Top-Down Computation Of Partial ROLAP Data Cubes [*]

Frank Dehne
School of Computer Science
Carleton University
Ottawa, Canada K1S 5B6
http://www.dehne.net

Todd Eavis and Andrew Rau-Chaplin
Faculty of Computer Science
Dalhousie University
Halifax, Canada B3H 1W5
http://www.cs.dal.ca/∼eavis
http://www.cs.dal.ca/∼arc

Abstract

The precomputation of the different summary views of a data cube is critical to improving the response time of data cube queries for On-Line Analytical Processing (OLAP). The computation of the full data cube, representing all 2^d views, has been studied extensively. However, the full cube is often too large to be computed and stored, and for some applications all views are not even required. Hence, it is important to provide efficient methods for the computation of *partial* data cubes consisting of an arbitrary, user selected, subset of the 2^d possible views. In this paper, we study the top-down computation of partial ROLAP data cubes. We present both sequential and parallel methods for top-down partial data cube construction. Our experimental results indicate close to linear performance improvement for partial data cube computation. For example, when selecting 50% of the views our method requires only 55% of the time required to build the full cube, and when selecting 75% of the views our method requires just 82% of the full cube time.

1 Introduction

The Problem: Generating the *Partial* Data Cube

The precomputation of the different summary views (group-bys) of a data cube is critical to improving the response time of data cube queries for On-Line Analytical Processing (OLAP) [9]. Numerous solutions for generating the entire data cube (i.e. all 2^d views) have been proposed. See, for example, the sequential and parallel solutions for building the entire data cube presented in [10, 17, 1, 20, 4] and [5, 8, 14, 12, 19], respectively. In contrast, this paper studies the problem of computing the *partial* data cube. That is, given a relation R of size n and dimension d as well as an arbitrary *subset* S of the set of all 2^d possible view identifies, we wish to compute the subset of views identified in S.

Despite the practical significance of partial cube generation for OLAP systems, little algorithmic work has been presented in the literature. In [4], Beyer and Ramakrishnan present a bottom-up technique for the computation of ROLAP Iceberg-cubes. Though primarily designed for full cube computation — where it is quite efficient — the algorithm can be extended to support partial cube generation. Specifically, if end users *only* require views up to a specific number of dimensions, the BUC technique can be utilized. Because the algorithm is not particularly efficient at computing low dimensional views, however, it is unlikely to be effective in practical partial cube environments such as data visualization. In contrast, the algorithms presented in the remainder of this paper allow for the specification of arbitrary subsets and can also support efficient computation for low dimensional environments.

Summary of Results

[*]Research partially supported by the Natural Sciences and Engineering Research Council of Canada.

Data cube methods can be categorized as follows [15]: (1) There are two standard datacube representations, MOLAP (multi-dimensional array) and ROLAP (set of relational tables). (2) Data cube construction methods can be grouped into top-down methods (aimed at dense relations) and bottom-up methods (aimed at sparse relations). In this paper, we study the top-down computation of *partial* ROLAP data cubes for both the sequential and parallel setting.

We present two methods, Tree_Partial_Cube(S, PC) and Lattice_Partial_ Cube(S, PC). Both methods create a schedule tree T to guide top-down data cube construction. In the former case, T is a subtree of the Pipesort schedule tree [1] whereas in the latter case, T is a subgraph of the lattice L. The heart of our algorithm is a method Partial_Cube_Schedule(S, G, T) which builds the schedule tree T from a guiding graph, G (pruned Pipesort tree). It first organizes the nodes of S into a tree of minimum total cost, using a greedy approach. Then, it adds intermediate nodes which are not in the selected set S but have the property of further minimizing the total cost. In Section 3, we show how the above methods can also be parallelized for a p processor shared disk multiprocessor, like the SunFire 6800 [13].

We have implemented our sequential and parallel methods and tested them on a SunFire 6800 [13]. In comparison to previous top-down full data cube methods, we observed that our method has the same performance for full cube construction but has much improved, close to linear, performance for partial cube construction. Our experimental results indicate, for example, that when selecting 50% of the views our method requires only 55% of the time required to build the full cube, and when selecting 75% of the views our method requires just 82% of the full cube time. In fact, Lattice_Partial_Cube can be used as a general purpose replacement for Pipesort, one that achieves equivalent performance in the generation of full cubes and is, in addition, capable of efficiently generating partial cube. Furthermore, our method is considerably easier to implement than Pipesort because it does not require minimum cost bipartite graph matching.

Comparison With Previous Results

For top-down partial cube construction the central problem is how to build a schedule tree T of minimum cost that connects 1) all the selected views (i.e. S), and 2) some intermediate nodes (views) chosen in order to reduce the total cost. An approach to this problem based on Pipesort was presented in [1]. Recall that Pipesort builds a schedule tree by proceeding level by level through the lattice and building minimum cost bipartite matchings between levels. Since, the schedule tree for a partial cube may require edges between nodes at arbitrary levels of the lattice the authors in [1] suggest augmenting the lattice with Steiner vertices and edges representing all possible orderings of the attributes of all views and edges between all vertices where the attributes of one vertex are a prefix of the attributes of the other. The authors in [1] then apply a minimum Steiner tree approximation algorithm to the augmented lattice in order to create a schedule tree. The main problem with this approach, besides the minimum Steiner tree problem being NP-complete, is that the augmented lattice can become extraordinarily large. The number of vertices and edges in the original lattice L are $\sum_{k=0}^{d} \begin{pmatrix} d \\ k \end{pmatrix}$ and $\sum_{k=1}^{d} \begin{pmatrix} d \\ k \end{pmatrix} k$, respectively, while the number of vertices and edges in the augmented lattice with Steiner vertices and edges are $\sum_{k=0}^{d} \begin{pmatrix} d \\ k \end{pmatrix} k! + |S|$ and $\sum_{k=1}^{d} \left[\begin{pmatrix} d \\ k \end{pmatrix} k! \sum_{j=1}^{k} \frac{k!}{(j-1)!} \right] + |S|$, respectively. The number of Steiner edges is greater than 79,000,000 for $d = 7$ and reaches almost 40 trillion for $d = 10$. This makes such an approach impractical for relations with more than just a very small number of dimensions. These examples suggest that, in order to handle real life data sets, it is important to find approaches that do not require Steiner vertices and edges in the lattice. The main contribution of this paper is to provide such a method.

2 Sequential Partial Data Cubes

For a given set S of selected view identifiers (i.e. sets of dimensions), we wish to create a partial cube PC containing the views identified in S. The main task is to create a *schedule tree* T which contains all views of S plus some additional intermediate views such that the total cost for computing all of these views is minimized. A *schedule tree* T is a tree where the nodes represent views and edge (u, v) from parent u to child v indicate that v is created from u. Each edge (u, v) is labelled "scan" or "sort" indicating that v is created via a "scan" or "sort", respectively.

We present two methods, Tree_Partial_Cube(S, PC) and Lattice_Partial_Cube(S, PC), which both create a schedule tree T that contains the views in S, and then build the partial data cube PC according to the schedule implied by T. As a preprocessing step, we compute the lattice, L, of the 2^d possible view indentifiers [9] and use a storage estimator [6, 18] to estimate the approximate sizes of the views.

Procedure 1 Tree_Partial_Cube(S, PC)

```
/* Input: set of selected group-bys, S. Output: partial data cube, PC. Variables: A schedule tree T representing S with added intermediate nodes and scan/sort relationships. */
(1) Compute the Pipesort spanning tree of the lattice $L$ and prune it by deleting all nodes which have no descendent in S. Let G denote the result.
(2) Partial_Cube_Schedule(S, G, T)
(3) Fix_Pipelines(T)
(4) Build the partial data cube PC according to the schedule tree T.
```

Procedure 2 Lattice_Partial_Cube(S, PC)

```
/* Input: set of selected group-bys, S. Output: partial data cube, PC. Variables: A schedule tree T representing S with added intermediate nodes and scan/sort relationships. */
(1) Prune all nodes in the lattice L which have no descendent in S. Let G denote the result.
(2) Partial_Cube_Schedule(S, G, T)
(3) Establish_Attribute_Orderings(T)
(4) Build the partial data cube PC according to the schedule tree T.
```

The difference between the two methods is that, in Tree_Partial_Cube(S, PC) the schedule tree T is a subgraph of the Pipesort tree for complete cube construction whereas in Lattice_Partial_Cube(S, PC) the schedule tree T is a subgraph of the lattice. The heart of our algorithm is the method Partial_Cube_Schedule(S, G, T) which builds the schedule tree T. The guiding graph, G, captures the valid relationships between views. For Tree_Partial_Cube(S, PC), G is a subgraph of the pipesort tree and for Lattice_Partial_Cube (S, PC), G is a subgraph of the lattice. Each vertex of G has an additional label indicating the estimated size of the respective view.

For two adjacent nodes v, w in G we require an estimate of the cost involved to create view w from view v. Let scan_cost(v,w) and sort_cost(v,w) denote the cost estimates to create w from v via a scan or complete re-sort, respectively, including the I/O overhead involved. The estimates scan_cost(v,w) and sort_cost(v,w) are functions of the number of rows of v, $|v|$, where scan_cost(v,w) $= c_{disk}c_{dim}(d)|v|$ and sort_cost(v,w) $= c_{disk}c_{dim}(d)|v| + c_{sort}(d)|v|\log|v|$ for machine dependent values c_{disk}, $c_{dim}(d)$, and $c_{sort}(d)$. The constant c_{disk}, called disk constant, reflects the ratio between the cost of external disk access and local memory access. The function $c_{dim}(d) \leq d$ represents the increased cost associated with reading/writing d dimensional records in comparison to one dimensional records. The function $c_{sort}(d)$ reflects the overhead incurred when sorting d dimensional records in main memory.

Let mode(v,w) be "scan" for $v, w \in G$ if w can be created from v via a scan, and "sort" otherwise. Note that, if G is a subgraph of the pipesort tree, where the attribute ordering has been fixed, a node w can be created from v iff the attributes of w are a prefix of the attributes of v. If G is a subgraph of the lattice, where the attribute orderings have not been fixed, a node w can be created from v iff the attributes of w are a subset of the attributes of v. Let cost(v,w) be scan_cost(v,w) if mode(v,w) = "scan", and sort_cost(v,w) otherwise. Let RawDataSet denote the original data set and let parent(v, T) be the parent node of v in a given tree T.

The method Partial_Cube_Schedule(S, G, T) proceeds in two steps. In Step 1, it organizes the nodes of S into a tree of minimum total cost. In Step 2, it adds intermediate nodes (from $G-T$) to the tree to further minimize the total cost. Both steps make use of "plan" variables. A plan represents the best way for a given node v to be inserted into T. More precisely, a *plan* variable contains the following fields: (1) **node**: the node v considered to be inserted, (2) **parent**: the chosen parent of v, (3) **parent_mode**: the chosen mode (scan or sort) for computing v from its parent, (4) **scan_child**: the chosen child of v that is computed via scan, (5) **insertion_scan_child**: the chosen scan child of v in the case of scan insertion, (6) **sort_children**: the chosen children of v that are computed via sort, (7) **benefit**: the improvement in total cost obtained by inserting v.

For a plan variable P, the procedure Clear(P) sets P.benefit to $-\infty$ and all other fields to NIL.

Procedure 3 Partial_Cube_Schedule(S, G, T)

```
/* Input: set of selected group-bys, S, and a guiding graph G. Output: A schedule tree T representing S with added intermediate nodes and scan/sort relationships. Variables: CP (current plan) and BP (best plan) of type Plan. */
(1) /* Intialize T with nodes from S */
        S' = S; T = ∅
        WHILE S' not empty
            clear(BP)
            FOR every v ∈ S' DO
                clear(CP); CP.node = v
                Find_Best_Parent(T, G, CP)
                Find_Best_Children(T, G, CP)
                IF CP.benefit > BP.benefit THEN BP = CP
            update T according to BP
            remove BP.node from S'
(2) /* Add nodes from G-S to T as long as the total cost improves */
        REPEAT
            clear(BP)
            FOR every v ∈ G-T-{RawDataSet} DO
                clear(CP); CP.node = v
                Find_Best_Parent(T, G, CP)
                Find_Best_Children(T, G, CP)
                IF CP.benefit > BP.benefit THEN BP = CP
            IF BP.benefit > 0 THEN add BP.node to T and update T according to BP
        UNTIL BP.benefit <= 0
```

Both, Step 1 and Step 2 of Partial_Cube_-Schedule(S, G, T) use the two methods Find_-Best_Parent(T, G, CP) and Find_Best_Children(T, G, CP). The method Find_Best_ Parent(T, G, CP) identifies for a given node v the least expensive node w in T from which v can be computed. We favor the lengthening of scan pipelines by considering first the cases where v is either added at the end of an existing pipeline or v is inserted into an existing pipeline. Otherwise we consider using a sort to create v as the start of a new pipeline. Note that, adding v to T creates a cost (negative benefit) in the first place and that the "real" benefit will follow from the improved computation of children of v.

Procedure 4 Find_Best_Parent(T, G, CP)

```
/* Input: current tree, T, and a guiding graph G. Output: sets the fields CP.parent, CP.parent_ mode and CP.benefit to represent best parent of CP.node. Variables: parents_scan_child. */
(1) /* Intialize best parent to RawDataSet */
        CP.parent = RawDataSet
        CP.benefit = 0 - cost(RawDataSet, CP.node)
        CP.parent_mode = mode(RawDataSet, CP.node)
(2) /* Improve best parent, if possible */
        FOR all w ∈ T - { RawDataSet } where the attributes of CP.node are a subset of the attributes of w DO
        /* Case 1: CP.node is added at the end of an existing pipeline */
            IF w has no scan child AND scan_-cost(w,CP.node) < abs(CP.benefit) THEN
                CP.parent = w
                CP.benefit = 0 - scan_cost(w,CP.node)
                CP.parent_mode = "scan"
        /* Case 2: CP.node is inserted into an existing pipeline */
            ELSE IF w has a scan child w' AND mode(w, CP.node) = "scan" AND mode(CP.node, w') = "scan" AND scan_cost(w,CP.node) < abs(CP.benefit) THEN
                CP.parent = w; CP.insertion_scan_child = w'
                CP.benefit = 0 - scan_cost(w,CP.node)
                CP.parent_mode = "scan"
        /* Case 3: CP.node is made the start of a new pipeline */
        ELSE IF sort_cost(w,CP.node) < abs(CP.benefit) THEN
            CP.parent = w
            CP.benefit = 0 - sort_cost(w,CP.node)
            CP.parent_mode = "sort"
```

The method Find_Best_Children(T, G, CP) identifies for a given node v the set of children that would create the largest benefit if they were created from v rather than their current parents in T. In Step 1, it finds the best scan child, either by the scan insertion indicated by Find_Best_-Parent(T, G, CP) or by comparing the potential benefit of all possible scan children. In Step 2, it finds all other potential children that lead to an improvement in total cost, i.e. can be better computed from v than from their current parent in T.

Procedure 5 Find_Best_Children(T, G, CP)

```
/* Input: current tree, T, and a guiding graph G. Output: sets the fields CP.scan_child, CP.sort_children and CP.benefit to represent best children of CP.node. Variable: best_scan_ child, best_scan_child_benefit. */
(1) /* Find either a scan insertion (Case 2) or the best scan child (i.e. the one with largest path cost), if one exists. */
    best_scan_child = nil; best_scan_child_benefit = $-\infty$
    IF CP.insertion_scan_child != nil THEN
        best_scan_child = CP.insertion_scan_child
    ELSE FOR all w $\in$ T where { the attributes of w are a subset of the attributes of CP.node } DO
        IF mode(parent(w, T), w) = "sort" THEN
            IF cost(parent(w,T),w) − cost(CP.node,w) > best_scan_child_benefit THEN
                best_scan_child = w
                best_scan_child_benefit = cost(parent(w,T),w) − cost(CP.node,w)
    IF best_scan_child != nil THEN
        CP.benefit += cost(parent(w,T),w) − cost(CP.node,w)
        CP.scan_child = best_scan_child
(2) /* Find other children with positive benefit */
    FOR all w $\in$ T where { the attributes of w are a subset of the attributes of CP.node
    AND w $\neq$ best_scan_child AND mode(parent(w,T),w)="sort" } DO
        IF cost(parent(w,T),w) > cost(CP.node,w) THEN
            CP.benefit += cost(parent(w,T),w) − cost(CP.node,w)
            CP.sort_children += w
```

This concludes the description of our method, Partial_Cube_Schedule(S, G, T). After Partial_-Cube_Schedule(S, G, T) has generated a schedule tree, both methods, Tree_Partial_ Cube(S, PC) and Lattice_Partial_Cube(S, PC), continue with a post-processing method Fix_Pipelines(T) and Establish_Attribute_Orderings(T), respectively.

The post-processing method Establish_-Attribute_Orderings(T) has the task of identifying pipes of possible scan orderings for Lattice_-Partial_Cube. Note that, while all edges in T have been identified as either "scan" or "sort" edges, the attribute orderings for the vertices, i.e. views, have yet to be established. The method Establish_Attribute_Orderings(T) identifies all leaves in the schedule tree T which are scan children. These leaves mark the bottoms of existing pipelines. For each such leaf x, a method Fix_Attributes(x) is called which recursively walks up the pipeline, starting at x. As the parent/child scan relationships are examined, the attribute order of the parent is modified to reflect the ordering of its child. For example, a pathway such as B — CB — CGB — DGBC would be re-ordered as B — BC — BCG — BCGD.

The post-processing method Fix_Pipelines(T), used in Tree_Partial_Cube, has the task of identifying nodes that have no scan child, create a scan child for such nodes, and fix the attribute orderings. Note that, since in Tree_Partial_Cube the guiding graph is a subgraph of the Pipesort tree for the entire cube, the scan child x of a node y in the guiding graph may not be in T and therefore y may not have a scan child at this point. The method Tree_Partial_Cube identifies all nodes y with at least one child but no scan child. For each such node y, one arbitrary child x is made it's scan child and Fix_Attributes(x) is invoked to correctly set the attribute orderings.

Following the construction of the schedule tree T both methods, Tree_Partial_Cube(S, PC) and Lattice_Partial_Cube(S, PC), construct the partial cube, PC, according to the "scan/sort" schedule given by T.

3 Parallel Partial Data Cubes

In this section we outline how to parallelize our partial cube generation methods for a p processor shared disk multiprocessor, like the SunFire 6800 [13]. The following methods Parallel_Tree_Partial_Cube(p, S, PC) and Parallel_Lattice_Partial_Cube(p, S, PC) describe parallel versions of Tree_Partial_Cube and Lattice_Partial_Cube, respectively. In both cases, our approach is to generate the schedule tree T using Partial_Cube_Schedule(S, G, T), partition T into subtrees representing workloads of equal size, and then distribute the workload over the p processors $P_1, \ldots, P_p$. The following procedures show the structure of our methods.

Procedure 6 Parallel_Tree_Partial_Cube(p, S, PC)

```
/* Input: number of processors, p, and set of selected group-bys, S. Output: partial data cube, PC. Variables: A schedule tree T representing S with added intermediate nodes and scan/sort relationships. */
(1) Processor $P_1$:
    • Compute the Pipesort spanning tree of the lattice $L$ and prune it by deleting all nodes which have no descendent in S. Let G denote the result.
    • Partial_Cube_Schedule(S, G, T)
    • Fix_Pipelines(T)
    • Tree_Partition(T, p, s, $\Sigma_1$, ..., $\Sigma_p$).
(2) On each processor $P_i$, in parallel:
    • Compute all group-bys in subset $\Sigma_i$ on processor $P_i$ according to the schedule in $T \cap \Sigma_i$.
```

Procedure 7 Parallel_Lattice_Partial_Cube(p, S, PC)

```
/* Input: number of processors, p, and set of selected group-bys, S. Output: partial data cube, PC. Variables: A schedule tree T representing S with added intermediate nodes and scan/sort relationships. */
(1) Processor $P_1$:
    • Prune all nodes in the lattice L which have no descendent in S. Let G denote the result.
    • Partial_Cube_Schedule(S, G, T)
    • Establish_Attribute_Orderings(T)
    • Tree_Partition(T, p, s, $\Sigma_1$, ..., $\Sigma_p$).
(2) On each processor $P_i$, in parallel:
    • Compute all group-bys in subset $\Sigma_i$ according to the schedule in $T \cap \Sigma_i$.
```

The challenge is how to partition T into subtrees representing workloads of equal size because the tree partitioning problem is known to be NP-complete. We apply a tree partitioning heuristic which we had previously developed in [5] for parallelizing the computation of the *full* data cube. This approximation method makes use of a related partitioning problem on trees for which efficient algorithms exist, the *min-max tree k-partitioning problem* [3, 7, 16]. Our tree partitioning heuristic developed in [5] adapts the algorithm in [3] to the partitioning of the schedule tree T. Note that, min-max k-partitioning does not necessarily result in a partitioning of T into subtrees representing *equal workload*. To achieve a better distribution of the workload we apply an over partitioning strategy: instead of partitioning the tree T into p subtrees, we partition it into $s \times p$ subtrees, where $s \in \{1, 2, 3\}$ is a chosen integer parameter. Then, we use a *"packing heuristic"* to determine which subtrees belong to which processors, assigning s subtrees to every processor. Our packing heuristic considers the weights of the subtrees and pairs subtrees by weights to control the number of subtrees. It consists of s matching phases in which the p largest subtrees (or groups of subtrees) and the p smallest subtrees (or groups of subtrees) are matched up. The above constitutes our method Tree_Partition(T, p, s, Σ_1, ..., Σ_p) which has as input the schedule tree, T, number of processors, p, and overpartitioning ratio, s, and creates as output p sets of trees, Σ_1, ..., Σ_p, where each set Σ_i contains the s subtrees of T which will be assigned to processor P_i. As shown in [5], an overpartitioning ratio of $s \leq 3$ is sufficient to obtain a good workload distribution.

4 Performance Evaluation

In this section we discuss the experimental examination of Tree_Partial_Cube, Lattice_ Partial_Cube, Parallel_Tree_Partial_Cube, and Parallel_Lattice_Partial_Cube. We first discuss our setup and methodology and then present the performance results obtained.

4.1 Experimental Setup and Methodology

We have implemented Tree_Partial_Cube, Lattice_Partial_Cube, Parallel_Tree_Partial_Cube, and Parallel_Lattice_Partial_Cube using C and the MPI communication library [2]. Most of the required graph algorithms, as well as data structures like hash tables and graph representations, were drawn from the LEDA library [11]. Our experimental platform consisted of a Sun Fire 6800 with 24x 750MHz (8 MB E-Cache) UltraSPARC-III processors, 24 GB of memory and a Sun Storedge T3 disk storage system. The operating system was Solaris 8 (HW 04/01) and we used Sun MPI-5.0 as our MPI platform.

All sequential times were measured as wall clock times in seconds, running on one processor of the Sun Fire 6800. All parallel times were measured as the wall clock time between the start of the first process and the termination of the last process. We will refer to the latter as *parallel wall clock time*. These times include all I/O. Furthermore, all wall clock times were measured with no other user except us on the Sun Fire 6800.

Without a partial cube algorithm available, there are essentially two possible approaches to build a partial cube: (1) build the full data cube and then return the selected views only, or (2) calculate each of the selected views by a separate sort of the raw data set, followed by a scan. Which of these two approaches is better depends essentially on the percentage of selected views. For a small number of selected views (less than 25%), the individuals sorts will often be faster, while building the full data cube is often faster when the percentage of selected views is high (more than 75%). The following method Simple_Partial_Cube(S, PC), which always selects the faster of these two approaches, will be used as a "baseline" against which our algorithms Tree_Partial_Cube and Lattice_Partial_Cube will be compared. Note that, in the remainder of this section, the wall clock time for Simple_Partial_Cube(S, PC) will be determined by simply running both approaches and selecting the wall clock time of the faster one.

Procedure 8 Simple_Partial_Cube(S, PC)

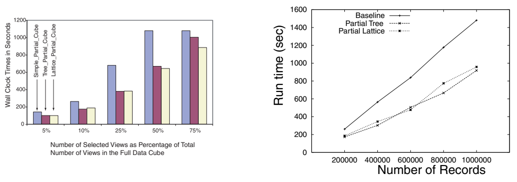
```
/* Input: set of selected group-bys, S. Output: partial
data cube, PC.*/
    Build the partial data cube, PC, for the set of selected
    group-bys, S, by using either
    (1) Pipesort, or
    (2) an individual sort and scan of the raw data set
        for each view in S
    which ever is faster.
```

We implemented a data generation program which creates data sets of various sizes and dimensions, with various cardinalities for the individual dimensions and various data distributions (from uniform to skewed data created via ZIPF distributions [21]). In the remainder, unless otherwise stated, our data sets were generated with uniform distribution and mixed cardinalities, varying between 2 and 1000 for the different dimensions. In order to eliminate influence of the storage estimator used on the comparison between Tree_Partial_Cube, Lattice_Partial_Cube and Simple_Partial_Cube, we used precise storage sizes for the views generated. For each experiment where there was variance in running times due to variances in input data sets, multiple data sets were run and data points represent the average over those experiments.

4.2 Performance Results: Sequential Experiments

Figure 1a shows the running time observed for Simple_Partial_Cube, Tree_Partial_Cube and Lattice_Partial_Cube as a function of the percentage of views from the complete data cube that are selected at random and generated. The data sets consisted of 200,000 rows with 8 dimensions and mixed cardinalities, varying between 2 and 1000 for the different dimensions. We observe that our two new methods are a significant improvement over Simple_Partial_Cube. When up to 50% of views are selected, a reduction in time of between 30% and 45% is observed. Even when as many as 75% of the views are selected an improvement of 18% is observed. When up to 50% of the views are selected, the methods Tree_Partial_Cube and Lattice_Partial_Cube exhibit very similar performance. Beyond that point the Lattice_Partial_-

Cube method appears to provide better performance.



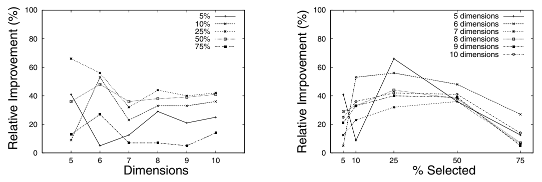
(a) (b)

Figure 1: Sequential wall clock time in seconds as a function of (a) the percentage of selected views (b) the data size.

Figure 1b shows the running time observed for Simple_Partial_Cube, Tree_Partial_Cube and Lattice_Partial_Cube as a function of the data size when 10% of the views in the complete data cube are selected at random and generated. The data sets range in size from 200,000 to 1,000,000 rows. Again we observe that our two new methods are a significant improvement over Simple_Partial_Cube. When only 10% of the views are selected, the new methods achieve an improvement of approximately 30%.

Figure 2a shows the relative improvement in running time observed for Tree_Partial_Cube with respect to Simple_Partial_Cube as a function of the dimensionality of the data sets when 5%, 10%, 25%, 50% or 75% of the views in the complete data cube are selected. We observe that when the dimensionality of the cube is low (i.e. 5 or 6) there is a lot of variation in the relative improvement. This is likely because in these cases there are only a small number of views in total (32 or 64) so that the addition of just a couple of intermediate views can have a very significant effect. As the number of dimensions grows, the curves become smoother and exhibit a consistent trend of slowly growing relative improvement.

Figure 2b presents the same data as Figure 2a in a different way. Here the relative improvement in running time observed for Tree_Partial_Cube with respect to Simple_Partial_Cube is presented as a function of the percentage of selected views when data sets with between 5 and 10 dimensions are considered. This figure highlights that regardless of dimensionality, the performance of Tree_Partial_Cube is best when between 10% and 50% of the views are selected. There is still some improvement below 10% and above 50% but it is relatively smaller, although not insignificant.

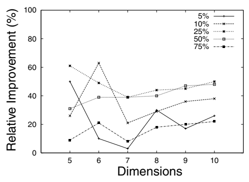


(a) (b)

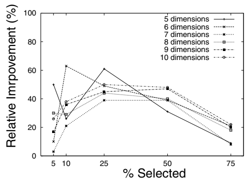
Figure 2: Relative improvement in wall clock time for sequential Tree_Partial_Cube W.R.T. Simple_Partial_Cube (a) as a function of the number of dimensions, for different percentages of selected views (b) as a function of the percentage of selected views, for different numbers of dimensions.

Figure 3a shows the relative improvement in running time observed for Lattice_Partial_ Cube with respect to Simple_Partial_Cube as a function of the dimensionality of the data sets while Figure 3b, using the same data, presents the relative improvement as a function of the percentage of selected views. It is interesting to observe how similar these curves are to the curves shown in Figure 2a and 2b. These results for Lattice_Partial_Cube are a slight improvement over the results for Tree_Partial_Cube but the general shape of the curves is the same. Again we can observe that beyond 7 dimensions the relative improvement is increasing as the dimensionality of the problem increases.

Figure 4a shows the running time observed for Simple_Partial_Cube, Tree_Partial_Cube and Lattice_Partial_Cube as a function of skew when 25% of the views in the complete data cube are selected. We note that skew is produced with the *Zipfian* power-law function, a technique commonly employed in the data cube literature [4, 18]. Here, we express the probability of encountering a particular value i in a given dimension d as $P_i \sim 1/i^{\alpha}$, where $1 \leq i \leq C_d$ and the probability P is normalized into the range



(a)

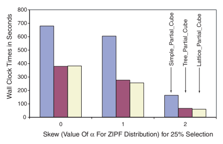


(b)

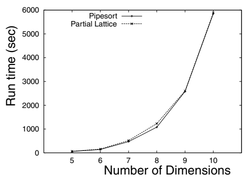
Figure 3: Relative improvement in wall clock time for sequential Lattice_Partial_Cube w.r.t. Simple_Partial_Cube (a) as a function of the number of dimensions, for different percentages of selected views (b)as a function of the percentage of selected views, for different numbers of dimensions.

$0 \ldots 1$. Effectively, the data set becomes significantly more skewed as α increases from zero (i.e., uniformly distributed).

Since data reduction in top-down generation methods increases with skew, the total time observed is expected to decrease with skew which is exactly what we observe in Figure 4b. One might expect that greedy methods like our Tree_Partial_Cube and Lattice_Partial_Cube might perform poorly in the presence of skew. However, the main observation of Figure 4b is that our methods appear to be robust in the presence of skew. In fact, they appear to do relatively better in situations of high skew.



(a)



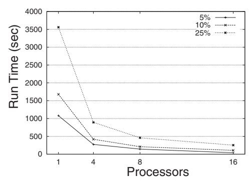
(b)

Figure 4: (a) Sequential wall clock time in seconds as a function of the skew (ZIPF) when 25% of the views are selected. (b) Computing the entire data cube (percentage of selected views = 100%). Sequential wall clock time in seconds as a function of the number of dimensions.

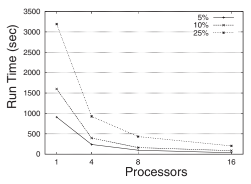
Although Lattice_Partial_Cube was designed for generating partial cubes it can of course also be used to generate full cubes by simply selecting all views. This is an interesting situation to study because in practice it would be very useful to have a single method (and code base) that could effectively generate an arbitrary percentage of the views of a complete data cube. Figure 4b shows the running time observed for Pipesort and Lattice_Partial_Cube as a function of the dimensionality of the data sets when the complete data cube is generated. Please observe how closely the run time of Lattice_Partial_Cube tracks the run time of Pipesort despite the fact that they are based on fundamentally different schedule tree generation methods. Note that, the two methods share the same code for the actual generation of views, given those schedule trees. The main observation that can be drawn from Figure 4b is that Lattice_Partial_Cube can be used as a general purpose replacement for Pipesort, one that achieves equivalent performance in the generation of full cubes and is in addition capable of efficiently generating partial cube.

4.3 Performance Results: Parallel Experiments

For our parallel methods Parallel_Tree_Partial_Cube and Parallel_Lattice_Partial_Cube we tested our methods on up to 16 processors of a SunFire 6800 and observed close to linear relative speedup.



(a)



(b)

Figure 5: Parallel wall clock time in seconds as a function of the number of processors, for different percentages of selected views for (a) Parallel_Tree_Partial_Cube(S, PC) and (b) Parallel_Lattice_Partial_Cube(S, PC).

Figures 5a and 5b show the parallel wall clock time in seconds for Parallel_Tree_Partial_ Cube and Parallel_Lattice_Partial_Cube, respectively, as a function of the number of processors when 5%, 10%, and 25%, of the views in the complete data cube are selected. (At time of submission, the curves for 50% and 75% were not available due to hardware problems. They will be included in the final version of this paper.) For both figures, the data sets consist of 1,000,000 rows with mixed cardinalities, varying between 2 and 1000 for the different dimensions. We observe that both, Parallel_Tree_Partial_Cube and Parallel_Lattice_Partial_Cube, achieve near linear relative speedup for up to 16 processors.

References

[1] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. pages 506–521, 1996.

[2] Argonne National Laboratory, http://www-unix.mcs.anl.gov/mpi/index.html. *The Message Passing Interface (MPI) standard*, 2003.

[3] R.I. Becker, Y. Perl, and S.R. Schach. A shifting algorithm for min-max tree partitioning. *J. ACM*, (29):58–67, 1982.

[4] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. In *Proc. of 1999 ACM SIGMOD Conference on Management of data*, pages 359–370, 1999.

[5] F. Dehne, T. Eavis, S. Hambrusch, and A. Rau-Chaplin. Parallelizing the data cube. *Distributed and Parallel Databases*, 11(2):181–201, 2002.

[6] P. Flajolet and G.N. Martin. Probablistic counting algorithms for database applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.

[7] G.N. Frederickson. Optimal algorithms for tree partitioning. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 168–177, 1991.

[8] S. Goil and A. Choudhary. A parallel scalable infrastructure for OLAP and data mining. In *Proc. International Data Engineering and Applications Symposium (IDEAS'99)*, Montreal, August 1999.

[9] 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 subtotals. *J. Data Mining and Knowledge Discovery*, 1(1):29–53, April 1997.

[10] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Implementing data cubes efficiently. In *Proc. ACM SIGMOD*, pages 205–216, 1996.

[11] Max Planck Institute. *LEDA*. http: // www.algorithmic - solutions.com / enleda.htm, 2003.

[12] H. Lu, X. Huang, and Z. Li. Computing data cubes using massively parallel processors. In *Proc. 7th Parallel Computing Workshop (PCW'97) Canberra, Australia*, 1997.

[13] Sun Microsystems. *SunFire 6800*. http: // www.sun.com / servers / midrange / sunfire6800, 2003.

[14] R.T. Ng, A. Wagner, and Y. Yin. Iceberg-cube computation with pc clusters. In *Proc. of 2001 ACM SIGMOD Conference on Management of Data*, pages 25–36, May 2001.

[15] N. Pendse and R. Creeth. OLAP Architectures. http://www.olapreport.com/Analyses.htm, 2003.

[16] Y. Perl and U. Vishkin. Efficient implementation of a shifting algorithm. *Disc. Appl. Math.*, (12):71–80, 1985.

[17] K.A. Ross and D. Srivastava. Fast computation of sparse datacubes. In *Proc. 23rd VLDB Conference*, pages 116–125, 1997.

[18] A. Shukla, P. Deshpande, J.F. Naughton, and K. Ramasamy. Storage estimation for mutlidimensional aggregates in the presence of hierarchies. In *Proc. 22nd VLDB Conference*, pages 522–531, 1996.

[19] J.X. Yu and H. Lu. Multi-cube computation. In *Proc. 7th International Symposium on Database Systems for Advanced Applications*, Hong Kong, April 18-21, 2001.

[20] Y. Zhao, P.M. Deshpande, and J.F.Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *Proc. ACM SIGMOD Conf.*, pages 159–170, 1997.

[21] W. Zipf. *The Psycho-Biology of Language: An Introduction to Dynamic Philology.* Houghton Mifflin, 1935.