Building Large ROLAP Data Cubes in Parallel [*]

Ying Chen
Dalhousie University
Halifax, Canada
ychen@cs.dal.ca

Frank Dehne
Carleton University
Ottawa, Canada
www.dehne.net

Todd Eavis
Carleton University
Ottawa, Canada
www.scs.carleton.ca/∼teavis

A. Rau-Chaplin
Dalhousie University
Halifax, Canada
www.cs.dal.ca/∼arc
(902) 494-2732, fax 492-1517
— contact author —

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. However, as the size of data warehouses grows, the time it takes to perform this pre-computation becomes a significant performance bottleneck. This paper presents a fast parallel method for generating ROLAP data cubes on a shared-nothing multiprocessor based on a novel optimized data partitioning technique. Since no shared disk is required, this method can be applied on highly scalable processor clusters consisting of standard PCs with local disks, connected via a data switch. The approach taken, which uses a ROLAP representation of the data cube, is well suited to large data warehouses on high dimensional data, and supports the generation of both fully materialized and partially materialized cubes. In comparison with previous approaches, our new method does significantly improve the scalability with respect to both, the number of processors and the I/O bandwidth (number of parallel disks).

We have implemented our new parallel shared-nothing data cube generation method and evaluated it on a PC cluster, exploring relative speedup, scaleup, sizeup, output sizes and data skew. For a fact table with 16 million rows and 8 attributes, our parallel data cube generation method achieves close to optimal speedup for as many as 32 processors, generating a full data cube in under 7 minutes. For a fact table with 256 million rows and 8 attributes, our parallel method achieves optimal speedup for 32 processors, generating a full data cube consisting of $\approx$ 7 billion rows (200 Gigabytes) in under 88 minutes.

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

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 [13] and can be instrumental in accelerating data mining tasks in large data warehouses [14]. As the size of data warehouses grows, the time it takes to perform this pre-computation becomes a significant performance bottleneck, one which may stretch into days in the very largest cases [1]. This paper presents a fast parallel method for generating ROLAP data cubes on a shared-nothing multiprocessor based on a novel optimized data partitioning technique. Since no shared disk is required, this method can be applied on highly scalable processor clusters consisting of standard PCs with local disks, connected via a high bandwidth (Ethernet) switch. Parallelism based on such shared-nothing machines is an attractive solution to improving system performance especially in the context of large data warehouses where scaling I/O bandwidth to disk is as important as scaling computational resources.

For a given raw data set, R, with N records and d attributes (dimensions), a view is constructed by an aggregation of R along a subset of attributes. As proposed in [13], the pre-computation of the full data cube (the set of all 2^d possible views) or a partial data cube (a subset of all 2^d possible views) supports the fast execution of subsequent OLAP queries. Many methods have been presented for generating the data cube on sequential [3, 15, 21, 22, 26, 27] and parallel systems [4, 5, 7, 10, 11, 17, 18, 20]. For parallel data cube construction, good data partitioning is a key factor in obtaining good performance on shared nothing multiprocessors. Some researchers partition data on one or several dimensions [19, 12]. They assume that the product of the cardinalities of these dimensions is much larger than the number of processors [12], in order to achieve sufficient parallelism. The advantage of their method is that they do not need to merge views across the network. 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. Therefore, those methods are often not scalable. One approach which avoids these problems is to partition on all dimensions and then apply a parallel merge procedure [4]. The challenge here is that merge procedures based on fixed data partitioning schemes often lead to excess inter-processor communications which may greatly reduce the speedup achieved by the parallel system and limit its effective scalability.

In this paper, we describe and evaluate an optimized data partition scheme for parallel ROLAP data cube generation. This dynamic data partitioning scheme adapts to both, the current data set and the performance parameters of the parallel machine. Using this scheme, data cube generation tasks involving millions of rows of input, that take days to perform on a single processor machine, can be completed in just hours on a 32 processor cluster. We have performed an extensive performance evaluation of our new method, exploring relative speedup, scaleup, sizeup, output sizes and data skew. For our experiments, our new optimized data partitioning method results in approximately twice the speedup achieved with fixed data partitioning. The optimized data partition scheme exhibited optimal, linear, speedup for full cube generation on as many as 32 processors, as well as excellent sizeup and scaleup behavior. For example, for a fact table with 16 million rows and 8 attributes, our parallel data cube generation method achieves close to optimal speedup for 32 processors, generating a full data cube in under 7 minutes. For a fact table with 256 million rows and 8 attributes, our parallel method achieves optimal speedup for 32 processors, generating a full data cube consisting of $\approx$ 7 billion rows (200 Gigabytes) in under 88 minutes.

In comparison with previous approaches, our new method has a significantly better scalability with respect to the number of processors. Optimal speedup for as many as 32 processors was not observed for previous parallel methods [19, 12, 4]. In addition, because of its shared nothing approach, our new method does also significantly improve the scalability with respect to the I/O bandwidth (number of parallel disks) which is of great importance for handling large data sets.

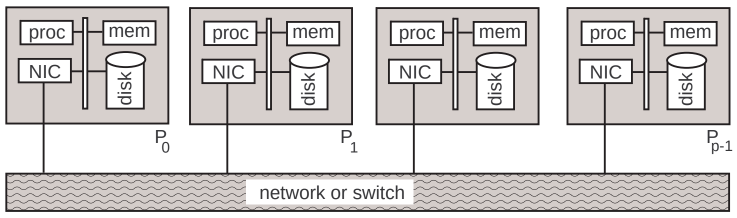
The remainder of this paper is organized as follows. In Section 2 we present an overview of the entire algorithm and in Section 3 we discuss in detail our new optimized data partitioning method. The performance of our method is discussed in Section 4.

2 Algorithm Overview

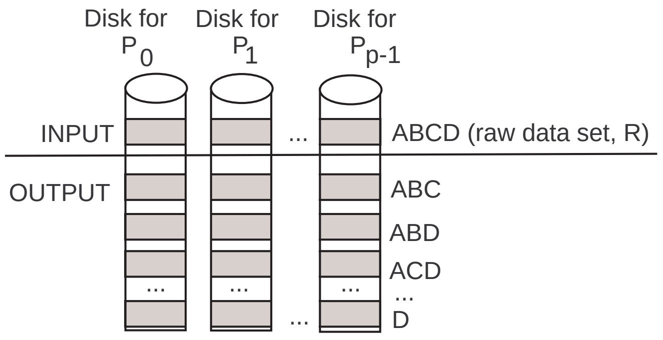
Consider a raw data set R with N rows and d attributes $D_1, \ldots, D_d$. Without loss of generality, let $|D_1| \geq |D_2| \geq \ldots \geq |D_d|$, where $|D_i|$ is the cardinality for dimension D_i, $1 \leq i \leq d$ (i.e. the number of distinct values for dimension D_i). As input, we assume a raw data set, R, with N records and d dimensions D_1, D_1 ... D_{d-1} distributed evenly over the p disks; see Figure 1b.

Let S be the set of all 2^d view identifiers. Each view identifier consists of a subset of $\{D_1, D_2$ ... $D_d\}$, ordered by the cardinalities of the selected dimensions (in decreasing order). Alternatively, S could also be a *subset* of the 2^d view identifiers selected by the users. We refer to this case as *partial* data cube construction. The input data set R is evenly distributed over the p disks of the p processors as shown in Figure 1b. The goal is to create a data cube DC containing the views in S. We assume that, when the algorithm terminates, every view is distributed evenly across the p disks; see Figure 1b. 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.

The basic communication operation used by our parallel data cube algorithm is the h-relation (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 [24]. We will present our



(a)



(b)

Figure 1: (a) Shared-Nothing Multiprocessor. (b) Data Partitioning.

method for a shared-nothing multiprocessor with one local disk per processor P_j. However, it is easy to generalize our methods for 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 [25].

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. DC_i is called the *i-subcube* and the view $D_i \ldots D_d$ is referred to as $Root_i$; see Figure 2. In the shared nothing environment considered in this paper, all data sets are distributed over the p disks of the p processors as shown in Figure 1b. We refer to the part of a data set stored on processor P_j as its *j-partition*. The j-partitions of R, DC_i, and $Root_i$ are denoted as R_j, DC_{ij}, and $Root_{ij}$, respectively.

Algorithm 1 describes the global structure of our parallel data cube construction algorithm for shared-nothing multiprocessors. The algorithm consists of d iterations $i = 1 \ldots d$. In iteration i, the i-subcube DC_i is created in five main steps: Computing $Root_i$, computing the schedule tree T_i, optimizing the partitioning of $Root_i$ into $Root_{i1}$ ... $Root_{ip}$, computing the local DC_{ij} from each $Root_{ij}$, and merging the DC_{ij} to obtain the correct i-subcube DC_i.

In the following, we will first present further details on Step 3 of Algorithm 1 (how the schedule tree T_i is built) and Step 6 (how locally generated i-subcubes are merged). Step 4, the optimized data partitioning, which is the main contribution of this paper, will be discussed in Section 3.

In Step 3, our parallel algorithm uses as a building block a standard sequential top-down data

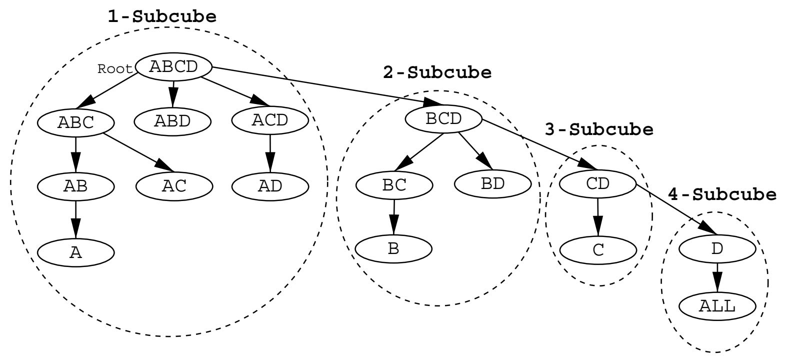


Figure 2: Subcubes of a data cube for $d = 4$. Dimensions are labelled D_1="A", D_2="B", D_3="C", D_4="D".

cube method such as Pipesort [22]. 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 the 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 [9, 23]. 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 and computing for each two subsequent levels of nodes, and the edges between them, a minimum cost bi-partite matching. We use Pipesort to compute the schedule tree T_i in Step 3 of Algorithm 1 if *all* 2^d views are to be computed. For building the *partial* data cube, i.e. a subset of the 2^d possible views, we use a modified schedule tree construction method presented in [6].

We now discuss how locally generated i-subcubes are merged in Step 6. In Step 5 of Algorithm 1, each processor P_j locally computes DC_{ij} from its local $Root_{ij}$. For a view v of S_i, let v_j be the local view created by processor P_j. We need to merge, for each view v in S_i, the p different views v_j created on the p different processors P_j. Consider Algorithm 1 for $i = 1$ and the 1-subcube shown in Figure 2. In Step 2 of Algorithm 1, $Root_0 = R$ is globally sorted by $ABCD$. In Step 5 , each processor P_j computes locally DC_{1j} from its data set $Root_{0j}$. Consider the views $ABCD_j$, ABC_j, AB_j, and A_j computed in Step 5. All these views are in the same sort order as the global sort order created in Step 2 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 $v_1, \ldots, v_p$ stored on processors $P_1, \ldots, P_p$. For example, consider the view v = AB in Figure 2 and the problem of merging $AB_1, \ldots, AB_p$. The

Algorithm 1 *Parallel–Shared–Nothing–Data–Cube*

```
Input: $R$, the raw data set; $N$, the number of rows in $R$; $d$, the number of attributes; $p$, the number
    of processors; $S$, the set of views to be generated. Every processor $P_j (1 \leqslant j \leqslant p)$ stores on its
    disk a set $R_j$ of $n/p$ rows of $R$.
Output: $DC$, the data cube distributed over the $p$ processors. Each view is evenly distributed over
    the $p$ processors' disks.
1: for $i = 1$ to $d$ do
2:     Compute $Root_i$ via a parallel global sort of $Root_{i-1}$ by key $D_i, \ldots, D_d$, where $Root_0 = R$. As
       a result, each processor $P_j$ stores a $j$-partition, $Root_{ij}$, of $Root_i$.
3:     Processor $P_0$ generates and broadcasts the schedule tree, $T_i$, for computing $S_i$ from $Root_i$.
4:     Execute Optimize–Partition($Root_i$) to obtain an optimized partitioning of $Root_i$ into $Root_{ij}$,
       $1 \leq j \leq p$.
5:     Every processor $P_j (1 \leqslant j \leqslant p)$ locally computes $DC_{ij}$ from its $Root_{ij}$ using the schedule tree
       $T_i$.
6:     Execute Merge–Subcube($DC_i$) to obtain the correct $i$-subcube $DC_i$.
7: end for
```

goal is to obtain a global AB sort order for $AB_1 \cup AB_2 \ldots \cup AB_p$ 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 $1 \leq j < p$. 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 3 illustrates the case of a prefix view v as "Case 1".

We now study the case of merging the views $v_1, \ldots, v_p$ stored on processors $P_1, \ldots, P_p$ for a non-prefix view v. For example, consider the view $v = AC$ in Figure 2 and the problem of merging $AC_1, \ldots, AC_p$. Again, the goal is to obtain a global AC sort order for $AC_1 \cup AC_2 \ldots \cup AC_p$ 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 the AC order. Figure 3 illustrates the case of a non-prefix view v 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. For each non-prefix view v, every processor P_j sends its first and last element to every 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 which are both illustrated in Figure 3. The distinguishing criterion is the imbalance between the v'_j defined as $\mathcal{I}(|v'_1|, |v'_2|, \ldots |v'_p|) = max\{(r_{max} - r_{avg})/r_{avg}, (r_{avg} - r_{min})/r_{avg}\}$ where r_{min}, r_{max},

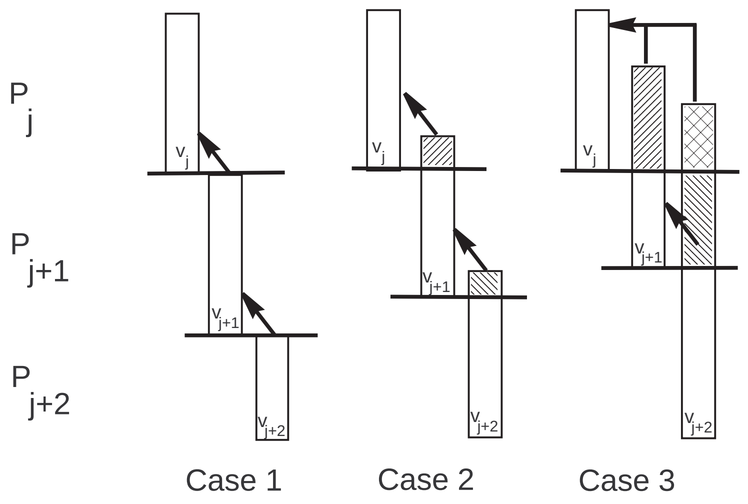


Figure 3: Illustration of cases for *Merge–Subcube*(DC_i).

and r_{avg} are the minimum, maximum and average of $\{|v'_1|, |v'_2|, \ldots |v'_p|\}$, respectively. “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 with same values for dimensions in v. “Case 3”: IF $\mathcal{I}(|v'_1|, |v'_2|, \ldots |v'_p|) > \gamma$ for a non-prefix view v THEN the v_j are merged by a global sort.

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.

3 Optimized Data Partitioning

Good data partitioning is a key factor in obtaining good performance on shared nothing multiprocessors. Some researchers partition data on one or several dimensions [19, 12]. They assume that the product of the cardinalities of these dimensions is much larger than the number of processors [12], in order to achieve sufficient parallelism. The advantage of their method is that they do not need to merge views across the network. For examples, if we partition on *A*, then *ABC* and *AC* do not need to be merged, or if we partition on *A* and *B*, then *ABC* and *ABD* do not need 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 parallel machine may be large, especially in clusters of workstations. Therefore, those methods are often not scalable. One approach which avoids these problems is to partition on all dimensions and then apply a parallel merge procedure [4]. The challenge here is that merge procedures based on fixed data partitioning schemes often lead to excess inter-processor communications which may greatly reduce the speedup achieved by the parallel system and hence its effective scalability. In this paper, we describe and evaluate an optimized dynamic data partition scheme for ROLAP data cube generation. This dynamic data partitioning scheme adapts to both, the current data set and the performance parameters of the parallel machine.

For a given parallel machine, we introduce four performance parameters $t_{compute}$, t_{read}, t_{write} and $t_{network}$ defined as follows: $t_{compute}$ is the average time in microseconds to fetch/compare/store a data item in main memory; t_{read} is the average time in microseconds to read a data item from disk; t_{write} is the average time in microseconds to write a data item to disk; $t_{network}$ is the average time in microseconds for communicating a data item between processors. For heterogeneous parallel machines (e.g. clusters with different generations of processors), the parameters $t_{compute}$, t_{read} and t_{write} can differ between processors. In this case, we choose the parameters for the slowest processor. The parameter $t_{network}$ depends on both, the network hardware and the number of processors used. Based on the above four parameters, we devise a cost model to estimate the time for communication and computation, and determine the best data partitioning for Algorithm 1. Before starting Algorithm 1, our software enters a test phase where it measures automatically the parameters $t_{compute}$, t_{read}, t_{write} and $t_{network}$ for the given machine.

After the i-th iteration of Step 2 of Algorithm 1, the j-partitions of the $Root_i$ are well balanced over processors P_j $(1 \leq j \leq p)$. However, as a result of the global sort, subsequent items with the same sort key may end up on two different (subsequent) processors. This is especially the case when the cardinality of some dimensions is small, such as for attributes like gender, months and intervals for a numeric attribute. The situation is illustrated in Figure 4 for an attribute "A" with attribute values $a1, a2, \ldots, a10$. When the data is sorted by "A" in Step 2, each processor receives a range of data as indicated. Consider the range of items with value $a4$. Some items are on Processor 1 and some are on Processor 2. The problem is that during the merging of subcubes in Step 6 of Algorithm 1, data movement occurs because Processor 2 has to send its items with value $a4$ to Processor 1. Instead, we could have made $a4$ the dividing line between the data between Processors 1 and 2 and moved all items with value $a4$ to Processor 2. We call this process "pivoting" and refer to $a4$ as the pivot. If we choose $a4$ as a pivot, then no data will have to be transferred between Processors 1 and 2 during the merging of subcubes in Step 6 of Algorithm 1. However, on the negative side, choosing $a4$ as a pivot introduces an imbalance in data size between Processors 1 and 2, and other steps of Algorithm 1 may now have a longer computation time because of this imbalance, since the total computation time is always determined by the slowest processor.

Our strategy is to choose pivots in such a way that we obtain the best tradeoff between lower communication due to less data movement and longer computation due to imbalance. We build a cost model to measure the impact of each possible pivot and choose the one with the lowest cost.

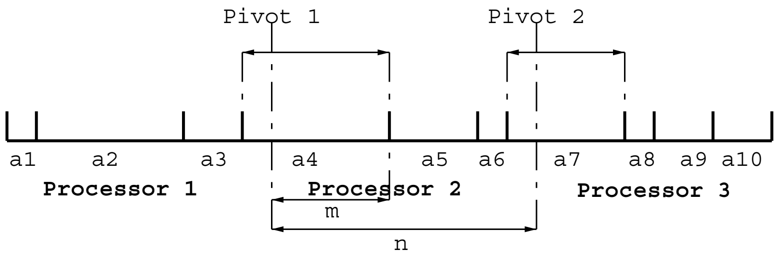


Figure 4: Data partitioning and *pivots*.

We iterate this process until the total cost can be no further reduced.

We now discuss our cost model for the performance of Algorithm 1 with respect to a chosen set of pivots. Note that Steps 2 and 3 are not impacted by pivots. Our model therefore measures only the performance of Step 4 (shifting partitions), Step 5 (computing cubes), and Step 6 (merging cubes).

An important factor to be taken into consideration is the impact of external memory. For our implementation of Algorithm 1, views that are small enough to fit into main memory are created in memory for better speed, while larger views are built in external memory through disk scan and external memory sort. In order to determine which version is used at run time, we define a maximal number of records, n_{max}. If the number of records of a view is smaller than n_{max}, we calculate the cost according to a formula for internal memory computation. Otherwise, we calculate the cost according to a formula for external memory computation. For example, if n_{max} is $1,000,000$, view *ABCD* has $2,000,000$ records and view *ABC* has $500,000$ records, then we process *ABC* in main memory using the internal memory cost calculation and process *ABC* in external memory using the external memory cost calculation.

To calculate the cost of Steps 4-6 of Algorithm 1 for each view v, we use two basic numbers for each processor: n, the number of records stored at the processor and m, the number of moved records. Figure 4 illustrates n and m for Processor 2. The n and m values for $Root_i$ are obtained through a local linear scan. For every other view v in DG_i, we estimate values n_{est} and m_{est} as follows: Set n to the estimated view size calculated in Step 3 of Algorithm 1. Set $m = \frac{m_{Root_i}}{n_{Root_i}} n$ where n_{Root_i} and m_{Root_i} are the n and m values for $Root_i$, respectively. Note that a record is composed of d feature attributes and 1 measure attribute so that the size of a record is proportional to $d+1$.

We are now ready to analyze Sets 4 to 6 of Algorithm 1. For each step, we will give the cost for internal and external memory calculation and outline our rationale for the give formulas.

Step 4. Scanning: $n(d/2) * t_{compute}$ (internal memory), $n(d+1) * t_{read} + n(d/2) * t_{compute}$ (external memory). Exchanging: $m(d+1) * t_{network}$ (internal memory), $m(d+1) * t_{network}$ (external memory). Merging: $n(d/2) * t_{compute}$ (internal memory), $n(d+1) * t_{write} + n(d/2) * t_{compute}$ (external memory). Rationale: Step 4 in Algorithm 1 shifts partitions of root views among processor. It consists of three sub-steps: scanning data, exchanging data and merging data. Each processor scans the local data and compares each row with the pivots considerd. To compare a row with a pivot, we compare attribute values one by one. In the best case, only one comparison is needed, and d comparisons in the worst case, where d is the number of attributes. The average number of comparisons is $d/2$. In the external memory version, the cost for reading data from disk is $n(d+1)*t_{read}$, where $n(d+1)$ is the number of item in $Root_i$ since each row contains $d+1$ items. In both versions, the communication cost is $m(d+1)*t_{network}$, where $m(d+1)$ is the number of items moved across the network. The cost of the last sub-step is $n(d/2)*t_{compute}$. For the merging, the number of comparisons is a function of both, n and m. However m is much smaller than n and we ignore m in order to simplify calculations. In the external version, the cost for writing the data to disks is $n(d+1)*t_{write}$. Note that, this is also an approximation since data is exchanged between processors.

Step 5. Sorting: $n\log n*t_{compute}+n(d/2)*t_{compute}$ (internal memory), $n(d+1)*t_{read}+n\log n*t_{compute}+n(d/2)*t_{compute}$ (external memory). Scanning: $n(d+1)*t_{compute}$ (internal memory), $n(d+1)*t_{write}+n(d+1)*t_{compute}$ (external memory). Rationale: Step 5 of Algorithm 1 calculates the schedule tree used to generate the views. As described in [22], we compute pipelines one by one. For each pipeline, the first view is sorted and the remaining views are generated by scanning. For example, in Figure 2, the schedule tree for the 1-subcube consists of a pipeline, $ABCD \rightarrow BCD \rightarrow BC \rightarrow B$. The cost of sorting is $n\log n*t_{compute}+n(d/2)*t_{compute}$ [6] for the internal memory version. The external version includes an additional cost for disk reading: $n(d+1)*t_{read}$. The cost for scanning is $n(d/2)*t_{compute}$ for the internal memory version, plus $n(d+1)*t_{write}$ for the external memory version.

Step 6. Scanning: $n(d/2)*t_{compute}$ (internal memory), $n(d+1)*t_{read}+n(d/2)*t_{compute}$ (external memory). Exchanging: $m(d+1)*t_{network}$ (internal memory), $m(d+1)*t_{network}$ (external memory). Merging: $n(d/2)*t_{compute}$ (internal memory), $n(d+1)*t_{write}+n(d/2)*t_{compute}$ (external memory). Rationale: Step 6 of Algorithm 1 merges i-subcubes between processors. The cost calculation is similar to the calculation for Step 4.

Based on the above cost model, we may evaluate possible partitionings and choose an optimal partition with minimum cost. Algorithm 2 shows our method to select a set of pivots and shift the partitions. The function $Cost()$ represents the cost function for a given set of pivots as discussed above. Algorithm 2 first calculates the cost of the partitioning generated by Steps 2 and 3 of Algorithm 1 without any pivots. We then select pivots, calculate the cost based on those pivots and update the partitioning if the new cost is smaller than the old one. This process will continue until the cost can not be reduced any further. Unfortunately, the number of possible pivot combinations is very high. For p processors, the maximum number of possible pivots is $p-1$. Each pivot can either be not selected or selected for its left adjacent processor (all data move left) or its right adjacent processor (all data move right). Hence, the total number of possible data partitionings is 3^{p-1}. If we have 32 processors in a cluster, the total number of partitionings is $3^{32-1}=617,673,396,283,947$.

In Algorithm 2, we choose a greedy method to reduce the cost as much as possible. In each iteration of the repeat-until loop, we choose the pivot which generates the greatest cost reduction among all possible remaining pivots. We update the partitioning and the cost, and search again until we cannot reduce the cost further by adding another pivot. Algorithm 2 then re-partitions $Root_i$ using the chosen set of pivots.

Algorithm 2 *Optimize–Partition*$(Root_i)$

```
 1: Each processor $P_j$ collects locally, for its data set $Root_{ij}$, the partitioning information (pivots
    and their $n$, $m$ values) required for the evaluation of the function $Cost()$. The partitioning
    information is broadcast to all processors.
 2: Each processor $P_j$ computes $cost = Cost$(current partition without pivots).
 3: done = FALSE.
 4: repeat
 5:   for each processor $P_j$ in parallel do
 6:     Processor $P_j$ calculates the new cost $cost_j^{new}$ obtained by adding pivot $j$, (moving the re-
        spective data to the left or right processor, whichever is lower cost).
 7:   end for
 8:   Let $cost^{new} = Min(cost_1^{new}, cost_2^{new}, ..., cost_{p-1}^{new})$
 9:   if $cost^{new} < cost$ then
10:     update partition by adding the chosen pivot.
11:     $cost = cost^{new}$
12:   else
13:     done = TRUE
14:   end if
15: until done
16: $Root_i$ is re-partitioned using the chosen set of pivots.
```

4 Performance Evaluation

We have implemented our optimized data partitioning method for shared-nothing data cube generation using C++ and the MPI communication library. This implementation evolved from the code base for a fast sequential Pipesort [5] and the sequential partial cube method described in [6]. Most of the required sequential graph algorithms, as well as data structures like hash tables and graph representations, were drawn from the LEDA library [16].

Our experimental platform consists of a 32 node Beowulf style cluster with 16 nodes based on 2.0 GHz Intel Xeon processors and 16 more nodes based on 1.7 GHz Intel Xeon processors. Each node was equipped with 1 GB RAM, two 40GB 7200 RPM IDE disk drives and an onboard Inter

Pro 1000 XT NIC. Each 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 a Cisco 6509 GigE switch.

Our implementation of Algorithm 1 initially runs a performance test to calculate the key machine specific cost parameters, $t_{compute}$, t_{read}, t_{write} and $t_{network}$, that drive our optimized dynamic data partitioning method. On our experimental platform these parameters were as follows: $t_{compute} =$ 0.0293 microseconds, t_{read} = 0.0072 microseconds, $t_{write} =$ 0.2730 microseconds. The network parameter, $t_{network}$, captures the performance characteristics of the MPI `MPI_ALL_TO_ALL_v` operation on a fixed amount of data relative to the number of processors involved in the communication. On our experimental platform, $t_{network} = 0.0551, 0.0873, 0.1592, 0.2553, 0.4537, 0.5445$ microseconds for p = 2, 4, 8, 16, 24, and 32, respectively.

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 users on the machine.

For this series of experiments 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, $|D_1|, |D_2| \ldots |D_d|$ - cardinality in each dimension, and $\alpha_1, \alpha_2 \ldots \alpha_d$ - skew in each dimension. Unless otherwise stated, the following defaults were used for these parameters: dimensions d = 8, cardinalities $|D_i|$ = 256, 128, 64, 32, 16, 8, 4, 2, skew $\alpha = 0$ in all dimensions, and percentage of views selected k = 100%.

Throughout these experiments, as we increased the number of processors we observed two countervailing trends. Increasing processors, while holding total data size constant, leads to less data per processor and therefore better relative speedup because each processor can fit more of its data in memory, thereby reducing disk related overheads. On the other hand, using standard GigE switches and a standard MPI implementation, increasing the number of processors reduces the speed of communication, even when total data size communicated is held constant, and therefore tends to reduce relative speedup. The slight super linear effects observed in some of these experiments, for example at 16 processors in Figure 5, result when the benefits of fitting data in memory outweigh the penalties associated with higher communication overheads.

4.1 The Effects of Optimized Data Partitioning

Figure 5 shows for full cube construction the parallel wall clock time observed for data sets of $N = 8$ million records, with and without optimized data partitioning, as a function of the number of processors used. Figure 6 shows the corresponding relative speedup. Note that optimized data partitioning leads to a significant improvement in speedup. Without it, relative speedup hovers around 50% of optimal, while linear (i.e. optimal) relative speedup is achieved with optimized data partitioning.

Figure 7 shows for full cube construction the parallel wall clock time in seconds on a $p = 32$ node cluster as a function of the data size N = 16M, 32M, 48M, 64M, 128M, and 256M records. We observe that, with optimized partitioning, when we double the input size of the cube being generated at most twice the time is required. This holds true even for extremely large cubes where the input consists of 256 million rows of data (9.2 Gigabytes) and the output consists of a data cube consisting of 2^d views containing a total of $\approx$ 7 billion rows (200 Gigabytes), despite the fact that we are not scaling network bandwidth, in large part because of the improved data balance.

4.2 Relative Speedup

Speedup is one of key metrics for evaluation of parallel database systems [8] as it indicates the degree to which adding processors decreases the running time. The relative speedup for p processors is defined as $S_p = \frac{t_1}{t_p}$, where t_1 is the running time of the parallel program using one processor, all communication overhead having been removed from the program, and t_p is the running time using p processors. Figure 8 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 Figure 9 the corresponding relative speedup.

As is typically the case, relative speedup improves as we increase the size of the input and consequential the total amount of work to be performed. With N = 8,000,000 records, optimal linear relative speedup can be observed all the way up to 32 processors, while with only N = 1,000,000 records speedup drops off beyond 4 processors. In general, linear speedup is observed when there is at least N/p = 250,000 records per processor.

4.3 Partial Cubes

In many applications, users do not require all of the 2^d views contained in a full data cube but rather only a selected subset. The challenge for a partial cube generation method is to efficiently construct the set of selected views, maintaining relative efficiency even as the number of views (and therefore total work) is decreased.

Figure 10 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 Figure 11 the corresponding relative speedup. Note that near optimal speedup is achieved for a range of different percentages of selected views up to 16 processors. Beyond that there is a reduction in speedup for smaller set of selected views, in large part because there is simply not enough work to keep all of the processors busy.

4.4 Scaleup

Scaleup is another key metric for evaluation of parallel database systems [8]. It indicates whether a constant running time can be maintained as the workload is increased by adding a proportional numbers of processors and disks.

Figure 12 shows for full cube construction the parallel wall clock time observed as a function of the number of processors used when $N/p = 0.125M, 0.25M, 0.5M, 1M$ records per processor. Overall, we observe good scaleup. Initially, when we double the number of processors and double the size of the input, we observe a better than linear scaleup for all curves in Figure 12. This is due to the fact that we are holding the cardinalities of attributes constant as we increase the data size and therefore the relative density of the data cube is increasing which is beneficial for top-down generation methods [22, 2]. This increase in relative density leads to more agglomeration and therefore a smaller output data size per processor, as illustrated in Figure 13. However, this effect is offset by the fact that the network bandwidth is not being scaled as we increase the total input size N. As we increase the data size per processor more data has to be moved across the network as illustrated in Figure 13. When the total input size N is equal to, or greater than, 8M records network congestion on our switch begins to degrade the scaleup performance. However this effect can be observed to flatten out after N reaches 16M records.

4.5 Sizeup

Sizeup is similar to scaleup but fixes the number of processors. It indicates whether a proportional running time can be maintained as the workload is increased. The sizeup for x units of workload is defined as $U_x = \frac{t_x}{t_1}$, where t_1 is the running time of one unit workload and t_x is the running time of x unit workload. An ideal U_x is x, which implies that x units of workload costs x times more time than one unit of workload, so the curve of an ideal sizeup is a linear line. Figure 14 shows for full cube construction the sizeup observed for data sets of between 1 and 8 million records using between 1 and 32 processors. We observe that the sizeup curves are all approximately linear. The actual slope of the curves is determined by the ratio of the parallel overhead for fixed p when N = 1,000,000.

Figure 15 shows for cube construction the relative sizeup observed for data sets of between 1 and 256 million records on $p = 32$ processors. Even with these large record counts we observe good sizeup performance.

4.6 Data Skew

Data sets with skewed distributions can pose an interesting challenge to parallel data cube generation methods. As skew increases, data reduction tends also to increase, particularly in top-down generation methods [22, 2]. 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 [28] distribution in each dimension with $\alpha = 0$ (no skew) to $\alpha = 2$ (high skew).

Figure 16 shows for cube construction the parallel wall clock time observed as a function of the skew for $\alpha = 0, 1, 2$, and Figure 17 the corresponding relative speedup. We observe that, in general, as skew is increased parallel time decreases due to data reduction and decreased local computation. Our data partitioning optimization appears to handle gracefully the resulting data imbalance by shifting data appropriately. However, if this data reduction is very large, as for $\alpha = 2$, it reduces the opportunities for speedup as there is simply much less work to be parallelized.

4.7 Cardinality of Dimensions

The cardinality of the dimensions in a data set can affect the performance of our algorithm. As cardinalities increase so does the sparsity of the data set and this may adversely effect parallel time especially given that top-down methods [22, 2] are designed primarily for dense data cubes. Curves A, B and C of Figure 18 clearly illustrate this effect. The sparser data sets require significantly more time, although, as can be seen in Figure 19, this has a positive effect on the relative speedup achieved.

5 Conclusion

In this paper, we show how optimizing the data partitioning for parallel ROLAP data cube construction methods on shared-nothing multiprocessors can provide a *significant* performance improvement. Our optimized data partitioning method adapts to both, the current data set and the performance parameters of the machine. In comparison with previous approaches, our new method has a significantly better *scalability* with respect to the number of processors. Optimal speedup for as many as 32 processors was not observed for previous parallel methods [19, 12, 4]. In addition, because of its shared nothing approach, our new method does also significantly improve the scalability with respect to the I/O bandwidth (number of parallel disks) which is of great importance for handling large data sets.

References

[1] T3 Project Technical Overview. White paper, Microsoft, EMC, and Unisys, 2001.

[2] S. Agarwal, R. Agarwal, P.M. Deshpande, A. Gupta, J.F. Naughton, R. Ramakrishnan, and S. Srawagi. On the computation of multi-dimensional aggregates. In *Proc. 22nd VLDB Conf.*, pages 506–521, 1996.

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

[4] Y. Chen, F.Dehne, T.Eavis, and A.Rau-Chaplin. Parallel rolap data cube construction on shared-nothing multiprocessors. In *Proc. International Parallel and Distributed Processing Symposium (IPDPS 2003)*, page 70.2. IEEE Computer Society, 2003.

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

[6] F. Dehne, T. Eavis, and A. Rau-Chaplin. Computing partial data cubes. In *Proc. HICSS-37, January, 2004*, available online at http://www.cs.dal.ca/˜arc/publications/2-30/paper.pdf.

[7] 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.

[8] David DeWitt and Jim Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.

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

[10] S. Goil and A. Choudhary. High performance OLAP and data mining on parallel computers. *Journal of Data Mining and Knowledge Discovery*, 1(4):391–417, 1997.

[11] 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.

[12] S. Goil and A. N. Choudhary. High performance multidimensional analysis of large datasets. In *International Workshop on Data Warehousing and OLAP*, pages 34–39, 1998.

[13] 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*, 1(1):29–53, 1997.

[14] 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)*, pages 250–255, Portland, Oregon, 1996.

[15] V. Harinarayan, A. Rajaraman, and J.D. Ullman. Implementing data cubes efficiently. *ACM SIGMOD Record*, 25(2):205–216, 1996.

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

[17] 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.

[18] Seigo Muto and Masaru Kitsuregawa. A dynamic load balancing strategy for parallel datacube computation. In *ACM Second International Workshop on Data Warehousing and OLAP (DOLAP 1999)*, pages 67–72, 1999.

[19] Seigo Muto and Masaru Kitsuregawa. A dynamic load balancing strategy for parallel datacube computation. In *Proceedings of the second ACM international workshop on Data warehousing and OLAP*, pages 67–72. ACM Press, 1999.

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

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

[22] S. Sarawagi, R. Agrawal, and A. Gupta. On computing the data cube. Technical report rj10026, IBM Almaden Research Center, San Jose, CA, 1996.

[23] 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.

[24] J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.

[25] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2-3):110–147, 1994.

[26] J.X. Yu and H. Lu. Multi-cube computation. In *Proc. 7th International Symposium on Database Systems for Advanced Applications*, pages 126–133, Hong Kong, 2001.

[27] 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.

[28] G.K. Zipf. *Human Behavior and The Principle of Least Effort*. Addison-Wesley, 1949.

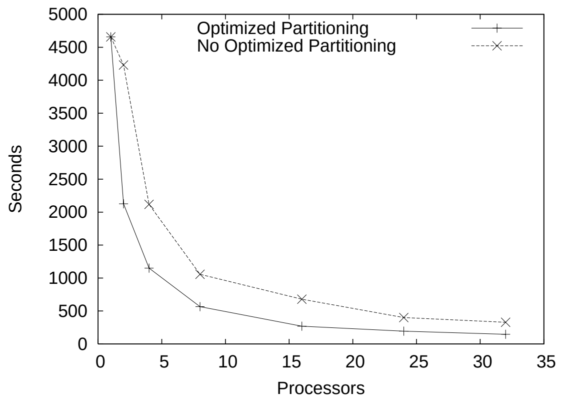


Figure 5: Parallel wall clock time in seconds as a function of the number of processors with and without optimized data partitioning.

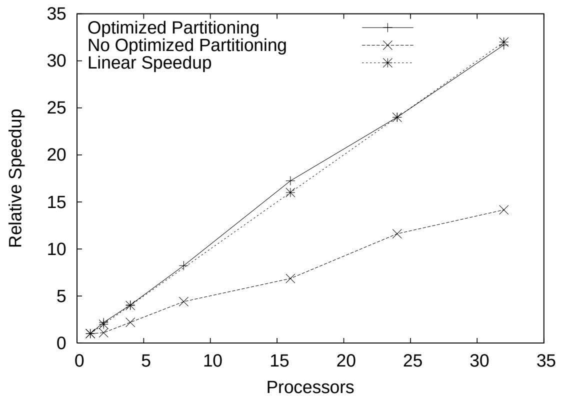


Figure 6: Relative speedup corresponding to Figure 5.

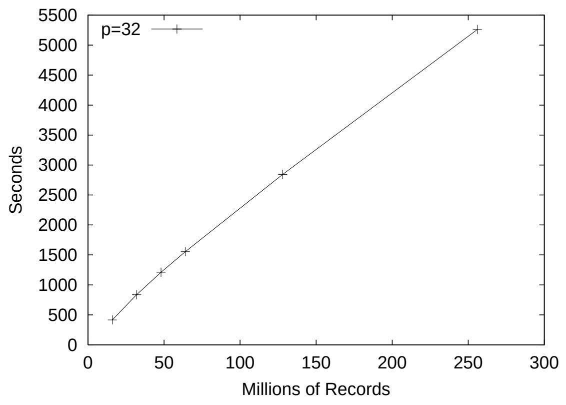


Figure 7: Parallel wall clock time in seconds as a function of the data size N = 16M, 32M, 48M, 64M, 128M, and 256M records.

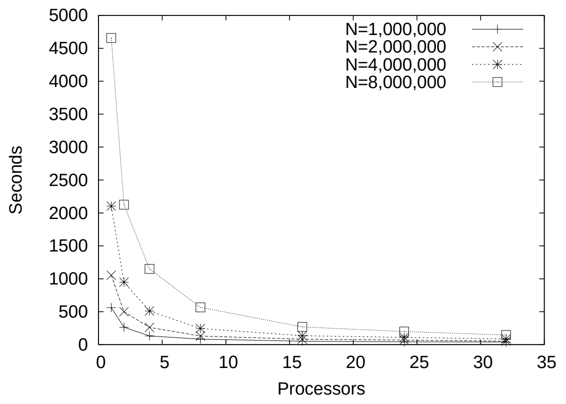


Figure 8: Parallel wall clock time in seconds as a function of the number of processors for data of size N = 1M, 2M, 4M and 8M records.

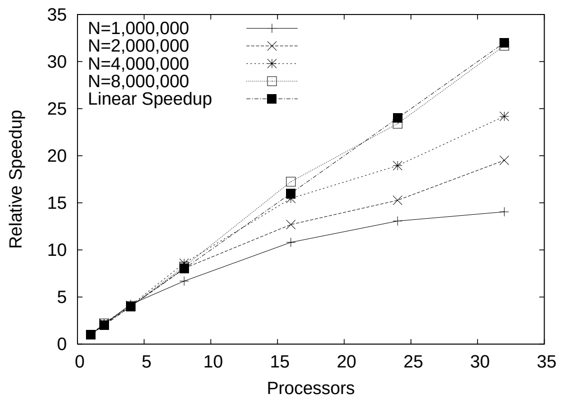


Figure 9: Relative speedup corresponding to Figure 8.

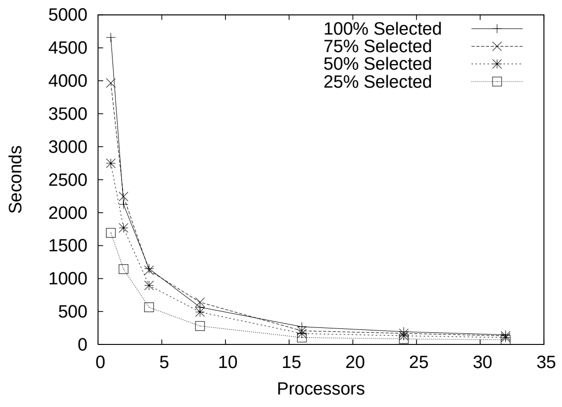


Figure 10: Parallel wall clock time in seconds as a function of the number of processors for partial cubes with percentage of selected views $k =$ 100%, 75%, 50%, and 25%.

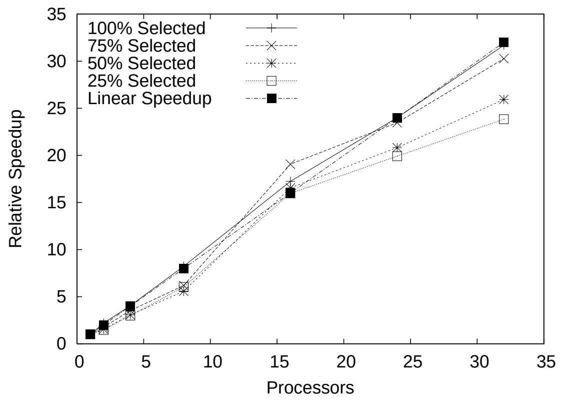


Figure 11: Relative speedup corresponding to Figure 8.

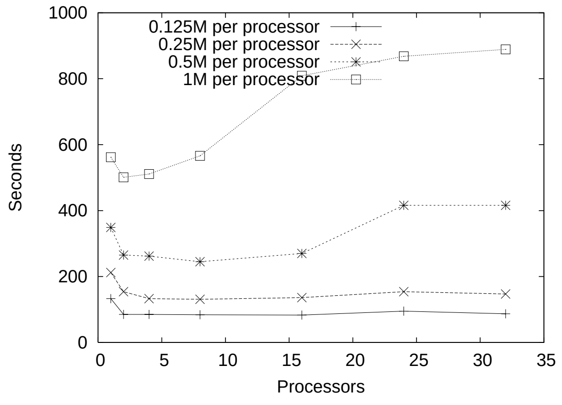


Figure 12: Scaleup for data size of N/p = 1M, 0.5M, 0.25M and 0.125M records per processor.

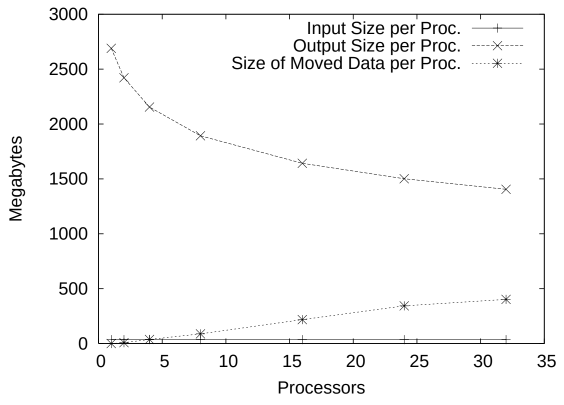


Figure 13: Output sizes per processor for input of N/p = 1M records per processor.

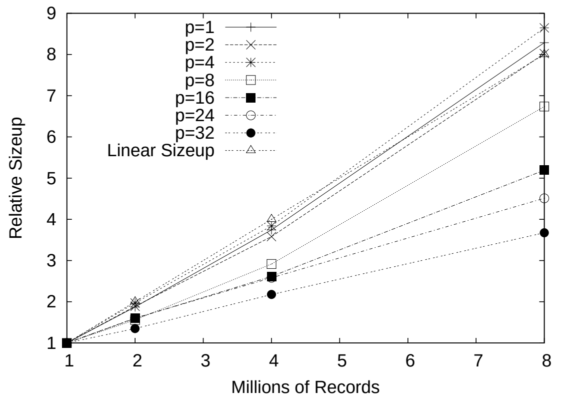


Figure 14: Relative sizeup for data sizes $N = 1M$ to $8M$ records on $p = 1$ to 32 processors.

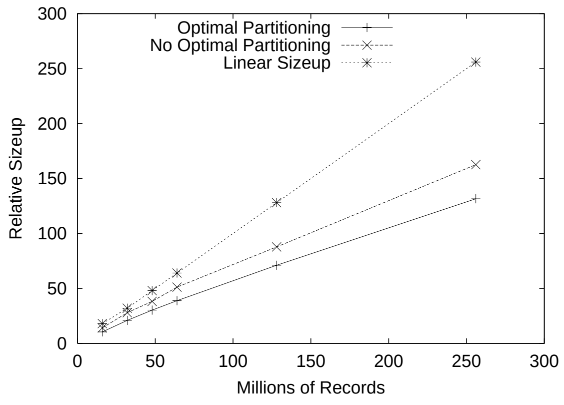


Figure 15: Relative sizeup for $p = 32$ and input data of size N = 16M, 32M, 48M, 64M, 128M, and 256M records. Corresponds to Figure 7.

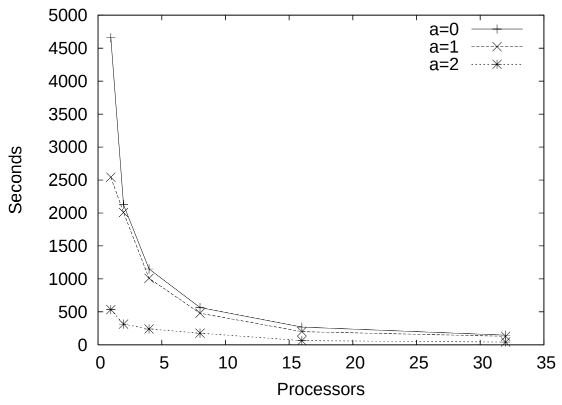


Figure 16: Parallel wall clock time in seconds as a function of the skew for $\alpha = 0, 1, 2$.

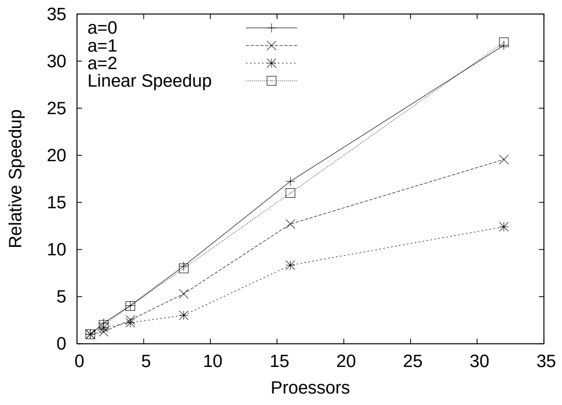


Figure 17: Relative speedup corresponding to Figure 16.

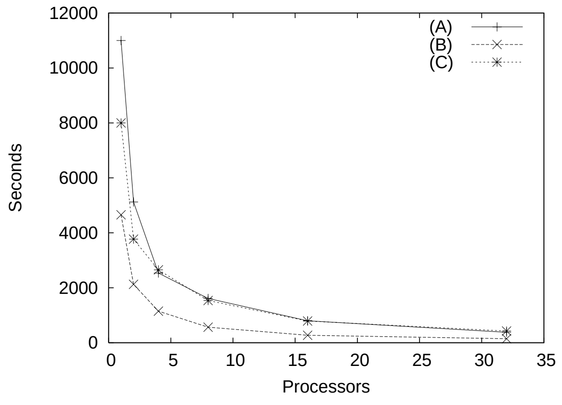


Figure 18: Parallel wall clock time in seconds as a function of the number of processors for data sets with different cardinality mixes (A)$|D_i|$ = 256, 256, 256, 256, 256, 256, 256, 256. (B)$|D_i|$ = 256, 128, 64, 32, 16, 8, 4, 2. (C)$|D_i|$ = 32, 32, 32, 32, 32, 32, 32, 32.

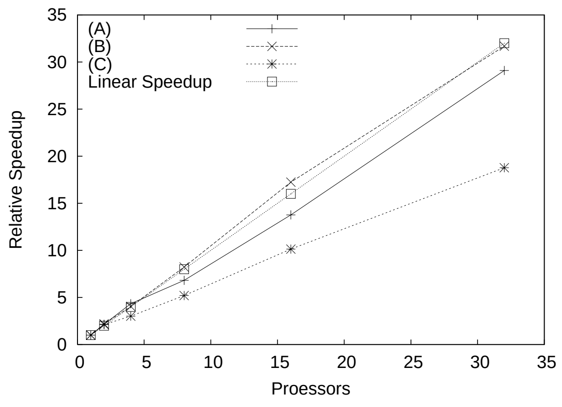


Figure 19: Relative speedup corresponding to Figure 18.