Computing Partial Data Cubes [*]

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

Todd Eavis
Faculty of Computer Science
Dalhousie University
Halifax, Canada B3H 1W5
eavis@cs.dal.ca
http://www.cs.dal.ca/∼eavis

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

Abstract

The precomputation of the different views of a data cube is critical to improving the response time of data cube queries for On-Line Analytical Processing (OLAP). However, the user is often not interested in the set of all views of the data cube but only in a certain subset of views. In this paper, we study the problem of computing the partial data cube, i.e. a subset of selected views in the lattice. We consider the case of dense relations, using top-down cube construction methods like Pipesort. This paper presents, both, sequential and parallel methods for partial data cube construction as well as an experimental performance evaluation of our methods.

1 Introduction

The precomputation of the different views (group-bys) of a data cube [9] is critical to improving the response time of data cube queries for On-Line Analytical Processing (OLAP). Numerous solutions for generating the entire data cube (i.e. all views) have been proposed; see e.g. [3, 12, 18, 19, 22]. One of the main differences between the many solutions is whether they are aimed at sparse or dense relations. To meet the need for improved performance and to effectively handle the increase in data sizes, parallel solutions for generating the data cube have been proposed in [4, 7, 8, 16, 14, 21]. However, the user is often not interested in the set of *all* views of the data cube but only in a certain *subset* of views. For example, only the views up to a certain number of dimensions may be of interest as these views are more easily visualized. More importantly, for a large number of dimensions and for realistic size data sets, where the original relation may be terabytes in size, it is often impractical to compute the entire data cube. For such applications it is critical to be able to compute a select subset of views rather than the entire data cube. The problem of *selecting* a subset of views that minimizes the query response time is studied in [10, 11, 12]. However, once such a subset of views is selected, it is critical to have efficient methods available that materialize the given set of views. This is the problem addressed in this paper.

Given a relation R of size n and dimension d as well as a subset S of the set of all possible views in the lattice L, we refer to the problem of computing all views in S as the *partial data cube* problem. In this paper, we study the problem of computing the partial data cube for the case of dense relations, using top-down cube construction methods like Pipesort [19].

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

We present, both, sequential and parallel methods for partial data cube construction as well as an experimental performance evaluation of our methods.

The central problem for top-down partial cube construction is to build a schedule tree T of minimum cost connecting all views of S and some intermediate nodes (views) chosen in order to reduce the total cost. This problem has also been discussed in [19]. A particular challenge for the Pipesort approach is that it builds a schedule tree by proceeding level by level through the lattice and building minimum cost bipartite matchings between levels. However, the schedule tree for a partial cube may require edges between nodes at arbitrary levels of the lattice. To solve this problem, the authors in [19] 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. See Figure 1 for an illustration. The authors in [19] 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 $\sum_{k=0}^{d} \left(\begin{array}{c} d \\ k \end{array} \right)$ and $\sum_{k=1}^{d} \left(\begin{array}{c} d \\ k \end{array} \right) k$, respectively, while the number of vertices and edges in the augmented lattice with Steiner vertices and edges are $\sum_{k=0}^{d} \left(\begin{array}{c} d \\ k \end{array} \right) k! + |S|$ and $\sum_{k=1}^{d} \left[\left(\begin{array}{c} d \\ k \end{array} \right) k! \sum_{j=1}^{k} \frac{k!}{(j-1)!} \right] + |S|$, respectively. Table 1 provides some examples for $d = 3, \ldots, 10$. As indicated, the number of Steiner edges exceeds 2,000,000,000 for $d \geq 9$. This makes such an approach impractical for relations with more than just a very small number of dimensions. The examples show 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.

In this paper, we present two methods, Tree__Partial__Cube(S, PC) and Lattice__Partial__Cube(S, PC), which create a schedule tree T without the use of Steiner vertices or edges. For our method Tree__Partial__Cube(S, PC), the nodes of the schedule tree T are a subset of the nodes of the Pipesort tree of the complete cube, whereas for our method Lattice__Partial__Cube(S, PC) the schedule tree T is a subgraph of the lattice L. The heart of our algorithm is a method Partial__Cube__Schedule(S, G, T) which builds, in two steps, the schedule tree T from a guiding graph, G, which is a subgraph of either the Pipesort tree or lattice L. First, Partial__Cube__Schedule(S, G, T) organizes the nodes of S into a tree of minimum total cost, using a greedy approach. Then, it adds intermediate nodes (from $G - T$) to the tree to further minimize the total cost, using a greedy approach as well. Our algorithm also introduces the use of "plan" variables which represent the best way for a given node v to be inserted into T. In addition, we present two parallel methods Parallel__Tree__Partial__Cube(p, S, PC) and Parallel__Lattice__Partial__Cube(p, S, PC) which parallelize our partial cube generation methods for a p processor shared disk multiprocessor, like the SunFire 6800 [15]. 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 workloads over the p processors.

We have implemented Tree__Partial__Cube, Lattice__Partial__Cube, Parallel__Tree__Partial__Cube, and Parallel__Lattice__Partial__Cube and tested them on a SunFire 6800 [15]. For comparison purpose, a practical alternative method for computing a small set of views might be to compute the directly via separate sorts, whereas computing larger sets of views might be done more efficiently by computing the entire cube via Pipesort. Therefore, we compared the performance of Tree__Partial__Cube and Lattice__Partial__Cube with the performance of algorithm Simple__Partial__Cube consisting of computing either the entire data cube via Pipesort or, for small subsets S, computing the views in S individually through separate sorts, whichever is faster. We observed that, when up to 50% of all possible views are selected, Tree__Partial__Cube and Lattice__Partial__Cube show a 30% to 45% performance improvement in comparison with Simple__Partial__Cube. For up to 50% of all views selected, the methods Tree__Partial__Cube and Lattice__Partial__Cube exhibit very similar performance. Beyond that point, the Lattice__Partial__Cube method appears to perform better. We also observe that Lattice__Partial__Cube has approximately the same performance as Pipesort when all views are selected. 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.

Our observations show 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. Note that, Lattice__Partial__Cube is also considerably easier to implement than Pipesort because it does not require minimum cost bipartite matching.

The remainder of this paper is organized as follows. In the following Section 2 we present our sequential methods Tree__Partial__Cube and Lattice__Partial__Cube. Section 3 outlines the parallel methods Parallel__Tree__Partial__Cube(p, S, PC) and Parallel__Lattice__Partial__Cube(p, S, PC). Section 4 presents the performance evaluation of our methods.

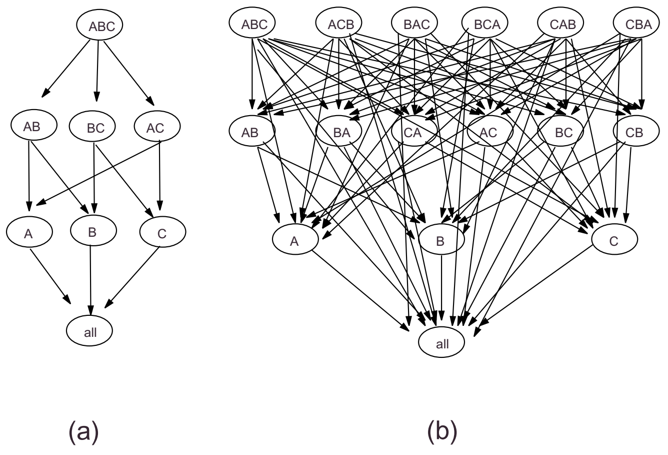

Figure 1: (a) Three-Dimensional Lattice (b) Three-Dimensional Augmented Lattice

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 via pipesort 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

Dimensions	No. of Nodes, Orig. Lattice	No. of Edges, Orig. Lattice	No. of Nodes, Augm. Lattice	No. of Edges, Augm. Lattice
3	8	12	16	117
4	16	32	65	1,948
5	32	80	326	47,665
6	64	192	1,957	1,667,286
7	128	448	13,700	79,777,285
8	256	1,024	109,601	718,178,136
9	512	2,304	986,410	N/A
10	1,024	5,120	9,864,101	N/A

Table 1: Number Of Nodes And Edges In The Original Lattice L, and Number Of Nodes And Edges In The Augmented Lattice With Steiner Vertices And Edges. N/A Denotes An Integer "Roll-Over" At 2,000,000,000.

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 apply Pipesort to T in order to create the partial cube PC. As a preprocessing step, we compute the lattice, L, of all 2^d possible views [9] and use a storage estimator [5, 20] to estimate the approximate sizes of all 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 partial data cube PC using Pipesort applied to 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 partial data cube PC using Pipesort applied to tree T.
```

The difference between the two methods is that, in Tree_Partial_Cube(S, PC) the schedule tree T is a subset of the Pipesort tree of the complete cube 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 ordering have not been fixed, a node w can be created from v iff the attributes of w are 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, using a greedy approach. In Step 2, it adds intermediate nodes (from $G-T$) to the tree to further minimize the total cost, again using a greedy approach. 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:

node: the node v considered to be inserted,
parent: the chosen parent of v,
parent__mode: the chosen mode (scan or sort) for computing v from its parent,
scan__child: the chosen child of v that is computed via scan,
insertion__scan__child: the chosen scan child of v in the case of scan insertion,
sort__children: the chosen children of v that are computed via sort,
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 = $\emptyset$
        WHILE S' not empty
            clear(BP)
            FOR every v $\in$ 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. An illustration of the three cases in Procedure 4 is given in Figure 2.

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)
```

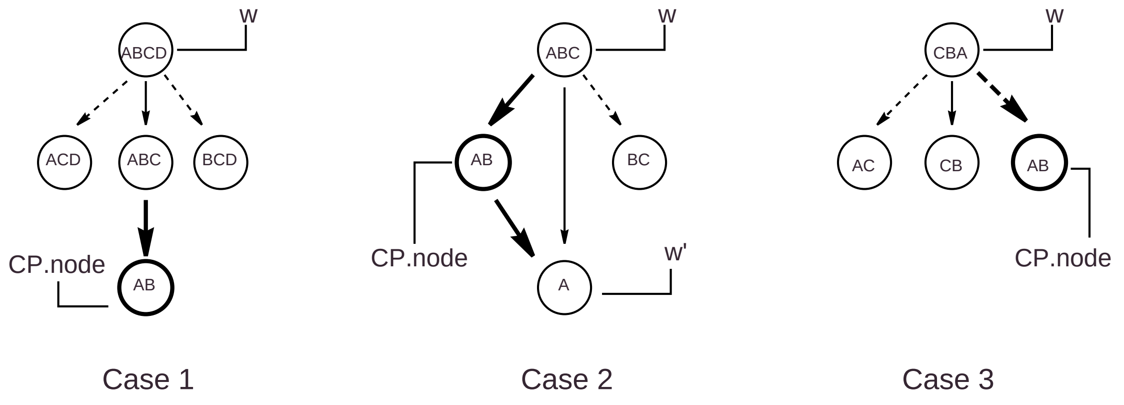

Figure 2: Illustration Of The Three Cases In Procedure 4.

```
        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; see Figure 3(a).

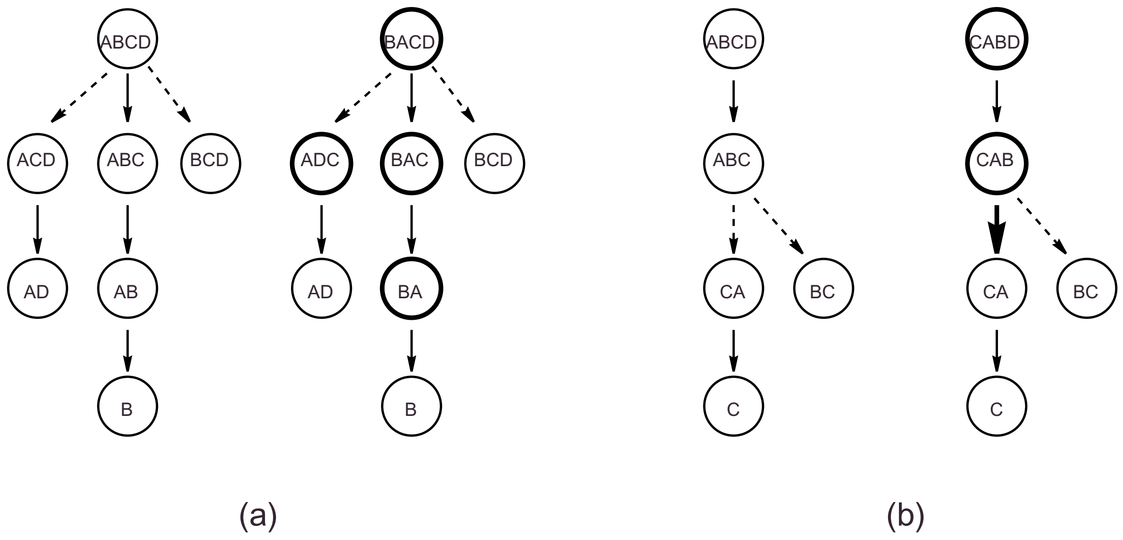

Figure 3: Illustration of Establish__Attribute__Orderings(T) and Fix__Pipelines(T).

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; see Figure 3(b).

Following the completion of the schedule tree T both methods, Tree__Partial__Cube(S, PC) and Lattice__Partial__Cube(S, PC), conclude by executing a modified version of pipesort where the standard pipesort tree is replaced 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 [15]. 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 $P_i$ using Pipesort.
```

```
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 $\Sigma_i$ using Pipesort.
```

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 [4] 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* [2, 6, 17]. Our tree partitioning heuristic developed in [4] adapts the algorithm in [2] 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, $\Sigma_1, \ldots, \Sigma_p$) which has as input the schedule tree, T, number of processors, p, and overpartitioning ratio, s, and creates as output p sets of trees, $\Sigma_1, \ldots, \Sigma_p$, where each set Σ_i contains the s subtrees of T which will be assigned to processor P_i. As shown in [4], 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_Cu be, and Parallel_Lattice_Partial_Cube using C and the MPI communication library [1]. Most of the required graph algorithms, as well as data structures like hash tables and graph representations, were drawn from the LEDA library [13]. 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, the individuals sorts will often be faster, while building the full data cube is often faster when the percentage of selected views is high. 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 can create 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). 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.

Our experiments proceeded in the following steps:

1. **Sequential Experiments:** We executed Simple_Partial_Cube(S, PC), Tree_Partial_Cube and Lattice_Partial_Cube on a single processor of our parallel machine and measured the sequential wall clock time.

2. **Parallel Experiments:** We executed Parallel_Tree_Partial_Cube and Parallel_ Lattice_Partial_Cube on up to 16 processors of our parallel machine and measured the parallel wall clock time.

4.2 Performance Results: Sequential Experiments

Figure 4 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.

Figure 5 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. Each data set has 8 dimensions and mixed cardinalities, varying between 2 and 1000 for the different dimensions. 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 6 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. The data sets consist of 200,000 rows with mixed cardinalities, varying between 2 and 1000 for the different dimensions. 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.

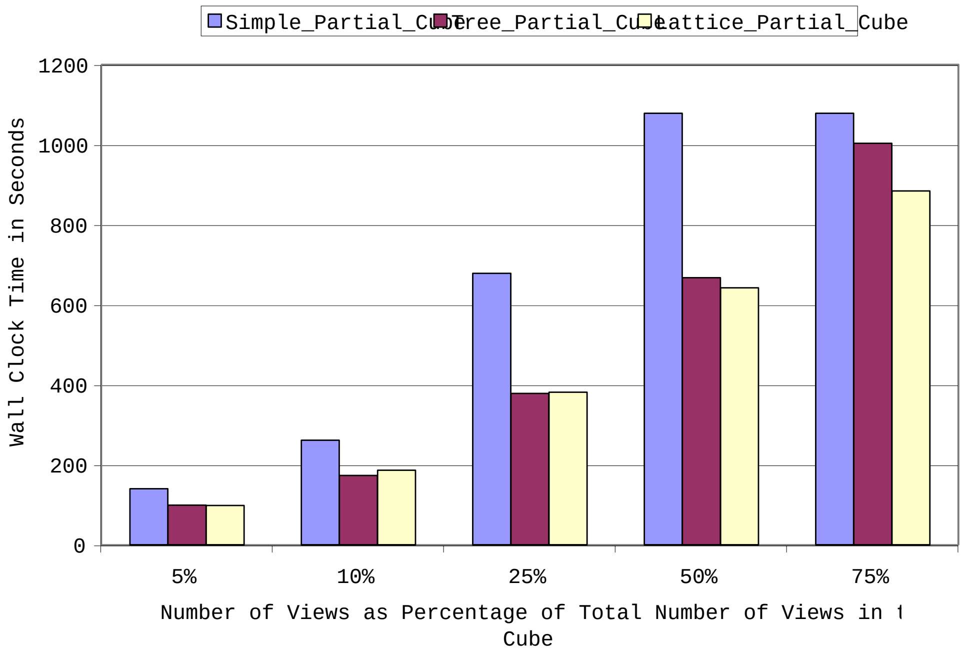

Figure 4: Sequential Wall Clock Time In Seconds As A Function Of The Percentage Of Selected Views. Comparison Between Simple__Partial__Cube, Tree__Partial__Cube And Lattice__Partial__Cube. (Fixed Parameters: No. Of Processors = 1. Data Size = 200,000 Rows. Dimensions = 8. Skew (ZIPF): $\alpha = 0$.)

Figure 7 presents the same data as Figure 6 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. The data sets consist of 200,000 rows with mixed cardinalities, varying between 2 and 1000 for the different dimensions. 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.

Figure 8 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 9, using the same data, presents the relative improvement as a function of the percentage of selected views. The data sets consist of 200,000 rows with mixed cardinalities, varying between 2 and 1000 for the different dimensions. It is interesting to observe how similar these curves are to the curves shown in Figure 6 and 7. 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

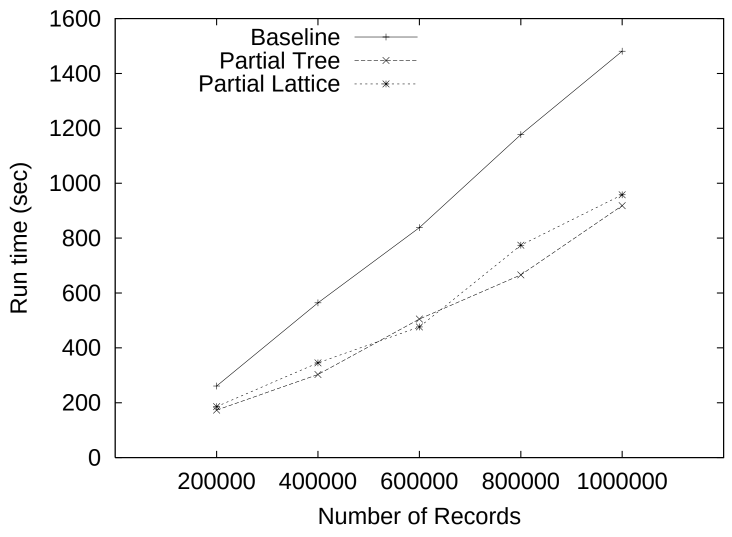

Figure 5: Sequential Wall Clock Time In Seconds As A Function Of The Data Size. Comparison Between Simple_Partial_Cube, Tree_Partial_Cube And Lattice_Partial_Cube. (Fixed Parameters: No. Of Processors = 1. Dimensions = 8. Percentage Of Views Selected = 10%. Skew (ZIPF): $\alpha = 0$.)

the relative improvement is increasing as the dimensionality of the problem increases.

Figure 10 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. Here we used the standard ZIPF distribution in each dimension with $\alpha = 0$ (no skew) to $\alpha = 2$. The data sets consist of 200,000 rows with mixed cardinalities, varying between 2 and 1000 for the different dimensions. 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 10. 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 10 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.

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 11 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. The data sets consist of 200,000 rows with mixed cardinalities, varying between 2 and 1000 for the different dimensions. 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 11 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

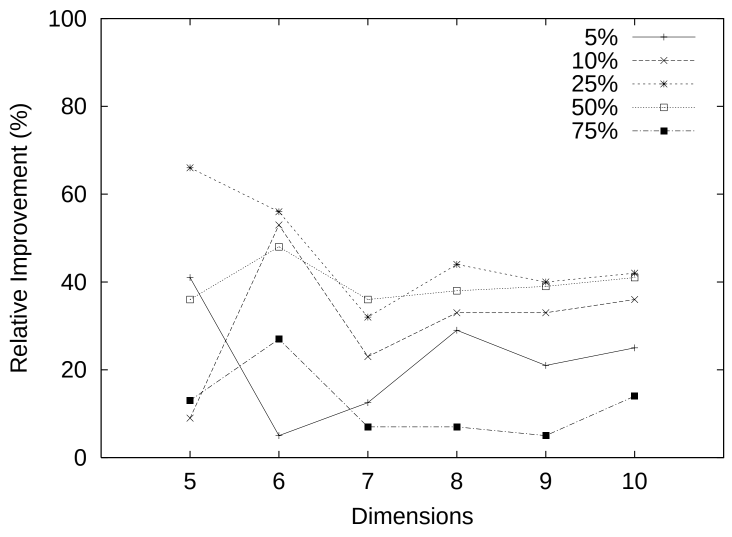

Figure 6: Relative Improvement In Wall Clock Time For Sequential Tree__Partial__Cube W.R.T. Simple__Partial__Cube As A Function Of The Number Of Dimensions, For Different Percentages Of Selected Views. (Fixed Parameters: No. Of Processors = 1. Data Size = 200,000 Rows. Skew (ZIPF): $\alpha = 0$.)

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.

Figures 12 and 13 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.

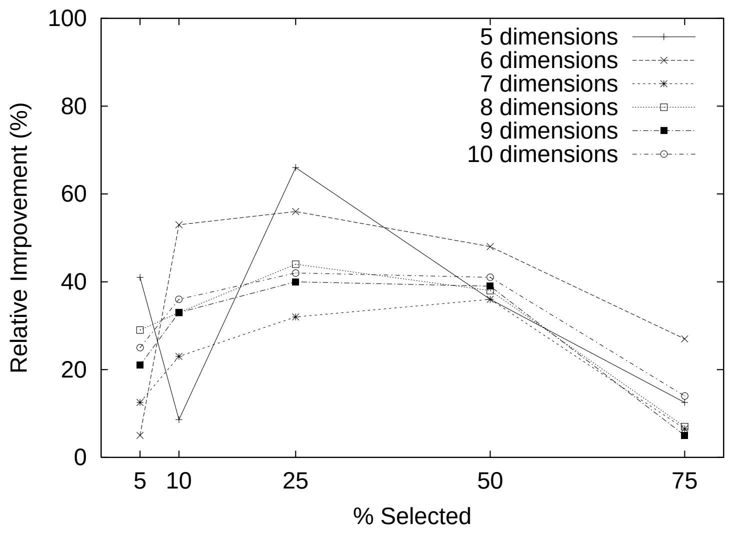

Figure 7: Relative Improvement In Wall Clock Time For Sequential Tree_Partial_Cube W.R.T. Simple_Partial_Cube As A Function Of The Percentage Of Selected Views, For Different Numbers Of Dimensions. (Fixed Parameters: No. Of Processors = 1. Data Size = 200,000 Rows. Skew (ZIPF): $\alpha = 0$.

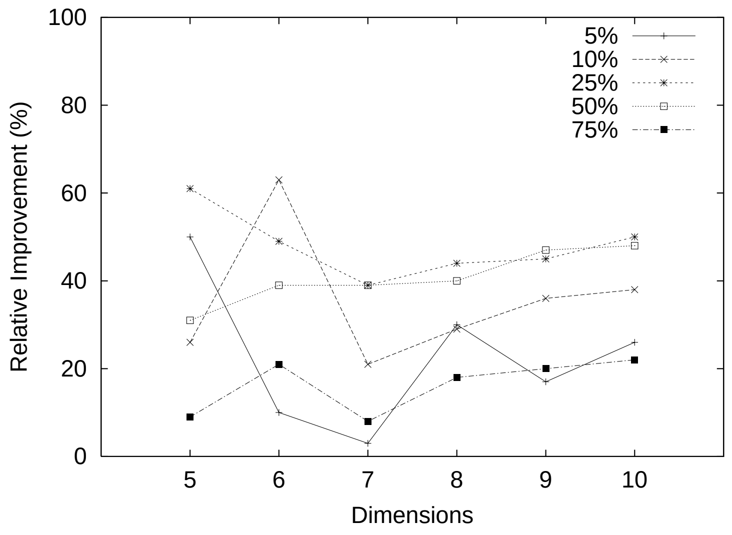

Figure 8: Relative Improvement In Wall Clock Time For Sequential Lattice_Partial_Cube W.R.T. Simple_Partial_Cube As A Function Of The Number Of Dimensions, For Different Percentages Of Selected Views. (Fixed Parameters: No. Of Processors = 1. Data Size = 200,000 Rows. Skew (ZIPF): $\alpha = 0$.)

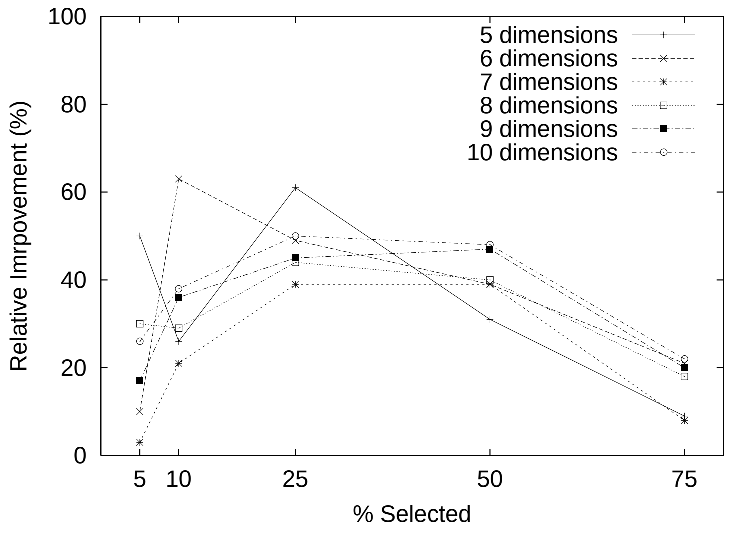

Figure 9: Relative Improvement In Wall Clock Time For Sequential Lattice_Partial_Cube W.R.T. Simple_Partial_Cube As A Function Of The Percentage Of Selected Views, For Different Numbers Of Dimensions. (Fixed Parameters: No. Of Processors = 1. Data Size = 200,000 Rows. Skew (ZIPF): $\alpha = 0$.)

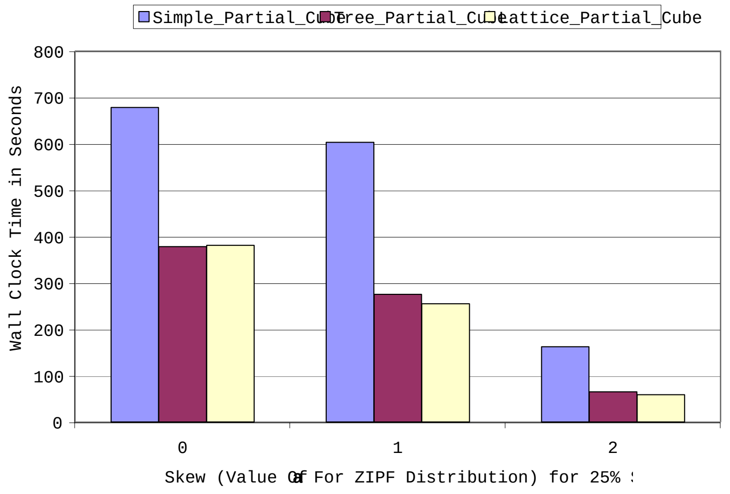

Figure 10: Sequential Wall Clock Time In Seconds As A Function Of The Skew (ZIPF) When 25% Of The Views Are Selected. Comparison Between Simple_Partial_Cube, Tree_Partial_Cube And Lattice_Partial_Cube. (Fixed Parameters: No. Of Processors = 1. Data Size = 200,000 Rows. Dimensions = 8.)

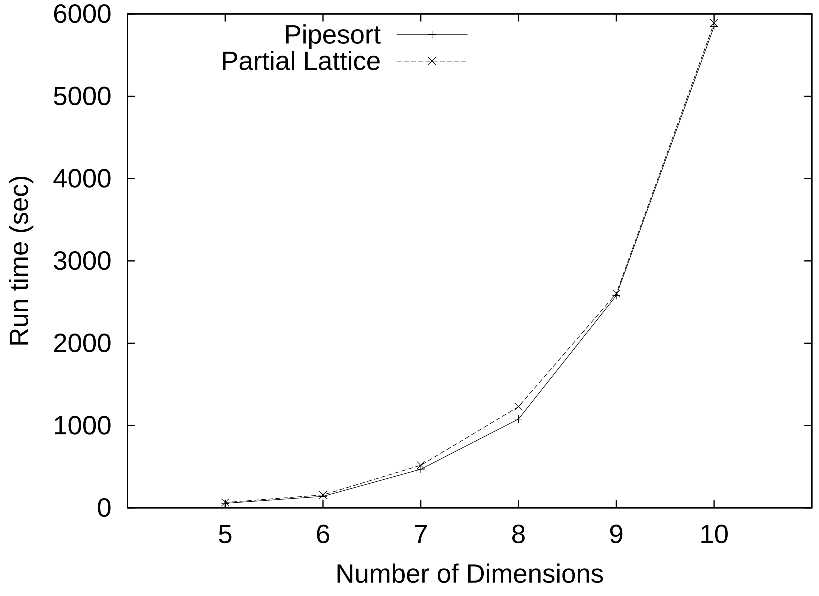

Figure 11: Comparison Between Pipesort And Lattice_Partial_Cube For Computing The Entire Data Cube (Percentage Of Selected Views = 100%). Sequential Wall Clock Time In Seconds As A Function Of The Number Of Dimensions. (Fixed Parameters: No. Of Processors = 1. Data Size = 200,000 Rows. Skew (ZIPF): $\alpha = 0$.)

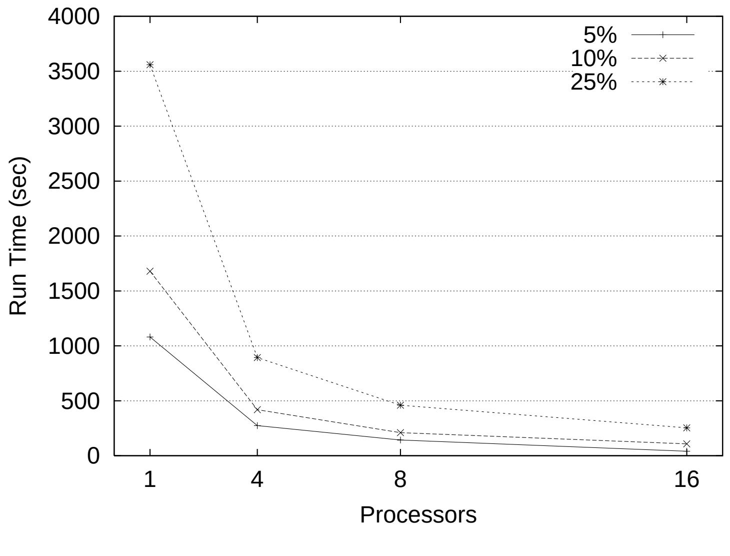

Figure 12: Parallel Wall Clock Time In Seconds For Parallel_Tree_Partial_Cube(S, PC) As A Function Of The Number Of Processors, For Different Percentages Of Selected Views. (Fixed Parameters: Data Size = 1,000,000 Rows. Dimensions = 8. Skew (ZIPF): $\alpha = 0$.)

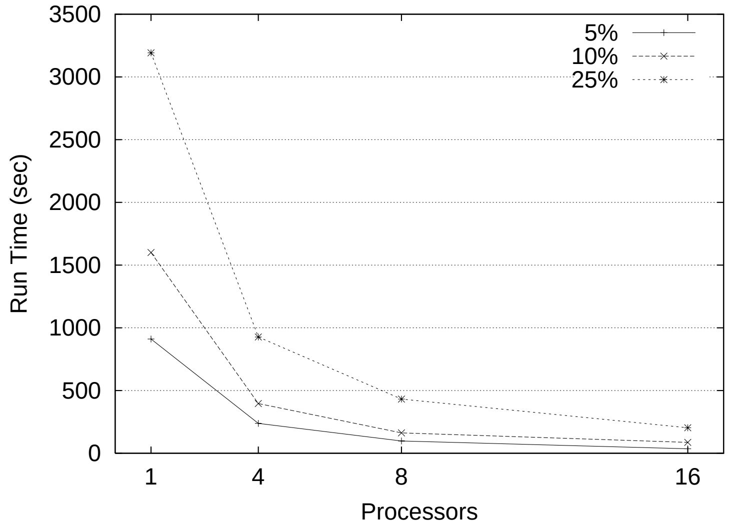

Figure 13: Parallel Wall Clock Time In Seconds For Parallel_Lattice_Partial_Cube(S, PC) As A Function Of The Number Of Processors, For Different Percentages Of Selected Views. (Fixed Parameters: Data Size = 1,000,000 Rows. Dimensions = 8. Skew (ZIPF): $\alpha = 0.$)

References

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

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

[3] 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.

[4] F. Dehne, T. Eavis, S. Hambrusch, and A. Rau-Chaplin. Parallelizing the data cube. *Distributed and Parallel Databases*, 11(2):181–201, 2002. Preliminary version appeared in Proc. 8th International Conference on Database Theory (ICDT 2001), Springer Lecture Notes in Computer Science, Vol. 1973, 2001, pp 129-143.

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

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

[7] S. Goil and A. Choudhary. High performance OLAP and data mining on parallel computers. *Journal of Data Mining and Knowledge Discovery*, 1(4), 1997.

[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 sub-totals. *J. Data Mining and Knowledge Discovery*, 1(1):29–53, April 1997.

[10] H. Gupta. Selection of views to materialize in a data warehouse. In *ICDT*, pages 98–112, 1997.

[11] H. Gupta, V. Harinarayan, A. Rajaraman, and J. D. Ullman. Index selection for OLAP. In *ICDE*, pages 208–219, 1997.

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

[13] Max Planck Institute. *LEDA*. http://www.mpi-sb.mpg.de/LEDA/.

[14] 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.

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

[16] 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.

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

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

[19] S. Sarawagi, R. Agrawal, and A. Gupta. On computing the data cube. Technical Report RJ10026, IBM Almaden Research Center, San Jose, CA, 1996.

[20] A. Shukla, P. Deshpende, 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.

[21] 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.

[22] 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.