Distributed and Parallel Databases, 15, 219–236, 2004

Parallel ROLAP Data Cube Construction on Shared-Nothing Multiprocessors*

YING CHEN ychen@cs.dal.ca
Faculty of Computer Science, Dalhousie University, Halifax, Canada

FRANK DEHNE frank@dehne.net
School of Computer Science, Carleton University, Ottawa, Canada

TODD EAVIS eavis@cs.dal.ca
ANDREW RAU-CHAPLIN arc@cs.dal.ca
Faculty of Computer Science, Dalhousie University, Halifax, Canada

Recommended by: Ahmed Elmagarmid

Abstract. The pre-computation of data cubes is critical to improving the response time of On-Line Analytical Processing (OLAP) systems and can be instrumental in accelerating data mining tasks in large data warehouses. In order to meet the need for improved performance created by growing data sizes, parallel solutions for generating the data cube are becoming increasingly important. This paper presents a parallel method for generating data cubes on a shared-nothing multiprocessor. Since no (expensive) shared disk is required, our method can be used on low cost Beowulf style clusters consisting of standard PCs with local disks connected via a data switch. Our approach uses a ROLAP representation of the data cube where views are stored as relational tables. This allows for tight integration with current relational database technology.

We have implemented our parallel shared-nothing data cube generation method and evaluated it on a PC cluster, exploring relative speedup, local vs. global schedule trees, data skew, cardinality of dimensions, data dimensionality, and balance tradeoffs. For an input data set of 2,000,000 rows (72 Megabytes), our parallel data cube generation method achieves close to optimal speedup; generating a full data cube of ≈227 million rows (5.6 Gigabytes) on a 16 processors cluster in under 6 minutes. For an input data set of 10,000,000 rows (360 Megabytes), our parallel method, running on a 16 processor PC cluster, created a data cube consisting of ≈846 million rows (21.7 Gigabytes) in under 47 minutes.

Keywords: data warehousing, OLAP, data cube, high performance computing

1. Introduction

The pre-computation of the different views (group-bys) of a data cube, i.e. the forming of aggregates for every combination of GROUP-BY attributes, is critical to improving the response time of On-Line Analytical Processing (OLAP) queries in decision support systems [10] and can be instrumental in accelerating data mining tasks in large data warehouses [11]. For a given raw data set, R, with n records and d attributes (dimensions), a view is

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

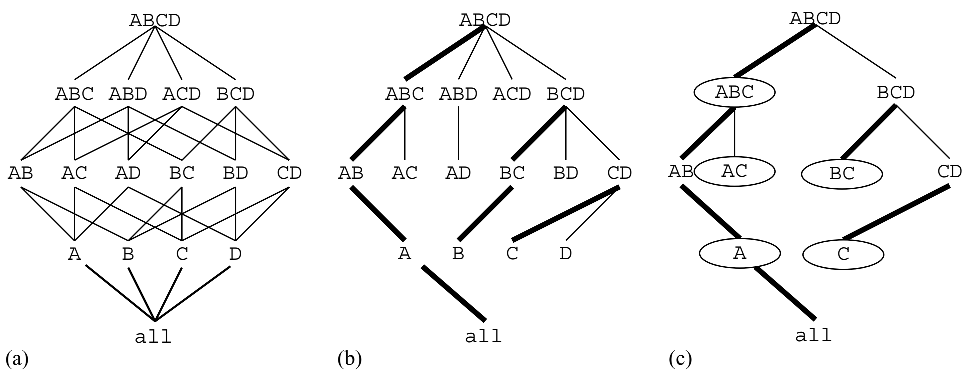

Figure 1. (a) Lattice for 4 dimensions “A”, “B”, “C”, and “D”. (b) Schedule tree for a full data cube (e.g. Pipesort). Bold edges represent “scan” operations and regular edges represent “sort” operations. (c) Schedule tree for a partial data cube. Selected views are marked by circles.

constructed by an aggregation of R along a subset of attributes. This results in 2^d different possible views. Figure 1(a) shows the different possible view identifiers for 4 dimensions “A”, “B”, “C”, and “D”. An edge between two view identifiers indicates that one of the respective views can be computed from the other by aggregation along one dimension. The resulting graph is called the *lattice*. As proposed in [10], the pre-computation of the entire data cube (the set of all 2^d views) allows for the fast execution of subsequent OLAP queries.

There are two basic data cube representations: ROLAP representations where views are represented as relational tables and MOLAP representations where views are represented as multi-dimensional arrays. Many methods have been proposed for generating the data cube on sequential [2, 12, 19, 20, 24, 25] and parallel systems [3, 4, 7, 9, 14, 16, 18]. The size of the data cube is potentially very large. In order to meet the need for improved performance created by growing data sizes in OLAP applications, *parallel* solutions for generating the data cube have become increasingly important. The current parallel approaches can be grouped into two broad categories: (1) *work partitioning* [3, 4, 14, 16, 18] and (2) *data partitioning* [7, 9].

Work partitioning methods assign different view computations to different processors. Consider, for example, the lattice for a four dimensional data cube as shown in figure 1(a). From the raw data set “ABCD”, 15 views need to be computed. Given a parallel computer with p processors, work partitioning schemes partition the set of views into p groups and assign the computation of the views in each group to a different processor. The main challenges for these methods are load balancing and scalability, which are addressed in different ways by the different techniques studied in [3, 4, 14, 16, 18]. One distinguishing feature of work partitioning methods is that all processors need simultaneous access to the entire raw data set. This access is usually provided through the use of a shared disk system (available e.g. for SunFire 6800 and IBM SP systems).

Data partitioning methods partition the raw data set into p subsets and store each subset locally on one processor. All views are computed on every processor but only with respect to the subset of data available at each processor. A subsequent “merge” procedure is required to agglomerate data across processors. The advantage of data partitioning methods is that they do not require all processors to have access to the entire raw data set. Each processor only requires a local copy of a portion of the raw data which can, e.g., be stored on its local disk. This makes such methods feasible for shared-nothing parallel machines like the popular, low cost, Beowulf style clusters consisting of standard PCs connected via a data switch and without any (expensive) shared disk array (see figure 2(a)). The main challenge for data partitioning methods is that the “merge”, which has to be performed for every view of the data cube, has the potential to create massive data movements between processors with serious consequences for performance and scalability of the entire system. A data partitioning method for MOLAP representations has been presented in [7, 9]. This method is based on a space partitioning of the multi-diminsional array and a spatial “merge” between different sub-cubes of the MOLAP cube. The spatial “merge” operation can be reduced to a parallel prefix which is a well studied operation for parallel computers.

In this paper, we study data partitioning methods for the ROLAP case where the raw data set is given as a d-dimensional relation (table of d-tuples) and all views are to be created as relational tables as well. The principal advantage of ROLAP is that it allows for tight integration with current relational database technology. Another advantage of ROLAP is that it requires only linear space and is therefore suitable for the construction of very large data cubes. Our algorithm is, to our knowledge, the first parallel ROLAP data cube construction method for shared-nothing multiprocessors. Our method has the additional advantage that it can be extended to the *partial* cube case where not all views but only a subset of views, selected by the user, are to be created. This case occurs frequently in practice because the user often knows that some views will not be required for on-line analytical processing (OLAP) queries on a given data set.

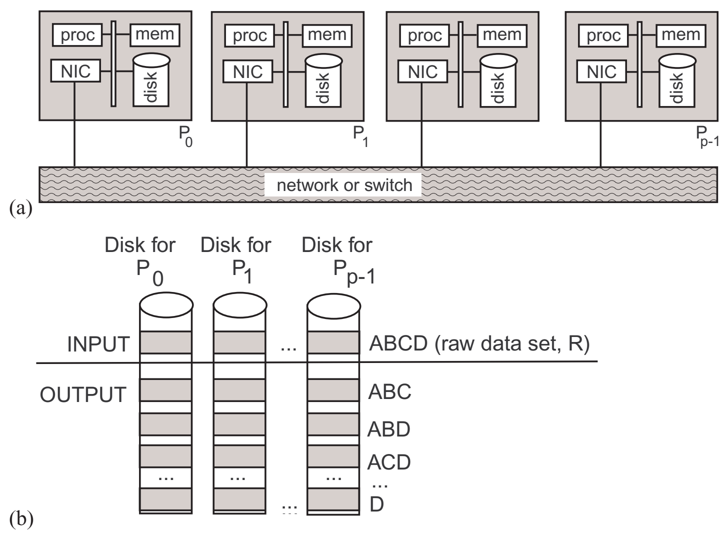

Figure 2. (a) A shared-nothing multiprocessor. (b) Input and output data distribution.

We have implemented our parallel data cube generation method and extensively evaluated it on a shared-nothing PC cluster. Our experiments have explored the following six performance issues: relative speedup, local vs. global schedule trees, data skew, cardinality of dimensions, data dimensionality, and balance tradeoffs. For a raw data set R of size $n = 2,000,000$ rows (72 Megabytes) our parallel ROLAP data cube generation method achieved close to optimal speedup; generating a full data cube of $\approx$227 million rows (5.6 Gigabytes) in under 6 minutes on 16 processors. For an input data set of $n = 10,000,000$ rows (360 Megabytes), our parallel method created the corresponding data cube consisting of $\approx$846 million rows (21.7 Gigabytes) in under 47 minutes.

The remainder of this paper is organized as follows. In Section 2 we present our parallel ROLAP data cube algorithm for shared-nothing multiprocessors. Section 3 shows how our method can be extended to the partial cube case where not all views but only a subset of selected views are to be created. Section 4 discusses our implementation and presents the performance results achieved by our method. Section 5 concludes the paper.

2. A parallel data cube algorithm for shared-nothing multiprocessors

We consider a shared-nothing parallel machine consisting of p processors $P_0, P_1 \dots P_{p-1}$, each with its own local memory and local disk, connected via a network or switch (see figure 2(a)). There is no shared memory or shared disk available. Systems of this type include the popular, low cost, Beowulf style clusters consisting of standard PCs connected via a switch (http://www.beowulf.org/). As input, we assume a raw data set, R, with n records and d dimensions $D_0, D_1 \dots D_{d-1}$ distributed evenly over the p disks (see figure 2(b)). The basic communication operation used by our data cube algorithm is the h-relation operation (method `MPI_ALL_TO_ALL_v` in MPI). Our method uses two basic local disk operations, applied by each processor to its local disk: (1) linear scan and (2) external memory sort [24]. For a processor P_j with local memory size m and a local disk with block transfer size B, a linear scan through a file of size n stored on its disk requires $O(\frac{n}{B})$ block transfers between disk and memory while an external memory sort of that file requires $O(\frac{n}{B} \log_{\frac{m}{B}} \frac{n}{B})$ block transfers [22]. We will present our method for a shared-nothing multiprocessor with one local disk per processor P_j. However, it is easy to generalize our methods to machines with multiple local disks per processor by applying the linear scan and external memory sort methods for a single processor with multiple local disks presented in [23].

Without loss of generality, let $|D_0| \geq |D_1| \geq \dots \geq |D_{d-1}|$, where $|D_i|$ is the cardinality for dimension D_i, $0 \leq i \leq d-1$ (i.e. the number of distinct values for dimension D_i). Let S be the set of all 2^d view identifiers. Each view identifier consists of a subset of $\{D_0, D_1 \dots D_{d-1}\}$, ordered by the cardinalities of the selected dimensions (in decreasing order). The goal is to create a data cube DC containing the views in S. We assume that, when our algorithm terminates, every view is distributed evenly across the p disks (see figure 2(b)). It is important to note that, for the subsequent use of the views by OLAP queries, each view needs to be evenly distributed in order to achieve maximum I/O bandwidth for subsequent parallel disk accesses.

2.1. Algorithm outline

Our parallel algorithm uses as a building block a standard sequential top-down data cube method such as Pipesort [20]. Such methods have in common that they consist of a two phase approach. In the first phase, a *schedule tree T* is constructed which is a subgraph of the lattice and contains as nodes the identifiers of all views to be constructed. Recall that view v is a parent of a view v' if v' can be created from v. The schedule tree T identifies a sequence in which the views are to be constructed in the second phase. The main difference between the various top-down data cube methods is the schedule tree T that they build. For example, Pipesort starts with the lattice and assigns to every view identifier an estimate of the size of the respective view [6, 21]. It then computes the cost of the aggregate operation associated with each edge of the lattice. The schedule tree T is then built by scanning the lattice level by level, starting at the raw data set, and computing for each two subsequent levels of nodes, and the edges between them, a minimum cost bi-partite matching.

Let $S_i \subset S$ be the subset of view identifiers in S that start with D_i, and let DC_i be the data cube for S_i. We call DC_i the D_i-partition of the data cube DC. Furthermore, we refer to the view consisting of all dimensions contained in views of S_i as the D_i*-root* (see figure 3).

The following describes the global structure of our parallel data cube algorithm for shared-nothing multiprocessors. The algorithm consists of three main phases: data partitioning, computation of local D_i-partitions, and merge of local D_i-partitions. Subsequent sections will discuss each phase in more detail.

Procedure 1. Parallel–Shared–Nothing–Data–Cube

/* Input: Raw data set R (n d-dimensional records) distributed arbitrarily over the p processors, n/p records per processor; Output: Data cube, DC, distributed over the p processors. Each views is evenly distributed over the p processors' disks. */

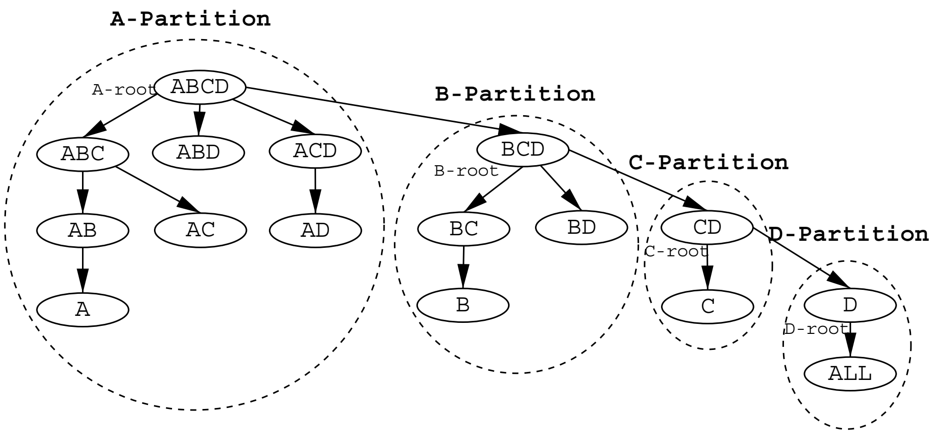

Figure 3. Partitions of a data cube for $d = 4$. Dimensions are labelled "A", "B", "C", "D".

```
FOR i=0 TO d-1

(1) /* Data Partitioning */
    (a) Each processor $P_j$ ($j = 0, \ldots p-1$) computes locally the $D_i$-root for its subset
        of data. (Essentially a sequential sort followed by a sequential scan.) Let $D_i$-root$|_j$
        denote the $D_i$-root created by processor $P_j$.
    (b) /* Sort $\cup_{j=0,\ldots p-1} D_i$-root$|_j$ by $D_i, \ldots D_{d-1}$ */
        Adaptive–Sample–Sort($D_i$-root$|_0, \ldots, D_i$-root$|_{p-1}$; $D_i, \ldots, D_{d-1}$; $\beta = 1\%$)
    (c) Each processor $P_j$ ($j = 0, \ldots p-1$) computes locally the $D_i$-root for its subset
        of data received in the previous step. Let $D_i$-root$||_j$ denote the $D_i$-root created by
        processor $P_j$.

(2) /* Computation of Local $D_i$-Partitions */
    (a) Processor $P_0$ locally computes, by applying the first phase of a sequential top-down
        data cube method, the schedule tree $T_i$ for building the $D_i$-partition with respect to
        $D_i$-root$||_0$.
    (b) Processor $P_0$ broadcasts $T_i$ to $P_1 \ldots P_{p-1}$.
    (c) Each processor $P_j$ ($j = 0, \ldots p-1$) computes locally the $D_i$-partition with respect
        to $D_i$-root$||_j$ by applying the second phase of a sequential top-down data cube
        method to the schedule tree $T_i$ received in the previous step.

(3) /* Merge of Local $D_i$-Partitions */
    Merge–Partitions($D_i$)
```

The following Sections 2.2, 2.3 and 2.4 discuss in detail the three phases of Procedure 1.

2.2. *Data partitioning*

Good data partitioning is a key factor in obtaining a good load balance and, consequently, good performance. Some researchers partition data on one or several dimensions [8, 17]. In order to achieve sufficient parallelism, they assume that the product of the cardinalities of these dimensions is much larger than the number of processors [8]. The advantage of their method is that they do not need to merge views. For examples, if we partition on A, then ABC and AC need not to be merged, or if we partition on A and B, then ABC and ABD need not to be merged. However, in practice, this assumption is often not true. The cardinality of some dimensions may be small, such as gender, months and intervals for a numeric attribute. The number of processors in a cluster may be large, especially for clusters of workstations. Therefore, those methods are often not scalable. Our method avoids these problems by partitioning on all dimensions and then applying a merge procedure. As our experiments show, the cost for the additional merge is more than compensated for by better overall performance and scalability.

To partition data, we use parallel sample sort [13]. As discussed in [13], one global data movement via one single h-relation is often sufficient to obtain sorted and well balanced data. The subsequent "global shift" operation, which needs another h-relation, is not always necessary. In our implementation of parallel sample sort we measure the imbalance of the sizes of the local data sets after the first h-relation and perform a second "global shift" h-relation only if necessary. Let $y_0, \ldots, y_{p-1}$ be the sizes of the sets $Y_0, \ldots, Y_{p-1}$ created on processors $P_0, \ldots, P_{p-1}$, respectively, after the first h-relation. We calculate the relative imbalance $\mathcal{I}(y_0, \ldots, y_{p-1}) = \max\{(y_{\max} - y_{\text{avg}})/y_{\text{avg}}, (y_{\text{avg}} - y_{\min})/y_{\text{avg}}\}$, where $y_{\max}$, $y_{\min}$, and y_{avg} are the maximum, minimum and average of $y_0, \ldots, y_{p-1}$, respectively. If $\mathcal{I}(y_0, \ldots, y_{p-1}) > \beta$ for some threshold value β, we apply a subsequent "global shift" operation. In our implementation we use a threshold value of $\beta = 1\%$. As discussed in [13], the imbalance after the first h-relation is less if there are no duplicate keys. However, in most data, there are many duplicate values. Therefore, in Step 1a of Procedure 1, we first compute locally on each processor P_j $(j = 0, \ldots p-1)$ the D_i-root for its subset of data. This eliminates all duplicate keys $D_i \ldots D_{d-1}$ for the sort in the subsequent Step 1b.

We refer to our sample sort implementation as Adaptive–Sample–Sort. Since there are so many "folk" versions of parallel sample sort in the literature, we briefly review the exact sequence of steps implemented in our system.

Procedure 2. Adaptive–Sample–Sort($X_0, \ldots, X_{p-1}; D_{i_1}, \ldots, D_{i_k}; \beta$)

Input: Sets $X_0, \ldots, X_{p-1}$ stored on processors $P_0, \ldots, P_{p-1}$, respectively.
Output: Sets $X_0, \ldots, X_{p-1}$ globally sorted by dimensions $D_{i_1}, \ldots, D_{i_k}$.

(1) Each processor P_j $(j = 0, \ldots p-1)$ locally sorts X_j by $D_{i_1}, \ldots, D_{i_k}$ and selects a set of p *local pivots* consisting of the elements with rank 0, $(n/p^2), \ldots ((p-1)n/p^2)$. Each processor P_j then sends its local pivots to processor P_0.
(2) Processor P_0 sorts the p^2 local pivots received in the previous step. Processor P_0 then selects a set of $p-1$ *global pivots* consisting of the elements with rank $(p + \lfloor p/2 \rfloor)$, $(2p + \lfloor p/2 \rfloor) \ldots ((p-1)p + \lfloor p/2 \rfloor)$ and broadcasts the *p global pivots* to all other processors.
(3) Using the $p-1$ *global pivots* received in the previous step, each processor P_j $(j = 0, \ldots p-1)$ locally partitions X_j (sorted by $D_{i_1}, \ldots, D_{i_k}$ from Step 1) into $p-1$ subsequences $X_j^0 \ldots X_j^{p-1}$.
(4) Using one global h-relation, every processor P_j, $j = 0 \ldots p-1$, sends each X_j^k, $k = 0 \ldots p-1$, to processor P_k.
(5) Each processor P_j, $j = 0, \ldots p-1$, receiving p sorted sequences X_k^j, $k = 0 \ldots p-1$, in the previous step, locally merges those sequences into a single sorted sequence Y_j and sends the size y_j of Y_j to processor P_0.
(6) IF $\mathcal{I}(y_0, \ldots, y_{p-1}) > \beta$, as determined by processor P_0 THEN all processors $P_0, \ldots, P_{p-1}$ balance the sizes of $Y_0, \ldots, Y_{p-1}$ via a "global shift", implemented by one h-relation operation.

Following the above global sort, each processor P_j $(j = 0, \ldots p-1)$ applies in Step 1c of Procedure 1 a sequential scan to its data set in order to compute the D_i-root (D_i-root$||_j$) for its local data.

2.3. *Computation of local D_i-partitions*

In this section, we discuss Step 2 of Procedure 1. The goal of this step is to compute on each processor P_j the D_i-partition with respect to D_i-root$||_j$. For this, we apply on each processor a sequential top-down data cube construction method. Such methods, like Pipesort, typically consist of two phases. In the first phase, a schedule tree T_i is constructed. The nodes of T_i are the view identifiers of the D_i-partition and an edge (u, v) from parent u to child v indicates that v is created from u. Each edge (u, v) is labelled "scan" or "sort". If v is a prefix of u, then v can be created via a linear scan of u. If v is only a subset of u (but not a prefix), then the computation of v requires a re-sort of u. Sequential top-down cube construction methods like Pipesort attempt to build a schedule tree T_i that minimizes the work required for cube construction.

For shared-nothing parallel data cube construction, a problem that arises is that each processor P_j has a different data set, namely D_i-root$||_j$, and that the schedule trees can be different for these different sets. Indeed, the computation of the schedule tree is usually very much data driven. Pipesort and most other methods make statistical estimates of the view sizes, based on the data available, and schedule tree construction is based on those view sizes. In our case, we could allow each processor P_j to build its own local schedule tree for its local data set D_i-root$||_j$ and build its D_i-partition accordingly. However, different local schedule trees for different processors imply that views of the D_i-partition created on different processors may be in different sort orders. This creates a problem during the subsequent merge phase in Step 3 of Procedure 1. When views of the same partition but for different subsets of data (i.e. on different processors) need to be merged, they need to have the same sort order or one of them has to be re-sorted. That re-sort creates a large amount of additional computation. Another possibility is to let one processor, say P_0, build the schedule tree for its data set D_i-root$||_0$, broadcast that schedule tree, referred to as the *global schedule tree*, and then let all processors use the same global schedule tree for their local cube construction. The advantage of this method is that we do not need to change the sort order of views during the merge. A potential disadvantage is that the sequential, local, top-down data cube methods (e.g. Pipesort) may not be using the "optimal" schedule tree for their data set. Recall that, the schedule trees generated by Pipesort and other top-down sequential methods are based on size *estimates*. As discussed in Section 4.2, our experiments indicate that, among the above two approaches, the latter method is far superior. For the data sets that we tested, the additional work on some processors because of non-optimal global schedule trees was much less than the overhead created through the need to re-sort views during the merge in Step 3. Therefore, Steps 2a, 2b and 2c of Procedure 1 implement the latter *global schedule tree* method.

2.4. Merge of local D_i-partitions

At the end of Step 2 of Procedure 1, each processor P_j has computed the D_i-partition for its local data set. For a view v of the D_i-partition, let v_j be the view created by processor P_j. In Step 3 of Procedure 1 we need to merge, for each view v in the D_i-partition, the p different views v_j created on the p different processors P_j. This merge is performed in Procedure Merge–Partitions(D_i) which will be discussed in the remainder of this section.

Consider Procedure 1 for $i = 0$ and the A-partition shown in figure 3. In Step 1 of Procedure 1, the A-roots are globally sorted by ABCD. Then, in Step 2, each processor P_j computes locally the A-partition for its data set. Consider the views ABCD$_j$, ABC$_j$, AB$_j$, and A$_j$ computed in Step 2. All these views are in the same sort order as the global sort order created in Step 1 because they are a prefix of ABCD. We shall refer to these views as the *prefix views*. The other views, ABD_j, AC_j, ACD_j and AD_j, are not a prefix of ABCD and are therefore in a sort order that is different from the global sort order. We shall refer to them as the *non-prefix views*.

Consider a prefix view v and the problem of merging $v_0, \ldots, v_{p-1}$ stored on processors $P_0, \ldots, P_{p-1}$. For example, consider the view v = AB in figure 3 and the problem of merging $\text{AB}_0, \ldots, \text{AB}_{p-1}$. The goal is to obtain a global AB sort order for $\text{AB}_0 \cup \text{AB}_1 \ldots \cup \text{AB}_{p-1}$ and then agglomerate those items that have the same values for dimensions A and B. Since AB is a prefix of the global sort order, ABCD, the first part is already done and the only items that, potentially, need to be agglomerated are the last item of v_j and the first item if v_{j+1} for each $0 \leq j < p-1$. Therefore, in Procedure Merge–Partitions(D_i), for each prefix view v every processor P_{j+1} simply sends the first item of v_{j+1} to processor P_j which compares it with the last item of v_j. Nothing else needs to be done in order to merge all v_j. figure 4 illustrates the case of a prefix view v, referred to as Case 1.

We now study the case of merging the views $v_0, \ldots, v_{p-1}$ stored on processors $P_0, \ldots, P_{p-1}$ for a non-prefix view v. For example, consider the view v = AC in figure 3 and the problem of merging $\text{AC}_0, \ldots, \text{AC}_{p-1}$. Again, the goal is to obtain a global AC sort order for $\text{AC}_0 \cup \text{AC}_1 \ldots \cup \text{AC}_{p-1}$ and then agglomerate those items that have the same values for dimensions A and C. However, AC is *not* a prefix of ABCD and, therefore, the different v_j can have considerable overlap with respect to AC order. Figure 4 illustrates the cases considered for a non-prefix view v, referred to as Case 2 and Case 3. The rectangles represent the v_j with respect to AC order. The shaded areas represent the overlap which, in contrast to Case 1 (prefix view), can now be considerably more than just one element. In Procedure Merge–Partitions(D_i), for each non-prefix view v every processor P_j sends its last element to every other other processor. Each processor P_k then determines its overlap with each P_j and sends that overlap to P_j. For each processor P_j let v'_j be the view v_j plus all the overlap received by processor P_j. We distinguish two cases as illustrated in figure 4, based on an imbalance threshold parameter γ. Case 2: IF $\mathcal{I}(|v'_0|, |v'_1|, \ldots |v'_{p-1}|) \leq \gamma$ for a non-prefix view v THEN each P_j locally sorts v'_j and agglomerates the items

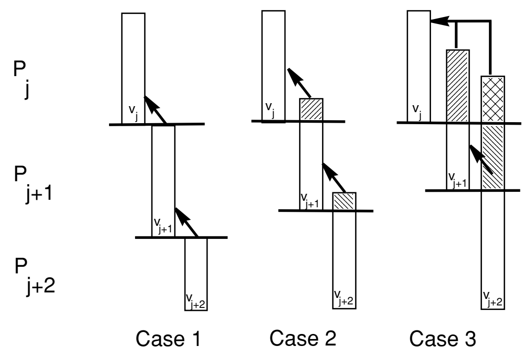

Figure 4. Illustration of cases in Procedure Merge–Partitions.

with same values for dimensions in v. Case 3: IF $\mathcal{I}(|v'_0|, |v'_0|, \ldots |v'_0|) > \gamma$ for a non-prefix view v THEN the v_j are merged by a global sort. The distinguishing criterion between Cases 2 and 3 is the imbalance between the v'_j. If the imbalance is smaller than γ (Case 2) then we proceed similar to Case 1. If the imbalance is larger than γ (Case 3) then we need to completely re-balance via a global sort. In fact, for Case 3 we do not wish to even route the overlap between processors. We rather re-sort immediately. Hence, in order to determine whether Case 2 or Case 3 applies, each processor P_k first determines the *size* of its overlap with each P_j and sends only the information about the size of that overlap to P_j.

The following is an outline of Procedure Merge–Partitions(D_i).

Procedure 3. Merge–Partitions(D_i)

(1) For each view $v \in DC_i$, each processor P_j broadcasts the last item of v_j to every other processor P_k and receives back the seizes of all overlaps.
(2) For each view $v \in DC_i$, every processor P_j determines $|v'_j|$ and sends all of its $|v'_j|$ values to processor P_0.
(3) Processor P_0 determines for each view $v \in DC_i$ whether Case 1, Case 2, or Case 3 applies.
(4) Every processor P_{j+1} sends for each Case 1 view $v \in DC_i$ the first item of v_{j+1} to processor P_j, and P_j compares/agglomerates that item with the last item of v_j.
(4) Every processor P_k sends for each Case 2 view $v \in DC_i$ its overlap with every v_j to the respective processor P_j. Every processor P_j merges/agglomerates all received overlaps with v_j.
(5) All remaining Case 3 views $v \in DC_i$ are merged via global sort, using Procedure 2.

In the following, we discuss a small extension of Procedure Merge–Partitions(D_i). Note that, all data sets are stored in secondary memory. For Step 2, a straight forward implementation would imply that the entire disk needs to be scanned on each processor in order to determine the $|v'_j|$ values. However, we do actually only require a $1/p\%$ accuracy for the $|v'_j|$ values in order to obtain a 1% accuracy for the imbalance $\mathcal{I}$ which is sufficient for distinguishing between cases 2 and 3. Hence, it is sufficient to use a sample of only 100 p equal spaced sample elements of the locally sorted v_j instead of the entire v_j. Such a sample can be build during the local computation of v_j in Step 2c of Procedure 1. Note that, while P_j writes v_j to its local disk, the size of v_j is not yet known and hence, the size of the sample and the distance between sample elements is not yet known. This problem can be solved as follows. A sample array $A[1 \ldots a]$ of size $a = 100p$ is allocated in main memory. While the first a elements of v_j are written to disk, each of them is also copied into A. While the second a elements of v_j are written to disk, every second is written into every second location of A, overwriting the previous element stored at that location. While the third and fourth groups of a elements of v_j are written to disk, every fourth is written into every second location of A, and so on. When the entire view v_j is written to disk, an appropriate sample will be available in array A to determine the $|v'_j|$ values with sufficient accuracy.

3. Partial data cube construction on shared-nothing multiprocessors

Our method has the advantage that it is easily extended to the case where not the entire data cube but only a subset of selected views are to be computed. This case occurs frequently in practice because the user often knows that some views will not be required for the subsequent OLAP queries that are executed on the data cube. For example, for a raw data set with 20 dimensions, it may be clear from the application that the OLAP queries will only require views with at most 5 dimensions. Therefore, it would be wasteful to create all 2^{20} views when most of them are never used.

For the purpose of computing a partial data cube, we redefine S. Instead of being the set of all 2^d view identifiers (as defined in Section 2.1), we define S to be the view identifiers of the subset of selected views. The definitions of S_i, DC_i, etc. are then all with respect to the new set S of *selected* views. The algorithm in Section 2 remains completely unchanged except for the construction of the schedule tree T_i in Step 2a of Procedure 1. Instead of using the first phase of an arbitrary sequential top-down data cube method, we apply the sequential schedule tree construction method for partial cubes that we have recently presented in [5]. For a set S of selected views, our algorithm in [5] can create a schedule tree that is either a subtree of the tree that would be generated by Pipesort for the entire cube, or it can create a schedule tree directly from the lattice. An example of a schedule tree for a partial cube is shown in figure 1(c). Note that, for optimal performance, some "intermediate" views need to be constructed in addition to the selected views.

4. Performance evaluation

We have implemented our parallel shared-nothing data cube generation method using C++ and the MPI communication library. This implementation evolved from the code base for a fast sequential Pipesort [3] and the sequential Partial cube method described in [5]. Most of the required sequential graph algorithms, as well as data structures like hash tables and graph representations, were drawn from the LEDA library [15].

Our experimental platform consisted of a 16 node Beowulf cluster with 1.8 GHz Intel Xeon processors, 512 MB RAM per node and two 40 GB 7200 RPM IDE disk drives per node. Every node was running Linux Redhat 7.2 with gcc 2.95.3 and MPI/LAM 6.5.6. as part of a ROCKS cluster distribution. All nodes were interconnected via an Intel 100 Megabyte Ethernet switch. Note that on this machine communication speed is extremely slow in comparison to computation speed. We will shortly be replacing our 100 Megabyte interconnect with a 1 Gigabyte Ethernet interconnect and expect that this will further improve the relative speedup results obtainable on this machine.

In the following experiments all sequential times were measured as wall clock times in seconds. 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*. All times include the time taken to read the input from files and write the output into files. Furthermore, all wall clock times were measured with no other user except us on the Beowulf cluster.

In our experimentation we generated a large number of synthetic data sets which varied in terms of the following parameters: n—number of records, d—number of dimensions,

$|D_0|, |D_1| \dots |D_{d-1}|$—cardinality in each dimension, and $\alpha_0, \alpha_1 \dots \alpha_{d-1}$—skew in each dimension.

Our experiments explored the following six performance issues:

1. *Relative speedup*: We investigated the effect of increasing the number of processors on the time required to solve data cube generation problems and measured the relative speedup, i.e. the ratio between observed sequential time and observed parallel time. Sequential times for computing full cubes and partial cubes were measured on a single processor of our parallel machine using our sequential implementations of Pipesort [3] and Partial cube [5], respectively.
2. *Local vs. global schedule trees*: We compared the effect on parallel wall clock time of using local vs. global schedule trees.
3. *Data skew*: We investigated the effect on parallel wall clock time of data sets with varying skewed distributions. We used the standard ZIPF [26] distribution with $\alpha = 0$ (no skew) to $\alpha = 3$ (high skew) and explored the relationship between data skew and the amount of data that must be communicated.
4. *Cardinality of dimensions*: We investigated the effect of varying dimension cardinalities on parallel wall clock time for both skewed and non-skewed data sets.
5. *Data dimensionality*: We investigated the effect of varying dimensionality, and therefore the effects of relative density or sparsity, on parallel wall clock time.
6. *Balance Tradeoffs*: Lastly, we investigated the effect of varying the imbalance threshold parameter γ. As γ is decreased we improve the balance in the distribution of views across processors, but at the cost of more data movement.

4.1. Relative speedup

Speedup experiments are at the heart of the experimental evaluation of our parallel shared-nothing data cube generation method. They consist of incrementally increasing the number of processors available to our data cube generation software to determine the parallel speedup obtained. Figure 5 shows for *full cube construction* the parallel wall clock time observed for data sets of varying sizes as a function of the number of processors used, and the corresponding relative speedup. We observe that for an input size $n = 2,000,000$ rows (72 Megabytes) our method achieves close to optimal speedup; generating a full data cube of ≈227 million rows (5.6 Gigabytes) in just under 6 minutes on 16 processors. The speedup for smaller problems is lower as there is insufficient local computation over which to amortize the cost of communications. On the other hand, our method works well on very large data sets. For example, on an input data set of 10,000,000 rows (360 Megabytes), our parallel method created the corresponding data cube consisting of ≈846 million rows (21.7 Gigabytes), on 16 processors, in under 47 minutes.

Note that our speedup results could be further improved by overlapping communication and local computation. Our current implementation does not overlap the local computation of D_i-Partitions with the global communication involved in merging D_{i-1}-Partitions. Doing so would mask between 40% and 60% of the communication overhead and further improve the speedup results.

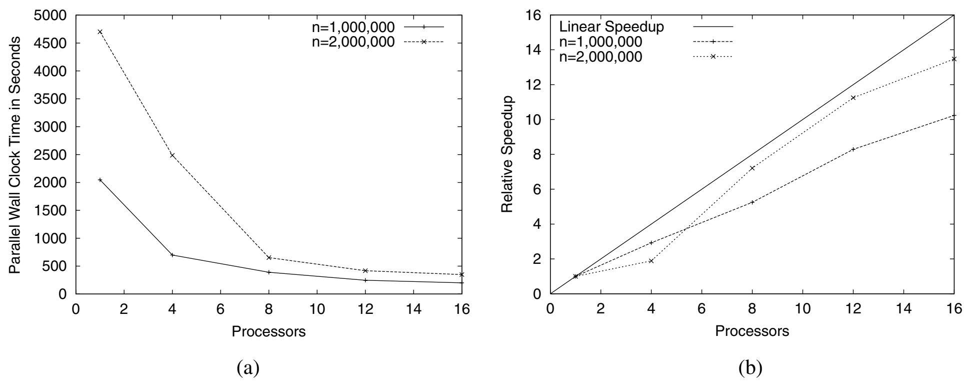

Figure 5. (a) Parallel wall clock time in seconds as a function of the number of processors for data of size $n = 1{,}000{,}000$ rows and $n = 2{,}000{,}000$ rows and (b) corresponding speedup. (Fixed parameters: Dimensions $d = 8$. Cardinalities $|D_i| = 256, 128, 64, 32, 16, 8, 6, 6$. Skew $\alpha = 0$. Percentage of views selected $k = 100\%$.)

Figure 6 shows for *partial cube construction* the parallel wall clock time observed for a range of different percentages of selected views as a function of the number of processors, and the corresponding relative speedup. We observe that when 50% or 75% of the views are selected, the speedup obtained decreases somewhat in comparison to the full cube case. However, for as little as 25% selected views, the speedup obtained is still more than half of optimal. Only when the number of views selected gets very small, approaching d, speedup falls off rapidly as the local work within partitions is little more than the computation of the root view. In such cases, when there are only a handful of selected views, creating each view from an independent sort of the original data set may be preferable.

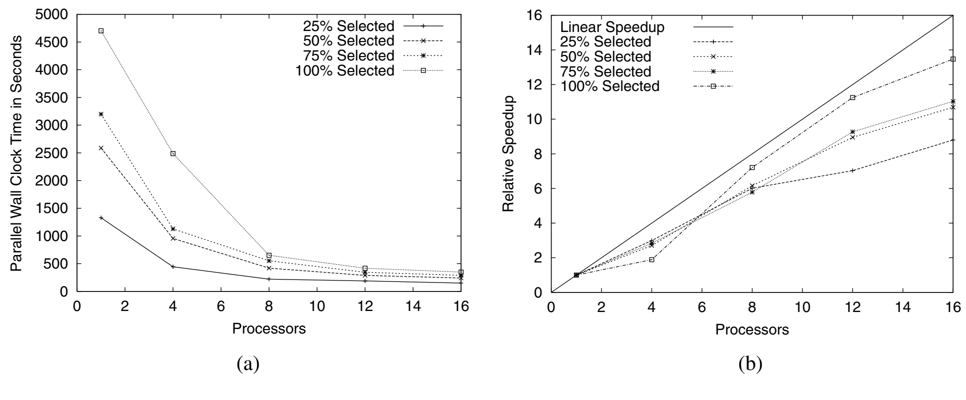

Figure 6. (a) Parallel wall clock time in seconds as a function of the number of processors for a range of different percentages of selected views and (b) corresponding speedup. (Fixed parameters: Data size $n = 2{,}000{,}000$ rows. Dimensions $d = 8$. Cardinalities $|D_i| = 256, 128, 64, 32, 16, 8, 6, 6$. Skew $\alpha = 0$.)

4.2. *Local vs. global schedule trees*

As described in Section 2.3, for shared-nothing parallel data cube construction it is possible for each processor to use either a local or a global schedule tree. Local schedule trees are built by each processor P_j relative to their own data set D_i-root$||_j$, whereas a global schedule tree is built by a single processor, say P_0, relative to its data set D_i-root$||_0$, and then broadcast to all other processors.

The use of local schedule trees might appear at first preferable, since they are optimized relative to a processor's own data set. However, they have one serious drawback. When views of the same partition but for different subsets of data (i.e. on different processors) need to be merged, they need to have the same sort order or one of them has to be re-sorted. That re-sort creates a large amount of additional computation. As can be seen in figure 7, our experiments indicate that local schedule trees offer superior performance in practice. For the data sets that we tested, the additional work on some processors because of non-optimal schedule trees was significantly less than the overhead created through the need to re-sort views during the Merge–Partitions( ) procedure.

4.3. *Data skew*

Data sets with skewed distributions can pose an interesting challenge to parallel data cube generation methods. As skew increases, data reduction tends to increase, particularly in top-down generation methods [1, 20]. Data reduction is typically positive, as it reduces the total amount of work to be performed. However, if data reduction is large and unevenly spread across the processors, it may unbalance the computation and cause the amount of data that has to be communicated to rise sharply.

To explore this issue we generated data sets using the standard ZIPF [26] distribution in each dimension with $\alpha = 0$ (no skew) to $\alpha = 3$ (high skew). Figure 8(a) shows the impact of skew on parallel wall clock time. Figure 8(b) shows, for the same data sets,

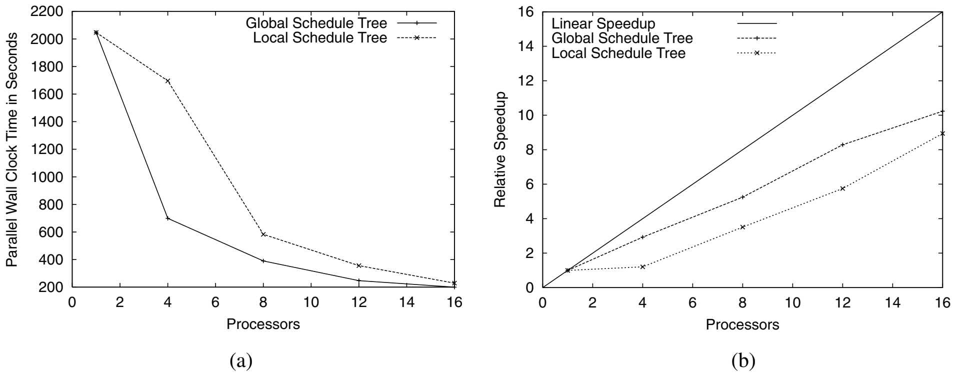

Figure 7. (a) Parallel wall clock time in seconds as a function of the number of processors for local and global schedule tree methods and (b) corresponding speedup. (Fixed parameters: Data size n = 1,000,000 rows. Dimensions $d = 8$. Cardinalities $|D_i|$ = 256, 128, 64, 32, 16, 8, 6, 6. Skew $\alpha = 0$. Percentage of views selected $k = 100\%$.)

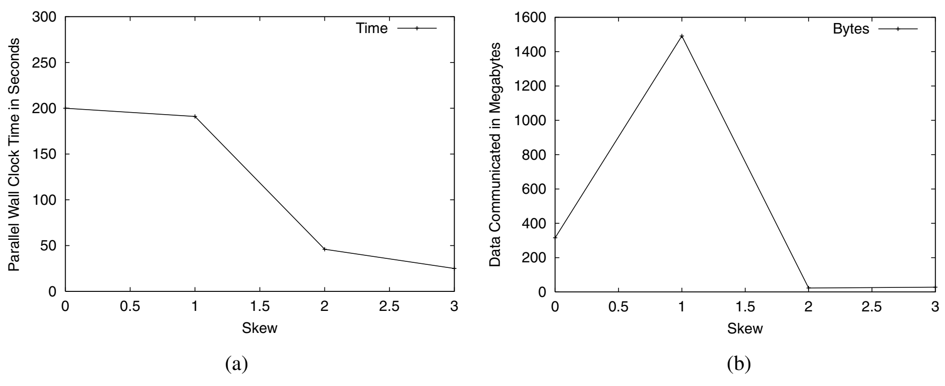

Figure 8. (a) Parallel wall clock time in seconds as a function of the skew for $\alpha = 0, 1, 2, 3$, and (b) the size of corresponding data movements. (Fixed parameters: Data size n = 1,000,000 rows. Dimensions $d = 8$. Cardinalities $|D_i|$ = 256, 128, 64, 32, 16, 8, 6, 6. Percentage of views selected k = 100%. Number of processors $p = 16$).

how skew affects the overall size of the data that must be communicated in performing the Merge–Partitions( ) procedure. We observe that, in general, as skew is increased parallel time decreases due to data reduction and decreased local computation. For $\alpha = 1$ there is a sharp rise in the amount of data to be communicated, which offsets some gains from the reduced local computation. However for $\alpha > 1$ the data reduction is so significant that only very little data needs to be communicated and parallel time drops significantly.

4.4. *Cardinality of dimensions*

The cardinality of the dimensions in a data set can affect the performance of our method. As cardinalities increase so does the sparsity of the data set and this may adversely effect parallel time especially given that top-down methods [1, 20] are designed primarily for dense data cubes. Curves A, B and C of figure 9(a) clearly illustrate this effect. The sparser data sets require somewhat more time, although, as can be seen in figure 9(b), this has little effect on the relative speedup achieved.

A close examination of the details of our algorithm suggests that a potentially difficult input data set for our method would be one in which the leading dimension has high skew and large cardinality, while the remaining dimensions have low skew. In such cases, the global sort used to create the D_0-*root* may do little to reduce the amount of communication required in building the views in DC_0. Curve D in figure 9 shows the results measured for such a case. We observe that whereas such situations do reduce the speedup obtained, the reduction is relatively small. Even for this difficult data set, the speedup obtained by our method is still close to half of the optimal speedup.

4.5. *Data dimensionality*

Figure 10 shows parallel wall clock time in seconds as a function of the dimensionality of the raw data set. Note that, the number of views that must be computed grows exponentially

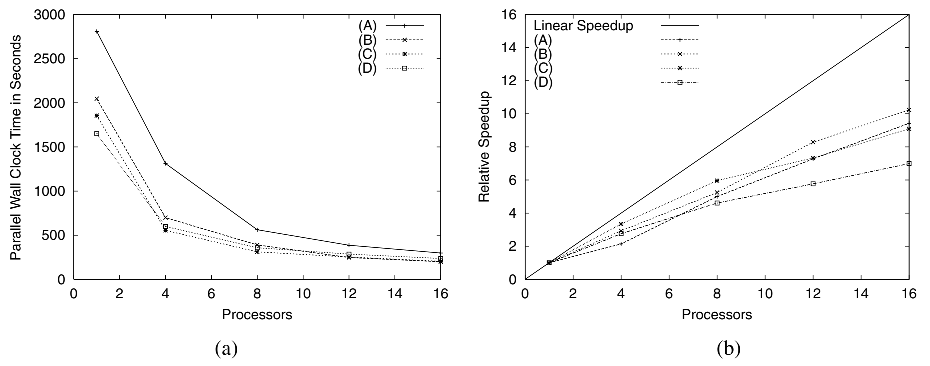

Figure 9. (a) Parallel wall clock time in seconds as a function of the number of processors for data sets with different cardinality mixes, and (b) corresponding relative speedup. (Fixed parameters: Data size $n = 1{,}000{,}000$ rows. Dimensions $d = 8$. Cardinalities and skews (A) $|D_i| = 256, 256, 256, 256, 256, 256, 256, 256$. Skew $\alpha = 0$. (B) $|D_i| = 256, 128, 64, 32, 16, 8, 6, 6$. Skew $\alpha = 0$. (C) $|D_i| = 16, 16, 16, 16, 16, 16, 16, 16$. Skew $\alpha = 0$. (D) $|D_i| = 256, 128, 64, 32, 16, 8, 6, 6$. Skew $\alpha_0 = 3$ and $\alpha_{i>0} = 0$.)

with respect to the dimensionality of the data set. In figure 10, we observe that the parallel running time grows essentially linearly with respect to the output size.

4.6. *Balance tradeoffs*

One important feature of our shared-nothing data cube generation algorithm is that it balances *each* view in the generated data cube over the processors within imbalance threshold γ. The more balanced each view is across the processors (i.e. the smaller γ) the more balanced any subsequent parallel computation on each view will be. However, the cost of selecting a small γ is that it may cause more data movement and therefore increase the time required for data cube generation.

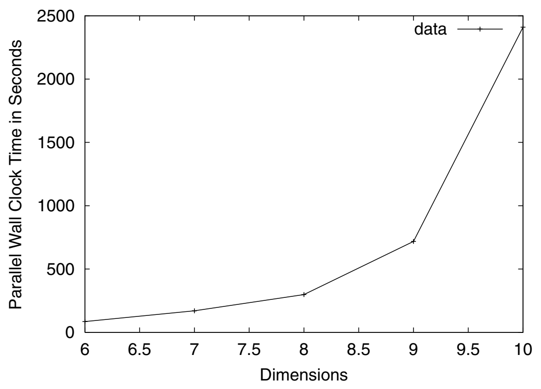

Figure 10. Parallel wall clock time in seconds as a function of the the number of dimensions. (Fixed parameters: Data size $n = 1{,}000{,}000$ rows. Cardinalities $|D_i| = 256$ in all dimensions. Percentage of views selected $k = 100\%$. Number of processors $p = 16$.)

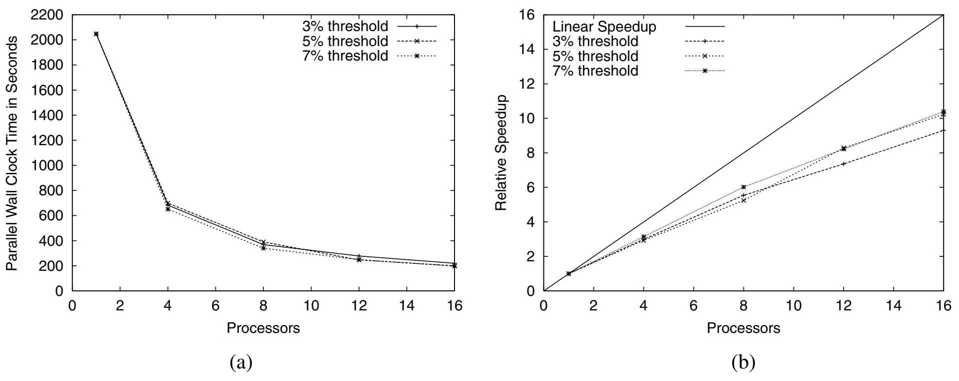

Figure 11. (a) Parallel wall clock time in seconds as a function of the number of processors for a range of different imbalance thresholds γ and (b) corresponding speedup. (Fixed parameters: Data size $n = 1{,}000{,}000$ rows. Dimensions $d = 8$. Cardinalities $|D_i| = 256, 128, 64, 32, 16, 8, 6, 6$. Skew $\alpha = 0$. Balance threshold $\gamma = 3\%, 5\%$, and 7%.)

Figure 11 shows parallel wall clock time in seconds as a function of the number of processors for a range of different imbalance thresholds, as well as the corresponding speedup curves. We observe that while reducing the imbalance threshold γ increases the parallel time, the effect is small. A γ of 3% appears to be a good threshold in practice. However, individual applications may want to tune this parameter according to their needs and the performance characteristics of their parallel machines.

5. Conclusion

In this paper, we study parallel data partitioning methods for ROLAP data cubes that can be executed on shared-nothing multiprocessors. The principal advantage of ROLAP is that it allows for tight integration with current relational database technology. We have implemented our parallel data cube method and evaluated it on a PC cluster, exploring relative speedup, local vs. global schedule trees, data skew, cardinality of dimensions, data dimensionality, and balance tradeoffs. We obtained promising speedup results for a wide range of input data sets. We are currently exploring the integration of our method with commercial parallel database systems.

References

1. S. Agarwal, R. Agarwal, P. Deshpande, A. Gupta, J. Naughton, R. Ramakrishnan, and S. Srawagi, “On the computation of multi-dimensional aggregates,” in Proc. 22nd VLDB Conf., 1996, pp. 506–521.
2. K. Beyer and R. Ramakrishnan, “Bottom-up computation of sparse and iceberg cubes,” in ACM SIGMOD Conference on Management of Data, 1999, pp. 359–370.
3. F. Dehne, T. Eavis, S. Hambrusch, and A. Rau-Chaplin, “Parallelizing the data cube,” Distributed and Parallel Databases, vol. 11, no. 2, pp. 181–201, 2002.

4. F. Dehne, T. Eavis, and A. Rau-Chaplin, "A cluster architecture for parallel data warehousing," in Proc IEEE International Conference on Cluster Computing and the Grid (CCGrid 2001), Brisbane, Australia, 2001.
5. F. Dehne, T. Eavis, and A. Rau-Chaplin, "Computing partial data cubes," Technical report, http://www.cs.dal.ca/~arc/publications/2-30/paper.pdf, 2003.
6. P. Flajolet and G. Martin, "Probablistic counting algorithms for database applications," Journal of Computer and System Sciences, vol. 31, no. 2, pp. 182–209, 1985.
7. S. Goil and A. Choudhary, "High performance OLAP and data mining on parallel computers," Journal of Data Mining and Knowledge Discovery, vol. 1, no. 4, pp. 391–417, 1997.
8. S. Goil and A.N. Choudhary, "High performance multidimensional analysis of large datasets," in International Workshop on Data Warehousing and OLAP, 1998, pp. 34–39.
9. 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, 1999.
10. J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, and M. Venkatrao, "Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals," J. Data Mining and Knowledge Discovery, vol. 1, no. 1, pp. 29–53, 1997.
11. J. Han, Y. Fu, W. Wang, J. Chiang, W. Gong, K. Koperski, D. Li, Y. Lu, A. Rajan, N. Stefanovic, B. Xia, and O.R. Zaiane, "DBMiner: A system for mining knowledge in large relational databases," in Proc. 1996 Int'l Conf. on Data Mining and Knowledge Discovery (KDD'96), Portland, Oregon, 1996, pp. 250–255.
12. V. Harinarayan, A. Rajaraman, and J. Ullman, "Implementing data cubes efficiently," ACM SIGMOD Record, vol. 25, no. 2, pp. 205–216, 1996.
13. X. Li, P. Lu, J. Schaeffer, J. Shillington, P.S. Wong, and H. Shi, "On the versatility of parallel sorting by regular sampling," Parallel Computing, vol. 19, no. 10, pp. 1079–1103, 1993.
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. K. Mehlhorn and S. Naeher, LEDA. http://www.mpi-sb.mpg.de/LEDA/, 1999.
16. S. Muto and M. Kitsuregawa, "A dynamic load balancing strategy for parallel datacube computation," in ACM Second International Workshop on Data Warehousing and OLAP (DOLAP 1999), 1999, pp. 67–72.
17. S. Muto and M. Kitsuregawa, "A dynamic load balancing strategy for parallel datacube computation," in Proceedings of the Second ACM International Workshop on Data Warehousing and OLAP, ACM Press, 1999, pp. 67–72.
18. R. Ng, A. Wagner, and Y. Yin, "Iceberg-cube computation with pc clusters," in ACM SIGMOD Conference on Management of Data, 2001, pp. 25–36.
19. K. Ross and D. Srivastava, "Fast computation of sparse datacubes," in Proc. 23rd VLDB Conference, 1997, pp. 116–125.
20. S. Sarawagi, R. Agrawal, and A. Gupta, "On computing the data cube," Technical report rj10026, IBM Almaden Research Center, San Jose, CA, 1996.
21. A. Shukla, P. Deshpende, J. Naughton, and K. Ramasamy, "Storage estimation for mutlidimensional aggregates in the presence of hierarchies," in Proc. 22nd VLDB Conference, 1996, pp. 522–531.
22. J.S. Vitter, "External memory algorithms and data structures: Dealing with MASSIVE DATA," ACM Computing Surveys, vol. 33, no. 2, pp. 209–271, 2001.
23. J.S. Vitter and E.A.M. Shriver, "Algorithms for parallel memory I: Two-level memories," Algorithmica, vol. 12, nos. 2/3, pp. 110–147, 1994.
24. J. Yu and H. Lu, "Multi-cube computation," in Proc. 7th International Symposium on Database Systems for Advanced Applications, Hong Kong, 2001, pp. 126–133.
25. Y. Zhao, P. Deshpande, and J.F. Naughton, "An array-based algorithm for simultaneous multidimensional aggregates," in Proc. ACM SIGMOD Conf., 1997, pp. 159–170.
26. G. Zipf, Human Behavior and The Principle of Least Effort, Addison-Wesley, 1949.