On the Computation of Multidimensional Aggregates

Sameet Agarwal   Rakesh Agrawal   Prasad M. Deshpande   Ashish Gupta
Jeffrey F. Naughton   Raghu Ramakrishnan   Sunita Sarawagi

Abstract

At the heart of all OLAP or multidimensional data analysis applications is the ability to simultaneously aggregate across many sets of dimensions. Computing multidimensional aggregates is a performance bottleneck for these applications. This paper presents fast algorithms for computing a collection of group-bys. We focus on a special case of the aggregation problem — computation of the **CUBE** operator. The **CUBE** operator requires computing group-bys on all possible combinations of a list of attributes, and is equivalent to the union of a number of standard group-by operations. We show how the structure of **CUBE** computation can be viewed in terms of a hierarchy of group-by operations. Our algorithms extend sort-based and hash-based grouping methods with several optimizations, like combining common operations across multiple group-bys, caching, and using pre-computed group-bys for computing other group-bys. Empirical evaluation shows that the resulting algorithms give much better performance compared to straightforward methods.

This paper combines work done concurrently on computing the data cube by two different teams as reported in [SAG96] and [DANR96].

1 Introduction

The group-by operator in SQL is typically used to compute aggregates on a set of attributes. For business data analysis, it is often necessary to aggregate data across many dimensions (attributes) [FINK, WELD95]. For example, in a retail application, one might have a table `Transactions` with attributes `Product(P)`, `Date(D)`, `Customer(C)` and `Sales(S)`. An analyst could then query the data for finding:

- `sum of sales by P, C`:
  For each product, give a breakdown on how much of it was sold to each customer.
- `sum of sales by D, C`:
  For each date, give a breakdown of sales by customer.
- `sum of sales by P`:
  For each product, give total sales.

Speed is a primary goal in these class of applications called On-Line Analytical Processing (OLAP) applications [CODD93]. To make interactive analysis (response time in seconds) possible, OLAP databases often precompute aggregates at various levels of detail and on various combinations of attributes. Speed is critical for this precomputation as well, since the cost and speed of precomputation influences how frequently the aggregates are brought up-to-date.

1.1 What is a CUBE?

Recently, [GBLP96] introduced the **CUBE** operator for conveniently supporting multiple aggregates in OLAP databases. The **CUBE** operator is the n-dimensional generalization of the group-by operator. It computes group-bys corresponding to all possible combinations of a list of attributes. Returning to our retail example, the collection of aggregate queries can be conveniently expressed using the cube-operator as follows:

```
SELECT P, D, C, Sum(S)
FROM Transactions
CUBE-BY P, D, C
```

This query will result in the computation of $2^3 = 8$ group-bys: `PDC`, `PD`, `PC`, `DC`, `D`, `C`, `P` and `all`, where `all` denotes the empty group-by. The straightforward way to support the above query is to rewrite it as a collection of eight group-by queries and execute them

Proceedings of the 22nd VLDB Conference
Mumbai(Bombay), India, 1996

separately. There are several ways in which this simple solution can be improved.

In this paper, we present fast algorithms for computing the data cube. We assume that the aggregating functions are *distributive* [GBLP96], that is, they allow the input set to be partitioned into disjoint sets that can be aggregated separately and later combined. Examples of distributive functions include `max`, `min`, `count`, and `sum`. The proposed algorithms are also applicable to the *algebraic* aggregate functions [GBLP96], such as `average`, that can be expressed in terms of other distributive functions (`sum` and `count` in the case of `average`). However, as pointed out in [GBLP96], there are some aggregate functions (*holistic* functions of [GBLP96]) e.g., `median`, that cannot be computed in parts and combined.

Related Work

Methods of computing *single* group-bys have been well-studied (see [Gra93] for a survey), but little work has been done on optimizing a collection of related aggregates. [GBLP96] gives some rules of thumb to be used in an efficient implementation of the cube operator. These include the smallest parent optimization and partitioning of data by attribute values, which we adopt in our algorithms. However, the primary focus in [GBLP96] is on defining the semantics of the cube operator [GBLP96]. There are reports of on-going research related to the data cube in directions complementary to ours: [HRU96, GHRU96] presents algorithms for deciding what group-bys to pre-compute and index; [SR96] and [JS96] discuss methods for indexing pre-computed summaries to allow efficient querying.

Aggregate pre-computation is quite common in statistical databases [Sho82]. Research in this area has considered various aspects of the problem starting from developing a model for aggregate computation [CM89], indexing pre-computed aggregates [STL89] and incrementally maintaining them [Mic92]. However, to the best of our knowledge, there is no published work in the statistical database literature on methods for optimizing the computation of related aggregates.

This paper is in two parts and combines work done concurrently on computing the data cube. Part I presents the methods proposed by [SAG96], whereas the methods proposed by [DANR96] are described in Part II. Section 10.3 presents a brief comparison of the two approaches.

Part I[1]

[1] This part presents work done by Sunita Sarawagi, Rakesh Agrawal and Ashish Gupta at IBM Almaden Research Center, San Jose.

2 Optimizations Possible

There are two basic methods for computing a group-by: (1) the sort-based method and (2) the hash-based method [Gra93]. We will adapt these methods to compute multiple group-bys by incorporating the following optimizations:

1. **Smallest-parent:** This optimization, first proposed in [GBLP96], aims at computing a group-by from the smallest previously computed group-by. In general, each group-by can be computed from a number of other group-bys. Figure 1 shows a four attribute cube ($ABCD$) and the options for computing a group-by from a group-by having one more attribute called its *parent*. For instance, AB can be computed from ABC, ABD or $ABCD$. ABC or ABD are clearly better choices for computing AB. In addition, even between ABC and ABD, there can often be big difference in size making it critical to consider size in selecting a parent for computing AB.
2. **Cache-results:** This optimization aims at caching (in memory) the results of a group-by from which other group-bys are computed to reduce disk I/O. For instance, for the cube in Figure 1, having computed ABC, we compute AB from it while ABC is still in memory.
3. **Amortize-scans:** This optimization aims at amortizing disk reads by computing as many group-bys as possible, together in memory. For instance, if the group-by $ABCD$ is stored on disk, we could reduce disk read costs if all of ABC, ACD, ABD and BCD were computed in one scan of $ABCD$.
4. **Share-sorts:** This optimization is specific to the sort-based algorithms and aims at sharing sorting cost across multiple group-bys.
5. **Share-partitions:** This optimization is specific to the hash-based algorithms. When the hash-table is too large to fit in memory, data is partitioned and aggregation is done for each partition that fits in memory. We can save on partitioning cost by sharing this cost across multiple group-bys.

For OLAP databases, the size of the data to be aggregated is usually much larger than the available main memory. Under such constraints, the above optimizations are often contradictory. For computing B, for instance, the first optimization will favor BC over AB if BC is smaller but the second optimization will favor AB if AB is in memory and BC is on disk.

Contributions In this part of the paper, we will present two algorithms for computing the data cube: the sort-based algorithm *PipeSort* (Section 3) and the hash-based algorithm *PipeHash* (Section 4) that includes the optimizations listed above. We have ex-

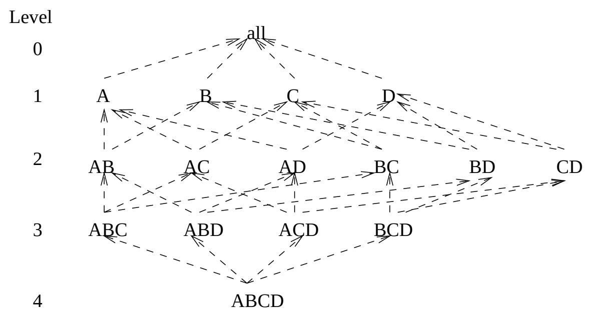


Figure 1: A search lattice for the cube operator

tended these algorithms to two important real-life OLAP cases. The first deals with the useful case of computing a specified subset of the group-bys in a cube. For this case, we identify a reduction of the problem to the *minimum steiner tree* [GJ79] problem. This enables us to find plans that consider computation of intermediate group-bys that are not part of the specified subset but can lead to smaller total cost. The second extension handles the case in which attributes have hierarchies defined on them. Due to space limitation, we have not included these extensions in this paper, and we refer the reader to [SAG96] for them.

3 Sort-based methods

In this section, we present the sort-based algorithm that incorporates the optimizations listed earlier. We include the optimization **share-sort** by using data sorted in a particular order to compute all group-bys that are prefixes of that order. For instance, if we sort the raw data on attribute order $ABCD$, then we can compute group-bys $ABCD$, ABC, AB and A without additional sorts. However, this decision could conflict with the optimization **smallest-parent**. For instance, the smallest parent of AB might be BDA although by generating AB from ABC we are able to share the sorting cost. It is necessary, therefore, to do global planning to decide what group-by is computed from what and the attribute order in which it is computed. We propose an algorithm called *PipeSort* that combines the optimizations share-sorts and smallest-parent to get the minimum total cost.

The PipeSort algorithm also includes the optimizations **cache-results** and **amortize-scans** to reduce disk scan cost by executing multiple group-bys in a *pipelined* fashion. For instance, consider the previous example of using data sorted in the order $ABCD$ to compute prefixes $ABCD$, ABC, AB and A. Instead of computing each of these group-bys separately, we can compute them in a pipelined fashion as follows. Having sorted the raw data in the attribute order $ABCD$, we scan the sorted data to compute group-by $ABCD$. Every time a tuple of $ABCD$ is computed, it is propagated up the pipeline to compute ABC; every time a tuple of ABC is computed, it is propagated up to compute AB, and so on. Thus, each pipeline is a list of group-bys all of which are computed in a single scan of the sort input stream. During the course of execution of a pipeline we need to keep only one tuple per group-by in the pipeline in memory.

Algorithm PipeSort

Assume that for each group-by we have an estimate of the number of distinct values. A number of statistical procedures (e.g., [HNSS95]) can be used for this purpose. The input to the algorithm is the *search lattice* defined as follows.

Search Lattice A search lattice [HRU96] for a data cube is a graph where a vertex represents a group-by of the cube. A directed edge connects group-by i to group-by j whenever j can be generated from i and j has exactly one attribute less than i (i is called the parent of j). Thus, the out-degree of any node with k attributes is k. Figure 1 is an example of a search lattice. Level k of the search lattice denotes all group-bys that contain exactly k attributes. The keyword `all` is used to denote the empty group-by (Level 0). Each edge in the search lattice e_{ij} is labeled with two costs. The first cost $S(e_{ij})$ is the cost of computing j from i when i is not already sorted. The second cost $A(e_{ij})$ is the cost of computing j from i when i is already sorted.

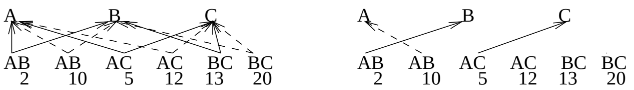
The output, O of the algorithm is a subgraph of the search lattice where each group-by is connected to a single parent group-by from which it will be computed and is associated with an attribute order in which it will be sorted. If the attribute order of a group-by j is a prefix of the order of its parent i, then j can be computed from i without sorting i and in O, edge e_{ij} is marked A and incurs cost $A(e_{ij})$. Otherwise, i has to be sorted to compute j and in O, e_{ij} is marked S and incurs cost S_{ij}. Clearly, for any output O, there can be at most one out-edge marked A from any group-by i, since there can be only one prefix of i in the adjacent level. However, there can be multiple out-edges marked S from i. The objective of the algorithm is to find an output O that has minimum sum of edge costs.

Algorithm The algorithm proceeds level-by-level, starting from level $k = 0$ to level $k = N - 1$, where N is the total number of attributes. For each level k, it finds the best way of computing level k from level $k+1$ by reducing the problem to a weighted bipartite matching problem[2] [PS82] as follows.

[2] The weighted bipartite matching problems is defined as follows: We are given a graph with two disjoint sets of vertices V_1 and V_2 and a set of edges E that connect vertices in set V_1 to vertices in set V_2. Each edge is associated with a fixed weight. The weighted matching problem selects the maximum weight subset of edges from E such that in the selected subgraph each vertex in V_1 is connected to at most one vertex in V_2 and

We first transform level $k+1$ of the original search lattice by making k additional copies of each group-by in that level. Thus each level $k+1$ group-by has $k+1$ vertices which is the same as the number of children or out-edges of that group-by. Each replicated vertex is connected to the same set of vertices as the original vertex in the search lattice. The cost on an edge e_{ij} from the original vertex i to a level k vertex j is set to $A(e_{ij})$ whereas all replicated vertices of i have edge cost set to $S(e_{ij})$. We then find the minimum [3] cost matching in the bipartite graph induced by this transformed graph. In the matching so found, each vertex h in level k will be matched to some vertex g in level $k+1$. If h is connected to g by an $A()$ edge, then h determines the attribute order in which g will be sorted during its computation. On the other hand, if h is connected by an $S()$ edge, g will be re-sorted for computing h.

For illustration, we show how level 1 group-bys are generated from level 2 group-bys for a three attribute search lattice. As shown in Figure 2(a), we first make one additional copy of each level 2 group-by. Solid edges represent the $A()$ edges whereas dashed edges indicate the $S()$ edges. The number underneath each vertex is the cost of all out-edges from this vertex. In the minimum cost matching (Figure 2(b)), A is connected to AB with an $S()$ edge and B by an $A()$ edge. Thus at level 2, group-by AB will be computed in the attribute order BA so that B is generated from it without sorting and A is generated by resorting BA. Similarly, since C is connected to AC by an $A()$ edge, AC will be generated in the attribute order CA. Since, BC is not matched to any level-1 group-by, BC can be computed in any order.



(a) Transformed search lattice (b) Minimum cost matching

Figure 2: Computing level 1 group-bys from level 2 group-bys in a 3 attribute cube

We use the algorithm in [PS82] for finding the minimum cost matching in a bipartite graph[4]. The complexity of this algorithm is $O(((k+1)M_{k+1})^3)$, where M_{k+1} is the number of group-bys in level $k+1$.

PipeSort:
(Input: search lattice with the A() and S() edges costs)
For level $k = 0$ to $N-1$
/* find how to generate level k from level $k+1$ */
Generate-Plan($k+1 \rightarrow k$);
For each group-by g in level $k+1$
Fix the sort order of g as the order of the group-by connected to g by an A() edge;

Generate-Plan($k+1 \rightarrow k$)
Make k additional copies of each level $k+1$ vertex;
Connect each copy vertex to the same set of level k vertices as the original vertex;
Assign cost $A(e_{ij})$ to edge e_{ij} from the original vertex and $S(e_{ij})$ to edge from the copy vertex;
Find the minimum cost matching on the transformed levels.

Example: We illustrate the PipeSort algorithm for the four attribute lattice of Figure 1. For simplicity, assume that for a given group-by g the costs A() and S() are the same for all group-bys computable from g. The pair of numbers underneath each group-by in Figure 3 denote the A() and S() costs. Solid edges denote A() edges and dashed edges denote S() edges. For these costs, the graph in Figure 3(a) shows the final minimum cost plan output by the PipeSort algorithm. Note that the plan in Figure 3(a) is optimal in terms of the *total cost* although the *total number of sorts* is suboptimal. For most real-life datasets there could be a big difference in the sizes of group-bys on a level. Hence, optimizing for the number of sorts alone could lead to poor plans.

In Figure 3(b) we show the pipelines that are executed. Sorts are indicated by ellipses. We would first sort data in the order $CBAD$. In one scan of the sorted data, $CBAD$, CBA, CB, C and `all` would be computed in a pipelined fashion. Then group-by $ABCD$ would be sorted into the new order $BADC$ and thereafter BAD, BA and B would be computed in a pipelined fashion.
We can make the following claims about algorithm PipeSort.

Claim 3.1 Generate-plan() finds the best plan to get level k from level $k+1$.

PROOF. Follows by construction assuming a cost function where the cost of sorting a group-by does not depend on the order in which the group-by is already sorted.

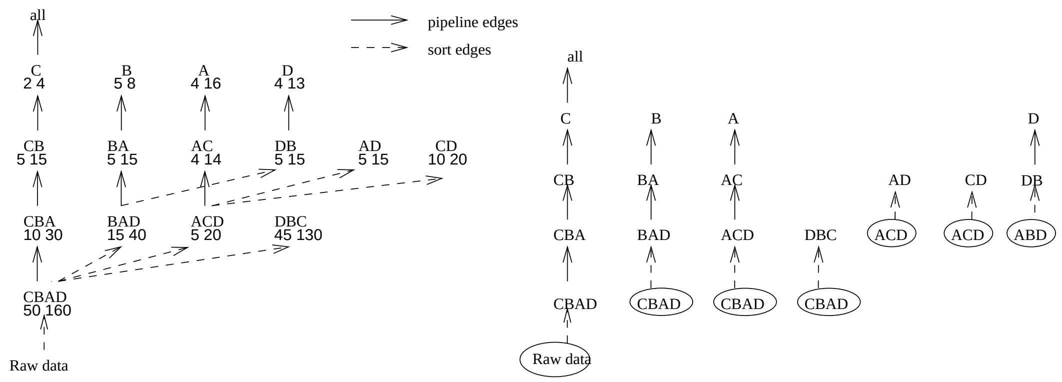
Claim 3.2 Generate-plan($k+1 \rightarrow k$) does not prevent Generate-plan($k+2 \rightarrow k+1$) from finding the best plan.

PROOF. After we have fixed the way to generate level k from level $k+1$ the only constraint we have on level $k+1$ is the order in which the group-bys should be generated. This ordering does not affect the minimum matching solution for generating level $k+1$ from $k+2$.

vice-versa.

[3] Note we can covert a minimum weight matching to a maximum weight matching defined earlier by replacing each edge weight w by $max(w) - w$ where $max(w)$ is the maximum edge cost.

[4] The code for the matching algorithm is available from `ftp-request@theory.stanford.edu`



(a) The minimum cost sort plan

(b) The pipelines that are executed

Figure 3: Sort-based method for computing a four attribute cube

After finding the best solution for generating level $k+1$ from level $k+2$, we can always change the order in which each group-by should be generated (as dictated by level k solution) without affecting the minimum cost.

Note that PipeSort computes each group-by from a group-by occurring only in the immediately preceding level. Although the level-by-level approach is not provably optimal, we have not been able to find any case where generating a group-by from a group-by not in the preceding level leads to a better solution. Our experiments reported in Section 5 also show that our solution is very close to empirically estimated lower bounds for several datasets.

Further Enhancements Our implementation of PipeSort includes the usual optimizations of aggregating and removing duplicates while sorting, instead of doing aggregation as a different phase after sorting[Gra93]. Often we can reduce the sorting cost by taking advantage of the partial sorting order. For instance, in Figure 3 for sorting ACD in the attribute order AD, we can get a sorted run of D for each distinct value of AC and for each distinct A we can merge these runs of D. Also, after the PipeSort algorithm has fixed the order in which each group-by is generated we can modify the sort-edges in the output search lattice to take advantage of the partial sorting orders whenever it is advantageous to do so.

4 Hash-based methods

We now discuss how we extend the hash-based method for computing a data cube. For hash-based methods, the new challenge is careful memory allocations of multiple hash-tables for incorporating optimizations **cache-results** and **amortize-scans**. For instance, if the hash tables for AB and AC fit in memory then the two group-bys could be computed in one scan of ABC. After AB is computed one could compute A and B while AB is still in memory and thus avoid the disk scan of AB. If memory were not a limitation, we could include all optimizations stated in Section 2 as follows.

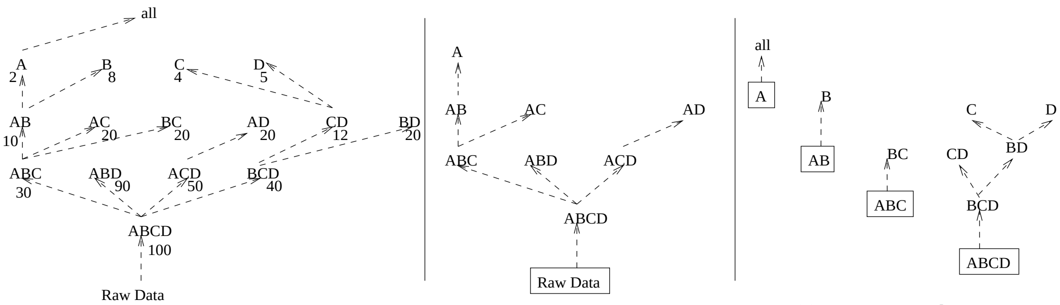
For $k = N$ to 0
  For each $k+1$ attribute group-by, g
    Compute in one scan of g all k attribute group-by
      for which g is the smallest parent;
    Save g to disk and destroy hash table of g;

However, the data to be aggregated is usually too large for the hash-tables to fit in memory. The standard way to deal with limited memory when constructing hash tables is to partition the data on one or more attributes. When data is partitioned on some attribute, say A, then all group-bys that contain A can be computed by independently grouping on each partition — the results across multiple partitions need not be combined. We can share the cost of data partitioning across all group-bys that contain the partitioning attribute, leading to the optimization **share-partitions**. We present below the *PipeHash* algorithm that incorporates this optimization and also includes the optimizations **cache-results**, **amortize-scans** and **smallest-parent**.

Algorithm PipeHash

The input to the algorithm is the search lattice described in the previous section. The PipeHash algorithm first chooses for each group-by, the parent group-by with the smallest estimated total size. The outcome is a minimum spanning tree (MST) where each vertex is a group-by and an edge from group-by a to b shows that a is the smallest parent of b. In Figure 4 we show the MST for a four attribute search lattice (the size of each group-by is indicated below the group-by).

In general, the available memory will not be sufficient to compute all the group-bys in the MST together, hence the next step is to decide what group-bys to compute together, when to allocate and deallocate memory for different hash-tables, and what attribute



(a) Minimum spanning tree (b) First subtree: partitioned on A (c) Remaining subtrees

Figure 4: PipeHash on a four attribute group-by

to choose for partitioning data. We conjecture this problem to be NP-complete because solving this problem optimally requires us to solve the following subproblem optimally: Divide the MST into smaller subtrees each of which can be computed in one scan of the group-by at the root of the MST such that the cost of scanning (from disk) the root group-by is minimized. This problem is similar to well-known NP-complete partitioning problems [GJ79]. Hence, we resort to using a heuristic solution. Later (in Section 5) we show that our solution is very close to empirically estimated lower bounds for several datasets.

Optimizations cache-results and amortize-scans are favored by choosing as large a subtree of the MST as possible so that we can use the method above to compute together the group-bys in the subtree. However, when data needs to be partitioned based on some attribute, the partitioning attribute limits the subtree to only include group-bys containing the partitioning attribute. We therefore, choose a partitioning attribute that allows the choice of the largest subtree as shown in the pseudo-code of the PipeHash algorithm below.

PipeHash:

Input: search lattice with group-by estimated sizes
Initialize worklist with MST of the search lattice;
While worklist is not empty
  Pick any tree T from the worklist;
  T' = Select-subtree of T to be executed next;
  Compute-subtree T';

Select-subtree

If memory required by $T <$ available, return T
Else, let S be the attributes of root(T)
  (We will pick $s \subset S$ for partitioning root(T).
  For any s we get a subtree T_s of T also rooted at
    T including all group-bys that contain s.)
  Let P_s = maximum number of partitions of root(T)
  possible if partitioned on $s \subset S$;
  We choose $s \subset S$ such that
    memory required by $T_s/P_s <$ memory available,
    and T_s is the largest over all subsets of S;
  Remove T_s from T;
  This leaves $T - T_s$, a forest of smaller trees; add
  this to the worklist;
return T_s;

Compute-subtree

M = memory available;
numParts = memory required by T'*fudge_factor/M;
Partition root of T' into numParts;
For each partition of root(T')
  For each node, n in T'
  (scanned in a breadth first manner)
    Compute all children of n in one scan;
    If n is cached, save it to disk and
    release memory occupied by its hash-table;

Example: Figure 4 illustrates the PipeHash algorithm for the four attribute search lattice of Figure 1. The boxed group-bys represent the root of the subtrees. Figure 4(a) shows the minimum spanning tree. Assume there is not enough memory to compute the whole tree in one pass and we need to partition the data. Figure 4(b) shows the first subtree T_A selected when A is chosen as the partitioning attribute. After removing T_A from the MST, we are left with four subtrees as shown in Figure 4(c). None of the group-bys in these subtrees include A. For computing T_A, we first partition the raw data on A. For each partition we compute first the group-by $ABCD$; then scan $ABCD$ (while it is still in memory) to compute ABC, ABD and ACD together; save $ABCD$ and ABD to disk; compute AD from ACD; save ACD and AD to disk; scan ABC to compute AB and AC; save ABC and AC to disk; scan AB to compute A and save AB and A to disk. After T_A is computed, we compute each of the remaining four subtrees in the worklist.

Note that PipeHash incorporates the optimization share-partitions by computing from the same partition all group-bys that contain the partitioning attribute. Also, when computing a subtree we maintain all hash-tables of group-bys in the subtree (except the root) in memory until all its children are created. Also, for each group-by we compute its children in one scan of the group-by. Thus PipeHash also incorporate the op-

Dataset	# grouping attributes	# tuples (in millions)	size (in MB)
Dataset-A	3	5.5	110
Dataset-B	4	7.5	121
Dataset-C	5	9	180
Dataset-D	5	3	121
Dataset-E	6	0.7	18

Table 1: Description of the datasets

timizations amortize-scans and cache-results. [5]

PipeHash is biased towards optimizing for the smallest-parent. For each group-by, we first fix the smallest parent and then incorporate the other optimizations. For instance, in Figure 4(c), we could have computed BC from BCD instead of its smallest parent ABC and thus saved the extra scan on ABC. However, in practice, saving on sequential disk scans is less important than reducing the CPU cost of aggregation by choosing the smallest parent.

5 Experimental evaluation

In this section, we present the performance of our cube algorithms on several real-life datasets and analyze the behavior of these algorithms on tunable synthetic datasets. These experiments were performed on a RS/6000 250 workstation running AIX 3.2.5. The workstation had a total physical memory of 256 MB. We used a buffer of size 32 MB. The datasets were stored as flat files on a local 2GB SCSI 3.5" drive with sequential throughput of about 1.5 MB/second.

Datasets Table 1 lists the five real-life datasets used in the experiments. These datasets were derived from sales transactions of various department stores and mail order companies. A brief description is given next. The datasets differ in the number of transactions, the number of attributes, and the number of distinct values for each attribute. For each attribute, the number within brackets denotes the number of its distinct values.

- **Dataset-A:** This data is about supermarket purchases. Each transaction has three attributes: store id(73), date(16) and item identifier(48510). In addition, two attributes cost and amount are used as aggregation columns.
- **Dataset-B:** This data is from a mail order company. A sales transaction here consists of four attributes: the customer identifier(213972), the order date(2589), the product identifier(15836), and the catalog used for ordering(214).

[5] Refer [SAG96] for a discussion of how we handle the problems of data skew and incorrect size estimates in allocating hash-tables

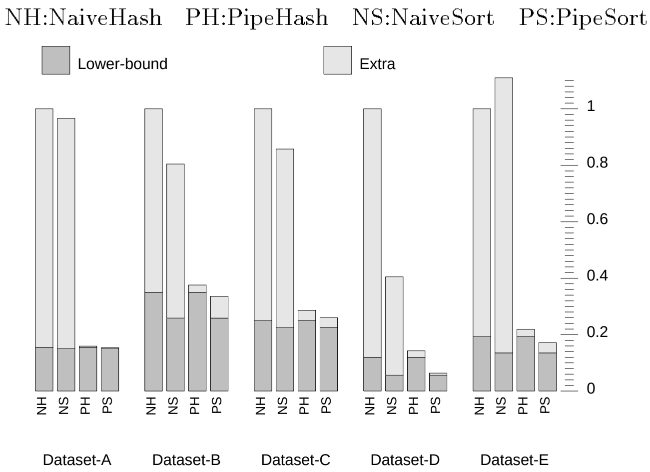
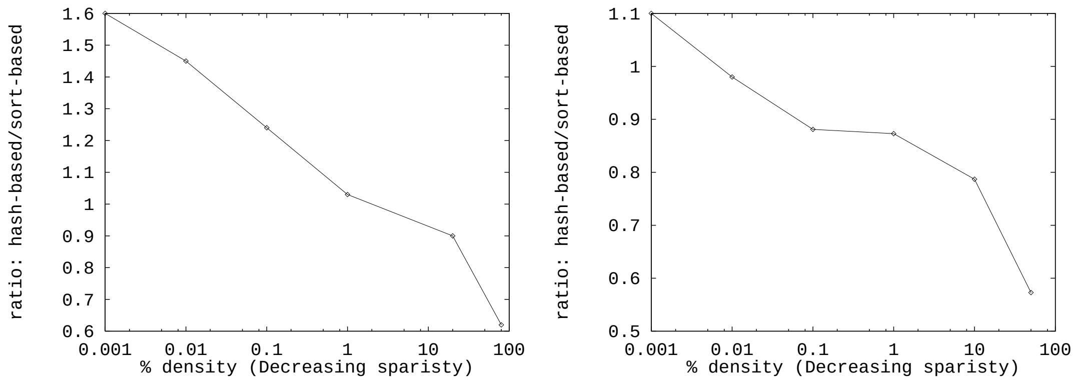


Figure 5: Performance of the cube computation algorithms on the five real life datasets. The y-axis denotes the total time normalized by the time taken by the NaiveHash algorithm for each dataset.

- **Dataset-C:** This is data about grocery purchases of customers from a supermarket. Each transaction has five attributes: the date of purchase(1092), the shopper type(195), the store code(415), the state in which the store is located(46) and the product group of the item purchased(118).
- **Dataset-D:** This is data from a department store. Each transaction has five attributes: the store identifier(17), the date of purchase(15), the UPC of the product(85161), the department number(44) and the SKU number(63895).
- **Dataset-E:** This data is also from a department store. Each transaction has total of six attributes: the store number(4), the date of purchase(15), the item number(26412), the business center(6), the merchandising group(22496) and a sequence number(255). A seventh attribute: the quantity of purchase was used as the aggregating column.

Algorithms compared For providing a basis of evaluation, we choose the straightforward method of computing each group-by in a cube as a separate group-by resulting in algorithms *NaiveHash* and *NaiveSort* depending on whether group-bys are computed using hash-based or sort-based methods. We further compare our algorithms against easy but possibly unachievable lower-bounds.

For the hash-based method the lower bound is obtained by summing up the following operations: Compute the bottom-most (level-N) group-by by hashing raw-data stored on disk; include the data partitioning cost if any. Compute all other group-bys by hashing the smallest parent assumed to be in memory; ignore data partitioning costs. Save all computed group-bys to disk.



(a) Ratio 1:2:4:20:300 (b) Ratio 1:1:1:1:1

Figure 6: Effect of sparseness on relative performance of PipeSort and PipeHash for a 5 attribute synthetic dataset.

For the sort-based method the lower bound is obtained by summing up the following operations: Compute the bottom-most (level-N) group-by by sorting the raw-data stored on disk. Compute all other group-bys from the smallest parent assumed to be in memory and sorted in the order of the group-by to be computed. Save all computed group-bys.

Performance results Figure 5 shows the performance of the proposed PipeHash and PipeSort relative to the corresponding naive algorithms and estimated lower bounds. The total execution time is normalized by the time taken by the NaiveHash algorithm for each dataset to enable presentation on the same scale. In [SAG96] we discuss the methods we used for estimating the size of each group-by and the hashing function used with NaiveHash and PipeHash. We can make the following observations.

- Our algorithms are two to eight times faster than the naive methods.
- The performance of PipeHash is very close to our calculated lower bound for hash-based algorithms. The maximum difference in performance is 8%.
- PipeSort is also close to the calculated lower bound for sort-based method in most cases. The maximum gap between their performance is 22%.
- For most of the datasets, PipeHash is inferior to the PipeSort algorithms. We suspected this to be an artifact of these datasets. To further investigate the difference between them, therefore, we did a series of experiments on a synthetically generated dataset described next.

5.1 Comparing PipeSort and PipeHash

For the datasets in Table 1, the sort-based method performs better than the hash-based method. For Dataset-D, PipeSort is almost a factor of two better than PipeHash. Based on results in [GLS94], we had expected the hash-based method to be comparable or better than the sort-based method. Careful scrutiny of the performance data revealed that this deviation is because after some parent group-by is sorted we compute more than one group-by from it whereas for the hash-based method we build a different hash table for each group-by. Even though we share the partitioning cost for the hash-based method, the partitioning cost is not a dominant fraction of the total cost unlike sorting.

We conjectured that the hash-based method can perform better than the sort-based method when each group-by results in a considerable reduction in the number of tuples. This is because the cost of hashing at higher levels of aggregations can become a negligible fraction of the total cost when the number of tuples reduces rapidly. To validate our conjecture that the performance difference between the hash-based method and sort-based method is mainly due to the rate of decrease in the number of tuples as we aggregate along more and more attributes, we took a series of measurements on synthetic datasets described below.

Synthetic datasets Each dataset is characterized by four parameters:

1. Number of tuples, T.
2. Number of grouping attributes, N.
3. Ratio amongst the number of distinct values of each attribute $d_1 : d_2 : \ldots : d_N$.
4. A parameter, p, denoting the degree of sparsity of the data. It is defined as the ratio of T to the total number of possible attribute value combinations. Thus, if D_i denotes the number of distinct values of attribute i, then p is defined as $T/(D_1 \times D_2 \ldots D_N)$. Clearly, higher the degree of sparsity (lower value of p), lower the reduction in the number of tuples after aggregation.

Given these four parameters, the dataset is generated as follows. We first determine the total number of values D_i along each dimension i as:

$$D_i = \left(\frac{T}{p}\right)^{\frac{1}{N}} \frac{d_i}{(d_1 \times d_2 \times \ldots \times d_N)^{\frac{1}{N}}}$$

Then, for each of the T tuples, we choose a value for each attribute i randomly between 1 and D_i.

Results We show the results for two sets of synthetic datasets with T is 5 million, N is 5. For dataset in Figure 6(a) the ratio between the number of distinct values of each attribute is 1:2:4:20:300 (large variance in number of distinct values). We vary the sparsity by changing p. The X-axis denotes decreasing levels of sparsity and the Y-axis denotes the ratio between the total running time of algorithms PipeHash and PipeSort. We notice that as the data becomes less and less sparse the hash-based method performs better than the sort-based method. We repeated the same set of measurements for datasets with a different ratio, 1:1:1:1:1 (Figure 6(b)). We notice the same trend for datasets with very different characteristics, empirically confirming that sparsity indeed is a predictor of the relative performance of the PipeHash and PipeSort algorithms.

Part II[6]

6 Contributions of this Part

We present a class of sorting-based methods for computing the CUBE that try to minimize the number of disk accesses by overlapping the computation of the various cuboids. They make use of partially matching sort orders to reduce the number of sorting steps required. Our experiments with an implementation of these methods show that they perform well even with limited amounts of memory. In particular, they always perform substantially better than the *Independent* and *Parent* method of computing the CUBE by a sequence of group-by statements, which is currently the only option in commercial relational database systems.

7 Options for Computing the CUBE

Let R be a relation with $k+1$ attributes $\{A_1, A_2, \ldots, A_{k+1}\}$. Consider the computation of a CUBE on k attributes $X = \{A_1, A_2, \ldots, A_k\}$ of relation R with aggregate function $F(\cdot)$ applied on A_{k+1}. A *cuboid* on j attributes $S = \{A_{i_1}, A_{i_2}, \ldots, A_{i_j}\}$ is defined as a group-by on the attributes $A_{i_1}, A_{i_2}, \ldots, A_{i_j}$ using the aggregate function F. This cuboid can be represented as a $k+1$ attribute relation by using the special value `ALL` for the remaining $k-j$ attributes [GBLP96]. The CUBE on attribute set X is the union of cuboids on all subsets of attributes of X. The cuboid (or group-by) on all attributes in X is called the *base cuboid*.

To compute the CUBE we need to compute all the cuboids that together form the CUBE. The base cuboid has to be computed from the original relation. The other cuboids can be computed from the base cuboid due to the distributive nature of the aggregation. For example, in a retail application relation with attributes *(Product, Year, Customer, Sales)*, *sum of sales by (product, customer)* can be obtained by using *sum of sales by (product, year, customer)*. There are different ways of scheduling the computations of the cuboids:

Multiple Independent Group-By Queries (Independent Method)

A straightforward approach (which we call *Independent*) is to independently compute each cuboid from the base cuboid, using any of the standard group-by techniques. Thus the base cuboid is read and processed for each cuboid to be computed, leading to poor performance.

Hierarchy of Group-By Queries (Parent Method)

Consider the computation of different cuboids for the CUBE on attributes $\{A, B, C, D\}$. The cuboid $\{A, C\}$ can be computed from the cuboid $\{A, B, C\}$ or the cuboid $\{A, C, D\}$, since the aggregation function is distributive. In general, a cuboid on attribute set X (called cuboid X) can be computed from a cuboid on attribute set Y iff $X \subset Y$. One optimization is to choose Y to be as small as possible to reduce cost of computation. We use the heuristic of computing a cuboid with $k-1$ attributes from a cuboid with k attributes, since cuboid size is likely to increase with additional attributes. For example, it is better to compute *sum of sales by (product)* using *sum of sales by (product, customer)* rather than *sum of sales by (product, year, customer)*.

We can view this hierarchy as a DAG where the nodes are cuboids and there is an edge from a k attribute cuboid to a $k-1$ attribute cuboid iff the $k-1$ attribute set is a subset of the k attribute set. The DAG captures the "consider-computing-from" relationship between the cuboids. The DAG for the CUBE on $\{A, B, C, D\}$ is shown in Figure 7.

In the *Parent* method each cuboid is computed from one of its parents in the DAG. This is better than the *Independent* method since the parent is likely to be much smaller than the base cuboid, which is the largest of all the cuboids.

[6] This part presents work done by Prasad M. Deshpande, Sameet Agarwal, Jeffrey F. Naughton and Raghu Ramakrishnan; {*pmd, sameet, naughton, raghu*}*@cs.wisc.edu*, University of Wisconsin-Madison.
It was supported by a grant from IBM under the University Partnership Programand NSF grant IRI-9157357

Overlap Method

This is a further extension of the idea behind the *Parent* method. While the *Independent* and *Parent* methods are currently in use by Relational OLAP tools, the *Overlap* method cannot be used directly by a standard SQL database system and to our knowledge it has not appeared in the literature to date. As in the *Parent* method, the *Overlap* method computes each cuboid from one of its parents in the cuboid tree. It tries to do better than *Parent* by overlapping the computation of different cuboids and using partially matching sort orders. This can significantly reduce the number of I/Os required. The details of this scheme are explained in Section 8.

7.1 Computing the Group-bys using Sorting

In relational query processing, there are various methods for computing a group-by, such as sorting or hashing [EPST79, Gra93, SN95]. These methods can be used to compute one cuboid from another. We concentrate on sorting based methods in this paper, though we believe that hashing could also be used similarly. Computing a CUBE requires computation of a number of cuboids (group-bys). Sorting combined with *Overlap* seems to be a good option due to the following observations which help in reducing the number of sorting steps.

- Cuboids can be computed from a sorted cuboid in sorted order.
- An existing sort order on a cuboid can be used while computing other cuboids from it. For example, consider a cuboid $X = \{A, B, D\}$ to be computed from $Y = \{A, B, C, D\}$. Let Y be sorted in ABCD order which is not the same as ABD order needed to compute X. But Y need not be resorted to compute X. The existing order on Y can be used. The exact details are explained in Section 8.

8 The Overlap Method

The method we propose for CUBE computation is a sort-based overlap method. Computations of different cuboids are overlapped and all cuboids are computed in sorted order. In this paper we give only a short description of our method. More details can be found in [DANR96]. We first define some terms which will be used frequently.

Sorted Runs :

Consider a cuboid on j attributes $\{A_1, A_2, \ldots, A_j\}$. We use $(A_1, A_2, \ldots, A_j)$ to denote the cuboid sorted on the attributes A_1, A_2, …, A_j in that order. Consider the cuboid $S = (A_1, A_2, \ldots, A_{l-1}, A_{l+1}, \ldots, A_j)$ computed using $B = (A_1, A_2, \ldots, A_j)$. A *sorted run* R of S in B is defined as follows: $R = \Pi_{A_1, A_2, \ldots, A_{l-1}, A_{l+1}, \ldots, A_j}(Q)$ where Q is a **maximal** sequence of tuples τ of B such that for each tuple in Q, the first l columns have the same value. Informally a sorted run of S in B is a maximal run of tuples in B whose ordering is consistent with their ordering in the sort order associated with S.

For example, consider $B = [(a, 1, 2), (a, 1, 3), (a, 2, 2), (b, 1, 3), (b, 3, 2), (c, 3, 1)]$. Let S be the cuboid on the first and third attribute. i.e., $S = [(a, 2), (a, 3), (b, 3), (b, 2), (c, 1)]$. The sorted runs for S are $[(a, 2), (a, 3)]$, $[(a, 2)]$, $[(b, 3)]$, $[(b, 2)]$ and $[(c, 1)]$.

Partitions :

B and S have a common prefix of $A_1, A_2, \ldots, A_{l-1}$. A *partition* of the cuboid S in B is a union of sorted runs such that the first $l - 1$ columns (the common prefix) of all the tuples of the sorted runs have the same value. In the above example, the partitions for S in B will be $[(a, 2), (a, 3)]$, $[(b, 2), (b, 3)]$ and $[(c, 1)]$.

This definition implies that all tuples of one partition are either less or greater than all tuples of any other partition. Tuples from different partitions will not merge for aggregation. Thus partition becomes a unit of computation and each partition can be computed independently of the others.

8.1 Overview of the Overlap Method

The overlap method is a muti-pass method. In each pass, a set of cuboids is selected for computing under memory constraints. These cuboids are computed in a overlapped manner. The tuples generated for a cuboid are used to compute its descendents in the DAG. This pipelining reduces the number of scans needed. The process is repeated until all cuboids get computed.

The algorithm begins by sorting the base cuboid. All other cuboids can be directly computed in sorted order without any further sorting. Instead of re-sorting for each cuboid, the existing sorted runs are merged to create the cuboid. This reduces the number of comparisons as well. Suppose the base cuboid for the CUBE on $\{A, B, C, D\}$ is sorted in the order (A, B, C, D). This decides the sort order in which the other cuboids get computed. The sort orders for the other cuboids of $\{A, B, C, D\}$ are shown in the Figure 7. A few heuristics for choosing this sort order are mentioned in [DANR96].

Computation of each cuboid requires some amount of memory. If there is enough to memory to hold all the cuboids, then the entire CUBE can be computed in one scan of the input relation. But often, this is not the case. The available memory may be insufficient for large CUBEs. Thus, to get the maximum overlap across computations of different cuboids, we could try to reduce the amount of memory needed to compute a particular cuboid. Since partition can be a unit of computation, while computing a cuboid from another

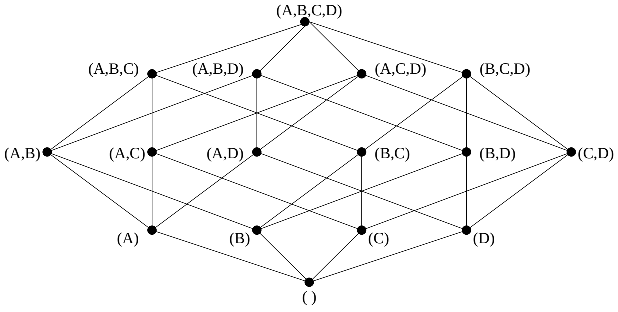


Figure 7: Sort orders enforced on the cuboids

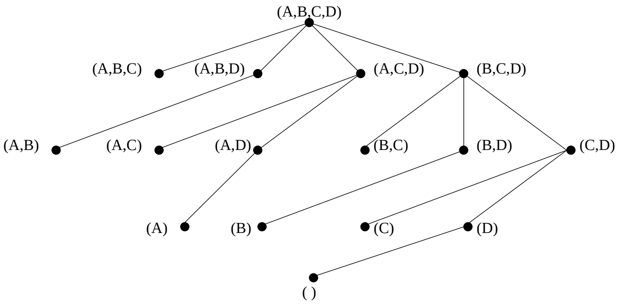


Figure 8: Cuboid Tree obtained from the cuboid DAG

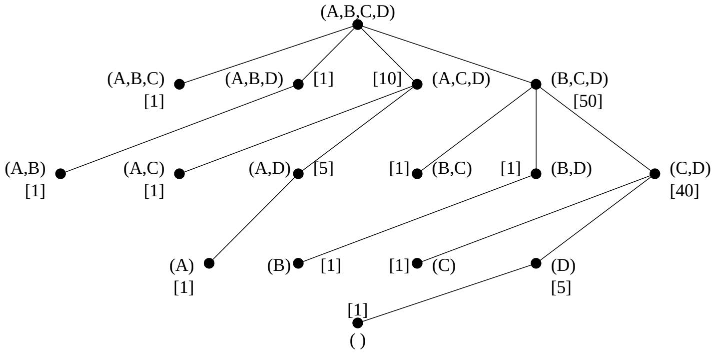
sorted cuboid we just need memory sufficient to hold a partition of the cuboid. As soon as a partition is completed, the tuples can be pipelined into the computation of descendant cuboids, or written out to disk; the same memory can then be used to start computation of the next partition. This is a significant reduction since for most cuboids the partition size is much less than the size of the cuboid. For example, while computing (A, B, C) and (A, B, D) from (A, B, C, D) the partition size for (A, B, C) is 1 tuple (since (A, B, C) sort order matches (A, B, C, D) sort order) whereas the partition size for (A, B, D) is bounded by the number of distinct values of D. So for computing these we just need space sufficient to hold a partition. Thus computation of many cuboids can be overlapped in the available memory effectively reducing the number of scans.

8.2 Details

8.2.1 Choosing a Parent to Compute a Cuboid

Each cuboid in the cuboid DAG has more than one parent from which it could be computed. We need to choose one of these parents thus converting the DAG to a rooted tree. The root of the tree is the base cuboid and each cuboid's parent is the cuboid to be used for computing it. For example, one possible tree for computing the DAG in Figure 7 is as shown in Figure 8.

There are many possible trees. The goal in choosing a tree is to minimize the size of the partitions of a cuboid so that minimum memory is needed for its computation. For example, it is better to compute (A, C) from (A, C, D) rather than (A, B, C). This is because (A, C, D) sort order matches the (A, C) sort order and the partition size is 1. This is generalized to the following heuristic: Consider the cuboid $S = (A_{i_1}, A_{i_2}, \ldots, A_{i_j})$, where the base cuboid is $(A_1, A_2, \ldots, A_k)$. S can be computed from any cuboid with one additional attribute, say A_l. Our heuristic is to choose the cuboid with the largest value of l to compute S. Maximizing the size of the common prefix minimizes the partition size. The tree in Figure 8 is obtained by using this heuristic. Note that among the children of a particular node, the partition sizes increase from left to right. For example, partition size for computing (A, B, C) from (A, B, C, D) is 1 whereas the partition size for (B, C, D) is the maximum (equal to size of the cuboid (B, C, D) itself).



[...] indicates estimated partition size in number of pages

Figure 9: Estimates of Partition Sizes

8.2.2 Choosing a Set of Cuboids for Overlapped Computation

The next step is to choose a set of cuboids that can be computed concurrently within the memory constraints. To compute a cuboid in memory, we need memory equal to the size of its partition. We assume that we have estimates of sizes of the cuboids. The partition sizes can be estimated from these using uniform distribution assumption [DANR96]. If this much memory can be allocated, the cuboid will be marked to be in *Partition* state. For some other cuboids it may be possible to allocate one page of memory. These cuboids will be *SortRun* state. The allocated page can be used to write out sorted runs for this cuboid on disk. This will save a scan of the parent when the cuboid has to be computed. These sorted runs are merged in further passes to complete the computation.

Given any subtree of a cuboid tree and the size of memory M, we need to mark the cuboids to be computed and allocate memory for their computation. When a cuboid is in *Partition* state, its tuples can be pipelined for computing the descendent cuboids in the same pass. This is not true for *SortRun* state. Thus we have the following constraints:

C1: A cuboid can be considered for computation if either its parent is the root of the subtree (this means either the parent cuboid itself or sorted-runs for the parent cuboid have been materialized on the disk), or the parent has been marked as being in the *Partition* state.

C2: The total memory allocated to all the cuboids should not be more than the available memory M.

There are a large number of options for selecting which cuboids to compute and in what state. The cost of computation depends critically on the choices made. When a cuboid is marked in *SortRun* state there is an additional cost of writing out the sorted runs and reading them to merge and compute the cuboids in the subtree rooted at that node. We have shown that finding an overall optimal allocation scheme for our cuboid tree is NP-hard [DANR96] . So, instead of trying to find the optimal allocation we do the allocation by using the heuristic of traversing the tree in a breadth first (BF) search order:

- Cuboids to the left have smaller partition sizes, and require less memory. So consider these before considering cuboids to the right.
- Cuboids at a higher level tend to be bigger. Thus, these should be given higher priority for allocation than cuboids at a lower level in the tree.

Because of the constraints there may be some subtrees that remain uncomputed. These are considered in subsequent passes, using the same algorithm to allocate memory and mark cuboids. Thus, when the algorithm terminates, all cuboids have been computed.

8.2.3 Computing a Cuboid From its Parent

This section describes the actual method of computation for the chosen cuboids. Every cuboid (say S) other than the base cuboid is computed from its parent in the cuboid tree (say B). If a cuboid has been marked in *Partition* state it means that we have sufficient memory to fit the largest partition of S in memory. We can compute the entire cuboid S in one pass over B and also pipeline the tuples generated for further computation if necessary. However, if the cuboid is marked to be in *SortRun* state, we can write out sorted runs of S in this pass. Writing out the sorted runs requires just one page of memory. The algorithm for computing a cuboid is specified below :

Output: The sorted cuboid S.

foreach tuple τ of B **do**
  if (state == *Partition*) **then**
    process_partition(τ)
  else
    process_sorted_run(τ)
  endif
  end_of_cuboid()
endfor

The process_partition() procedure is as follows:

- If the input tuple starts a new partition, output the current partition at the end of the cuboid, start a new one and make it current.
- If the input tuple matches with an existing tuple in the partition then recompute the aggregate of the existing tuple using the old aggregate value and the input tuple.
- If the input tuple is not the same as any existing tuple then insert the input tuple into the current partition at the appropriate location to maintain the sorted order of the partition.

The process_sort_run() procedure is as follows:

- If the input tuple starts a new sorted run, flush all the pages of the current sorted run, start a new sorted run and make it current.
- If the input tuple matches with the last tuple in the sorted run then recompute the aggregate of the last tuple using the old aggregate value and the input tuple.
- If the input tuple does not match with the last tuple of the sorted run, append the tuple to the end of the existing run. If, there is no space in the allocated memory for the sorted run, we flush out the pages in the memory to the end of the current sorted run on disk. Continue the sorted run in memory with the input tuple.

The end_of_cuboid() processing writes the final partition or sorted run currently in memory to disk. For the case of the *Partition* state, the cuboids get computed completely in the first pass. For *SortRun*, we now have a set of sorted runs on disk. We compute such a cuboid by merging these runs, like the merge step of external sort, aggregating duplicates if necessary. This step is combined with the computation of cuboids that are descendants of that cuboid. The runs are merged and the result pipelined for further computation (of descendants). Note that computation of a cuboid in the *SortRun* state involves the additional cost of writing out and merging the runs. Further, the child cuboids cannot be computed during the run-creation phase, and must be computed during the subsequent merging phase, as noted above.

8.3 Example computation of a CUBE

Consider the CUBE to be computed on $\{A, B, C, D\}$. The tree of cuboids and the estimates of the partition sizes of the cuboids are shown in Figure 9. If the memory available is 25 pages, BF allocation will generate three subtrees, each of which is computed in one pass. These subtrees are shown in Figure 10. In the second

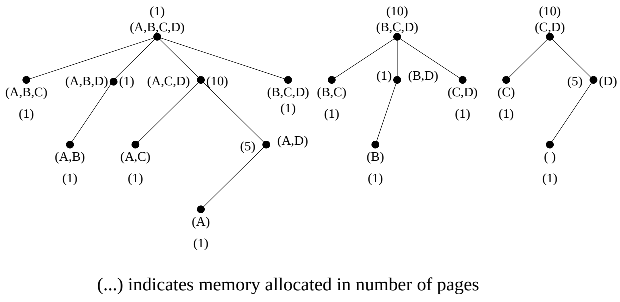


Figure 10: Steps of the algorithm

and third steps the cuboids (B, C, D) and (C, D) are allocated 10 pages as there are 9 sorted runs to merge.

Comparison with Independent and Parent method

The cost of writing out the computed cuboids is common to all the schemes. The only additional cost in this case was of writing the sorted runs of (B, C, D) and (C, D) and merging these sorted runs. The *Independent* scheme would have required 16 scans and sorts of the base cuboid (once for each cuboid to be computed) and the *Parent* scheme would require a number of scans and sorts of each non-leaf cuboid in the tree (one for each of its children). Thus our scheme incurs fewer I/Os and less computation compared to these two.

9 Implementation and Results

To test how well our algorithm performs, we implemented a stand-alone version of the algorithm and tested it for varying memory sizes and data distributions. All the experiments were done on a Sun SPARC 10 machine running SUN-OS or Solaris. The implementation uses the file system provided by the OS. All reads and writes to the files were in terms of blocks corresponding to the page size. Performance was measured in terms of I/Os by counting the number of page read and page write requests generated by the algorithm and is thus independent of the OS. A detailed performance study is described in [DANR96]. We mention only a few important experiments here.

Unless otherwise mentioned, the data for the input relation was generated randomly. The values for each attribute is independently chosen uniformly from a domain of values for that attribute. Each tuple has six attributes and the CUBE is computed on five attributes with the aggregation (computing the sum) on the sixth attribute. Each CUBE attribute has 40 distinct values. Each tuple is 24 bytes wide. The page size used was 1K.

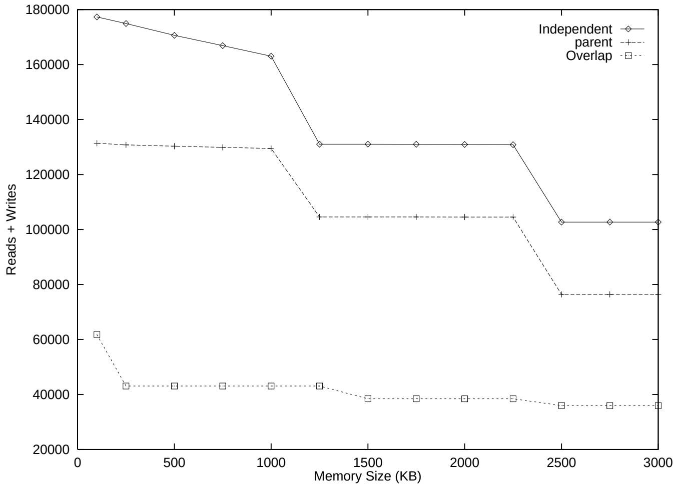


Figure 11: Uniform Data : Varying Memory : Input Size 2.4M, CUBE Size 27.1M

9.1 Comparison with Independent and Parent methods

To illustrate the gains of our algorithm over other methods, we compare the performance of our algorithm with the *Independent* and *Parent* methods described before. We varied different parameters like memory size, relation size, data distribution and the number of attributes on which the CUBE is computed.

9.1.1 Different data distributions

In order to run experiments that finished in a reasonable amount of time, for the bulk of our experiments the relation size was kept constant at $100,000$ tuples (2.4 MByte). While this is quite small, the important performance parameter in our algorithm is the ratio of the relation size and the memory size. To compensate for an artificially small input relation size, we used very small memory sizes, varying from a low of 100 pages (100 KByte) to a high of 3000 pages (3 MB). Section 9.1.2 shows that the performance characteristics of the algorithms we tested are unchanged if you scale the memory and data size to more realistic levels. For each of the methods, we plotted the sum of the number of reads and writes.

The graph in Figure 11 shows the performance of the three algorithms for uniform data. Figure 12 is for non-uniform data which is generated using zipf distribution for the attribute values. Values for A and B were chosen with a zipf factor of 2, C with a factor of 1, and D and E with a factor of 0 (uniform distribution)

The graphs in Figures 11 and 12 show that our method achieves a significant improvement over the *Independent* and *Parent* methods for both uniform and non-uniform data. There are some spikes in the graph in Figure 12. For example, the I/O performance at memory size 1500K is worse than that at 1250K for

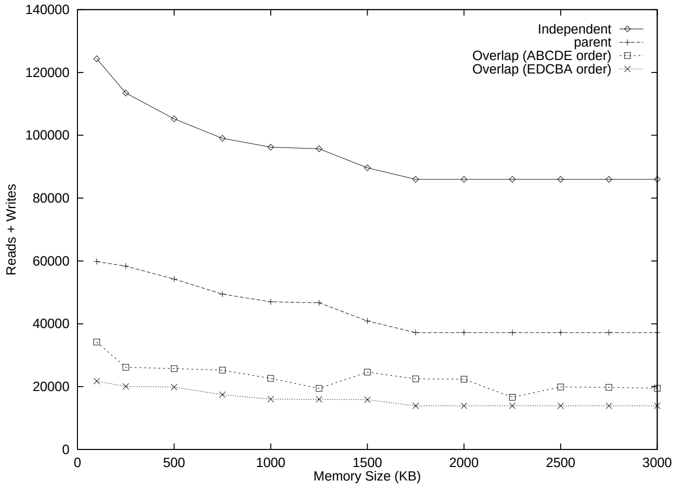


Figure 12: Non-uniform Data : Zipf Distribution : Input Size 2.4M, CUBE Size 10M

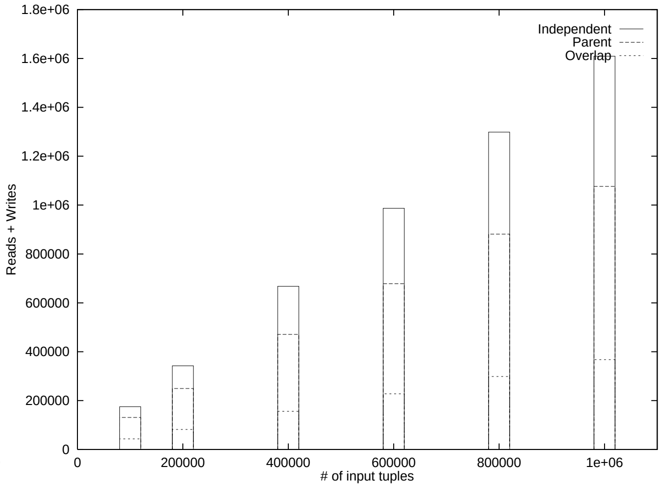


Figure 13: Scale up : I/Os

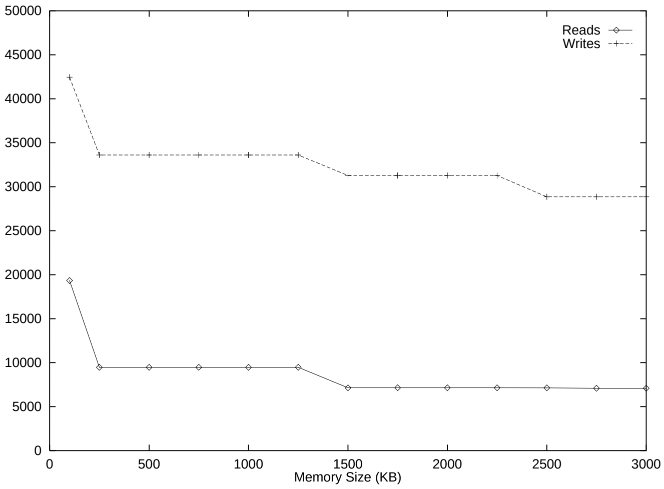


Figure 14: Varying memory : Relation : 2.4M; **CUBE** size : 27.1M

our algorithm. This only shows that the breadth-first heuristic that we are using for memory allocation is not always optimal.

The graphs also show that choosing a proper sort order is important. For non-uniform data, sort order 'EDCBA' is better than the order 'ABCDE'. This is due to different degrees of skewness in different attributes.

9.1.2 Scaleup Experiments

We performed some experiments to check how our method scales for larger input sizes with proportionately larger memory sizes. The relation size was varied from 100,000 (2.4M) to 1000,000 tuples (24 M). The memory used for each case was about 10% of the relation size. The graph in Figure 13 shows that the performance characteristics of the algorithms we consider are unchanged when the data sets are scaled to more realistic levels.

9.2 Relation between Memory and Input size for Overlap method

We performed some experiments to study how our method performs for different ratios of memory to the input size.

9.2.1 Varying Memory

Figure 14 plots the number of Reads and Writes for computing CUBE for a input size of 100,000 tuples (2.4MB). The memory is varied from 100K to 3MB. From the graphs in Figure 14, it is clear that the I/Os decrease with increasing memory since more and more cuboids are computed simultaneously, avoiding excess reading and writing of sorted runs. We observe that even for very low memory sizes, the number of writes is only slightly more than the size of CUBE and the number of reads is within two times the input relation size. This shows that we are getting near optimal performance with respect to number of I/Os.

9.2.2 Varying Relation size

In the other experiment, the memory was kept constant at 500 pages (500K). The input relation size was varied from 10000 to $100,000$ tuples. Each attribute has 20 distinct values. The graph is shown in the Figure 15. The X axis represents the size of the relation in bytes. On the Y axis, we plot the following ratios.

1. $\frac{Number\ of\ Writes}{Size\ of\ the\ \mathbf{CUBE}\ in\ Pages}$
2. $\frac{Number\ of\ Reads}{Size\ of\ the\ Input\ Relation\ in\ Pages}$

Any algorithm to compute the cube has to scan the input and write out the results. Hence these ratios give an idea of how close the algorithm is to ideal. Since the memory size is 500K, for relations of size up

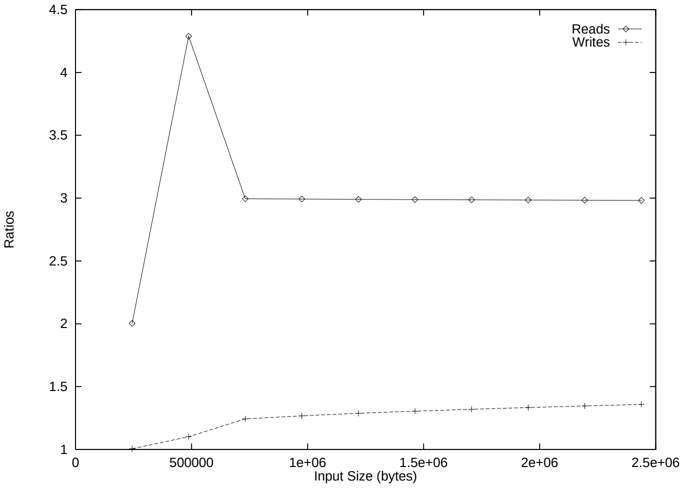


Figure 15: Varying relation sizes: Memory : 500K

to 500K, the performance is ideal. For bigger relations, the performance degrades slowly as the partitions no longer fit in memory and sorted runs have to be written out for many cuboids. The spikes show that the BF allocation may be non-optimal in some cases.

10 Conclusions and Summary

10.1 Summary of part I

We presented two algorithms for computing the data cube. Our algorithms extend the sort-based and hash-based methods for computing group-bys with five optimizations: *smallest-parent*, *cache-results*, *amortize-scans*, *share-sorts* and *share-partitions*. These optimizations are often conflicting. Our proposed algorithms combine them so as to reduce the total cost. The sort-based algorithm, called *PipeSort*, develops a plan by reducing the problem to a minimum weight matching problem on a bipartite graph. The hash-based algorithm, called *PipeHash*, develops a plan by first creating the minimum spanning tree showing what group-by should be generated from what and then choosing a partitioning that takes into account memory availability.

Measurements on five real-life OLAP datasets yielded a factor of two to eight improvement with our algorithms over straightforward methods of computing each group-by separately. Although the PipeHash and PipeSort algorithms are not provably optimum, comparison with conservatively calculated lower bounds show that the PipeHash algorithm was within 8% and the PipeSort algorithm was within 22% of these lower bounds on several datasets. We further experimented with the PipeHash and PipeSort algorithms using a tunable synthetic dataset and observed that their relative performance depends on the sparsity of data values. PipeHash does better on low sparsity data whereas PipeSort does better on high sparsity data. Thus, we can choose between the PipeHash and PipeSort algorithms for a particular dataset based on estimated sparsity of the dataset.

We extended the cube algorithms to compute a specified subset of the 2^N group-bys instead of all of them. Our proposed extension considers intermediate group-bys that are not in the desired subset for generating the best plan. We also extended our algorithms for computing aggregations in the presence of hierarchies on attributes. These extensions are discussed in [SAG96].

10.2 Summary of Part II

In this part we have examined various schemes to implement the CUBE operator. Sorting-based methods exploit the existing ordering to reduce the number of sorts. Also, pipelining can be used to save on reads.

- We have presented one particular sorting based scheme called *Overlap*. This scheme overlaps the computation of different cuboids and minimizes the number of scans needed. It uses estimates about cuboid sizes to determine a “good” schedule for the computation of the cuboids if the estimates are fairly accurate.
- We implemented the *Overlap* method and compared it with two other schemes.From the performance results, it is clear that our algorithm is a definite improvement over the *Independent* and the *Parent* methods. The idea of partitions allows us to overlap the computation of many cuboids using minimum possible memory for each. By overlapping computations and making use of partially matching sort orders, our algorithms will perform much better than the *Independent* and *Parent* method, irrespective of what heuristic is used for allocation.
- The *Overlap* algorithm gives reasonably good performance even for very limited memory. Though these results are for relatively small relations, the memory used was also relatively small. Scaleup experiments show that similar results should hold for larger relations with more memory available. Very often we may not want to compute all the cuboids. This can be handled in our algorithm by deleting nodes which are not to be computed from the cuboid tree. Results show that the algorithm gives good performance even for this case.
- We have shown that the optimal allocation problem is NP-hard. We have therefore used a heuristic allocation (BF) in our algorithm. The results suggest that the heuristics yield performance close to that of optimal allocation in most cases.

10.3 Comparison of PipeSort and Overlap

The PipeSort method takes into account the size of a group-by while selecting a parent with the aim of reducing both scanning cost and sorting cost. It views this as a matching problem to choose the optimal parent and sort order for each group-by. It may thus use more than one sort order.

The Overlap method on the other hand uses a single sort order. This helps in setting up multiple pipelines (as against the single pipeline of the PipeSort method) to achieve more overlap using Partitions. While choosing a parent, it tries to get maximum match in their sort orders. However, unlike PipeSort, it does not consider the size of the group-bys.

We have not compared the performance of these two methods. As future work, we plan to study their relative merits, and consider how their best features can be combined.

References

[CM89] M.C. Chen and L.P. McNamee. The data model and access method of summary data management. *IEEE Transactions on Knowledge and Data Engineering*, 1(4):519–29, 1989.

[GBLP96] Jim Gray, Adam Bosworth, Andrew Layman and Hamid Pirahesh. Data Cube: A Relational Operator Generalizing Group-By, Cross-Tab and Sub-Totals. *Proc. of the 12th Int. Conf. on Data Engineering*, pp 152–159, 1996.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability*, pages 45–76,65,96,208–209,247. W. H. Freeman, San Francisco, 1979.

[GLS94] G. Graefe, A. Linville, and L. D. Shapiro. Sort versus hash revisited. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):934–944, 1994.

[Gra93] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, Jun 1993.

[HNSS95] P.J. Haas, J.F. Naughton, S. Seshadri, and L. Stokes. Sampling-based estimation of the number of distinct values of an attribute. In *Proceedings of the Eighth International Conference on Very Large Databases (VLDB)*, pages 311–22, Zurich, Switzerland, September 1995.

[JS96] T. Johnson and D. Shasha. Hierarchically split cube forests for decision support: description and tuned design, 1996. Working Paper.

[PS82] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*, chapter 11, pages 247–254. Englewood Cliffs, N.J., Prentice Hall, 1982.

[FELL57] William Feller. *An Introduction to Probability Theory and Its Applications*, Vol. I, page 241. John Wiley & Sons, 1957.

[HRU96] Venky Harinarayan, Anand Rajaraman and Jeff Ullman. Implementing Data Cubes Efficiently. In *Proc. of the 1996 ACM-SIGMOD Conference*, 1996.

[GHRU96] Himanshu Gupta, Venky Harinarayan, Anand Rajaraman and Jeffrey D. Ullman. Index Selection for OLAP *Working Paper*, 1996.

[EPST79] Robert Epsteinr. Techniques for Processing of Aggregates in Relational Database Systems. *Memo UCB/ERL M79/8*, E.R.L., College of Engg., U. of California, Berkeley, Feb 1979.

[SN95] Ambuj Shatdal and Jeffrey F. Naughton. Adaptive Parallel Aggregation Algorithms. *Proc. of the 1995 ACM-SIGMOD Conference, San Jose, CA*, May 1995.

[SDNR96] Amit Shukla, Prasad M. Deshpande, Jeffrey F. Naughton and Karthik Ramasamy. Storage Estimation for Multidimensional Aggregates in the Presence of Hierarchies. To appear in *Proc. of the 22nd VLDB Conference*, 1996.

[SAG96] Sunita Sarawagi, Rakesh Agrawal, and Ashish Gupta. On computing the data cube. Research Report RJ 10026, IBM Almaden Research Center, San Jose, California, 1996. Available from `http://www.almaden.ibm.com/cs/quest`.

[DANR96] Prasad M. Deshpande, Sameet Agarwal, Jeffrey F. Naughton and Raghu Ramakrishnan. Computation of Multidimensional Aggregates. *Technical Report-1314*, University of Wisconsin-Madison, 1996.

[Mic92] Z. Michalewicz. *Statistical and Scientific Databases*. Ellis Horwood, 1992.

[NC95] Pendse, Nigel and Richard Creeth. The OLAP Report. *Business Intelligence*, London, England, 1995.

[CODD93] E. F. Codd. Providing OLAP: An IT Mandate Unpublished Manuscript, E.F. Codd and Associates, 1993.

[FINK] Richard Finkelstein. Understanding the Need for On-Line Analytical Servers. Unpublished Manuscript, Performance Computing, Inc.

[Sho82] A. Shoshani. Statistical databases: Characteristics, problems and some solutions. In *Proceedings of the Eighth International Conference on Very Large Databases (VLDB)*, pages 208–213, Mexico City, Mexico, September 1982.

[SR96] B. Salzberg and A. Reuter. Indexing for aggregation, 1996. Working Paper.

[STG95] Designing the Data Warehouse on Relational Databases. Unpublished Manuscript, Stanford Technology Group, Inc, 1995.

[STL89] J. Srivastava, J.S.E. Tan, and V.Y. Lum. TBSAM: An access method for efficient processing of statistical queries. *IEEE Transactions on Knowledge and Data Engineering*, 1(4), 1989.

[WELD95] Jay-Louise Weldon. Managing Multidimensional Data: Harnessing the Power. Unpublished Manuscript, 1995.