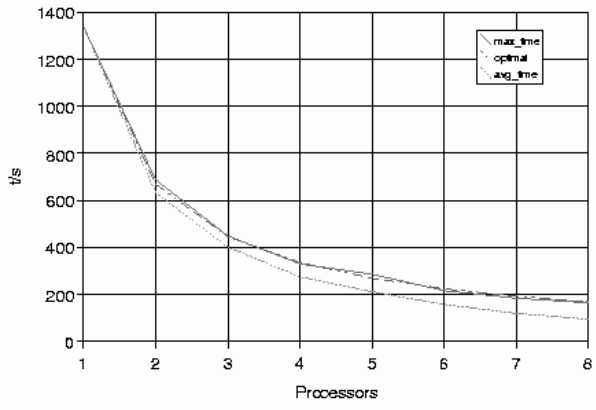

Fig. 4. Running Time In Seconds As A Function Of The Number Of Processors. (Fixed Parameters: Data Size = 1,000,000 Rows. Dimensions = 7. Experiments Per Data Point = 5.)

Figure 4 shows the running time observed as a function of the number of processors used. There are three curves shown. The *runtime* curve shows the time taken by the slowest processor (i.e., the processor that received the largest

workload). The second curve shows the *average time* taken by the processors. The time taken by the front-end machine to compute the model and schedule and distribute the work among the compute nodes was insignificant. The *theoretical optimum* curve shown in Figure 4 is the sequential Pipesort time divided by the number of processors used. Note that, these experiments were performed with schedule trees for complete datacubes rather than for partial datacubes but we expect the results to hold as these trees have very similar properties.

One can observe that the *runtime* obtained by our code and the *theoretical optimum* are essentially identical. Interestingly, the *average time* curve is always below the *theoretical optimum* curve, and even the *runtime* curve is sometimes below the *theoretical optimum* curve. One would have expected that the *runtime* curve would always be above the *theoretical optimum* curve. We believe that this *superlinear speedup* is caused by another effect which benefits our parallel method: improved I/O.

4 Conclusions

As data warehouses continue to grow in both size and complexity, so too does the need for effective parallel OLAP methods. In this paper we have discussed the design and implementation of an algorithm for the construction of partial datacubes. It was based on the construction of a schedule tree by a greedy algorithm that identifies additional intermediate nodes/views whose computation reduces the time to compute the partial datacube. Our preliminary results are very encouraging and we are currently investigating other related approaches.

References

1. S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. *Proceedings of the 22nd International VLDB Conference*, pages 506–521, 1996.
2. K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. *Proceedings of the 1999 ACM SIGMOD Conference*, pages 359–370, 1999.
3. F. Dehne, T. Eavis, S. Hambrusch, and A. Rau-Chaplin. Parallelizing the datacube. *International Conference on Database Theory*, 2001.
4. J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *J. Data Mining and Knowledge Discovery*, 1(1):29–53, April 1997.
5. V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing data cubes. *Proceedings of the 1996 ACM SIGMOD Conference*, pages 205–216, 1996.
6. K. Ross and D. Srivastava. Fast computation of sparse data cubes. *Proceedings of the 23rd VLDB Conference*, pages 116–125, 1997.
7. S. Sarawagi, R. Agrawal, and A.Gupta. On computing the data cube. Technical Report RJ10026, IBM Almaden Research Center, San Jose, California, 1996.
8. Y. Zhao, P. Deshpande, and J. Naughton. An array-based algorithm for simultaneous multi-dimensional aggregates. *Proceedings of the 1997 ACM SIGMOD Conference*, pages 159–170, 1997.