Computer Sciences Department

Computation of Multidimensional Aggregates

Prasad Deshpande
Sameet Agarwal
Jeffrey Naughton
Raghu Ramakrishnan

Technical Report #1314

May 1997

UNIVERSITY OF
WISCONSIN
MADISON

1314

Computation of Multidimensional Aggregates *

Prasad M. Deshpande
pmd@cs.wisc.edu

Sameet Agarwal†
sameeta@microsoft.com

Jeffrey F. Naughton
naughton@cs.wisc.edu

Raghu Ramakrishnan
raghu@cs.wisc.edu
Computer Sciences Department
U. Wisconsin, Madision, WI-53706

Abstract

At the heart of 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. We explore various schemes for implementing multidimensional aggregation; in particular, the **CUBE** operator [1] proposed by Gray et al. This operator computes aggregates over all subsets of dimensions specified in the **CUBE** operation, 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, and present a class of sorting-based algorithms that overlap the computation of different group-by operations using the least possible memory for each computation.

Our algorithms seek to minimize the number of sorting steps required to compute the many sub-aggregates that comprise a multidimensional aggregate, and make efficient use of the available memory to reduce the number of I/Os. The implementation results show that the method dramatically outperforms the straightforward implementation of multidimensional aggregation as a sequence of SQL group-by operations. Finding a guaranteed optimal strategy within our framework to compute the **CUBE** operation is likely to be expensive; we prove that a simplified version of the problem is NP-hard. Our algorithm also deals efficiently with the common case where only a subset of the **CUBE** is to be computed.

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) [5, 7]. For example, in a retail application, one might have a relation with attributes *(Product, Year, Customer, Sales)*. An analyst could then query the data by asking the system to display:

- *sum of sales by (product, customer)*:
  For each product, give a breakdown on how much of it was sold to each customer.
- *sum of sales by (date, customer)*:
  For each date, give a breakdown of sales by customer.

*This work supported by a grant from IBM under the University Partnership Program
†Currently at Microsoft Corp.

- *sum of sales by (product, date)*:

  For each product, give a breakdown of sales by date.

Speed is a primary goal in this kind of application. Interactive analysis (response time in seconds) is not possible if each of these aggregates is computed "on the fly" at query execution time; so most products for these applications (OLAP [4] or Multidimensional Database (MDDB) systems) precompute the aggregates and run interactive sessions against the precomputed data. Speed is critical for this precomputation as well, since the precomputation time is a lower bound on the recency of the data available to an analysts. Further, the cost of precomputation influences how frequently the precomputed data is brought up-to-date.

1.1 What is a CUBE?

Tools for multidimensional data analysis comprise a large and fast-growing industry (estimates vary widely, but go as high as $700M for 1995 [3]). Recently there has been a lot of research interest in this area. One of the earlier papers is by Gray et al [1], where multidimensional aggregation is formalized and expressed in SQL by a new operator called the **CUBE** operator.

The **CUBE** operator is the n-dimensional generalization of the group-by operator. The CUBE operator on n dimensions is equivalent to a collection of group-by statements, one for each subset of the n dimensions:

```
CUBE [DISTINCT | ALL] <select list> BY <aggregate list> <table expression>
```

The semantics of the CUBE operator is that it computes aggregates (specified in the `<aggregate list>`) by grouping on all possible subsets of the attributes in `<select list>`. We use the term **cuboid** to denote a group-by aggregate. The group-by aggregate over all the attributes in `<select list>` is called the **base cuboid**. Thus for a CUBE over n attributes, 2^n cuboids including the base cuboid would have to be computed.

Returning to our retail example, consider the relation *(Product, Year, Customer, Sales)*. The following statement will compute the sales aggregate cuboids on all 8 subsets of the set {Product, Year, Customer} (including the empty subset):

```
CUBE Product, Year, Customer By SUM(sales)
```

1.2 Contributions of this Paper

We consider the problem of computing the CUBE operator efficiently. The CUBE operation on n attributes consists of computing a collection of 2^n cuboids, and the main challenge is to understand how the cuboids in this collection are related to each other, and to exploit these relationships to minimize I/O. An important contribution is that we bring out the relationships between the cuboids using a hierarchical structure, and formalize the problem of computing a CUBE in terms of this hierarchy. Thus, we provide a framework in which algorithms for computing the CUBE can be understood and evaluated.

We then present a class of sorting-based methods that try to minimize the number of disk accesses by overlapping the computation of the various cuboids, thereby reducing 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* method of computing

the CUBE by a sequence of group-by statements, which is currently the only option in commercial relational database systems.

Our algorithm uses some heuristics, and it is natural to ask whether there is a provably optimal algorithm. We address this issue by showing that choosing an optimal strategy, within our framework, for computing the CUBE is NP-hard. However, irrespective of the heuristics our method performs much better than computing the CUBE by a sequence of group-by statements since it reduces the number of scans and sorts required. Indeed, we provide some experimental evidence indicating that its I/O performance is often close to optimal (Section 6.3).

Often, because of storage limitations we may not want to precompute all the cuboids. This problem is considered in [2] which decides on an optimal set of cuboids to be precomputed given storage constraints. Our algorithm can be easily adapted to compute only a given set of cuboids.

The rest of the paper is organized as follows. In Section 2 we present different approaches to computing the CUBE. In Section 3, we give a description of our method of computation. We show that finding an optimal strategy is NP-hard in Section 4. We discuss some important issues in Section 5 and describe our implementation and experimental results in Section 6. We present other related work and compare our approach with theirs in Section 7. Finally, we present our conclusions in Section 8.

2 Options for Computing the CUBE

Consider the computation of a CUBE on k attributes $X = \{A_1, A_2, \ldots, A_k\}$ of relation R with aggregate function $F(\cdot)$. A *cuboid* on j attributes $S = \{A_{i_1}, A_{i_2}, \ldots, A_{i_j}\}$ is defined as

$$\text{CUBOID(S)} = \{(\mathrm{a_{i_1}}, \mathrm{a_{i_2}}, \ldots, \mathrm{a_{i_j}}, \mathrm{F}(\tau)) \mid \tau \subseteq \mathrm{R},\ \mathrm{t} \in \tau \Leftrightarrow \Pi_{\mathrm{A_{i_1}}, \mathrm{A_{i_2}}, \ldots, \mathrm{A_{i_j}}}(\mathrm{t}) = (\mathrm{a_{i_1}}, \mathrm{a_{i_2}}, \ldots, \mathrm{a_{i_j}})\}$$

Informally, the above cuboid is 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 [1]. The CUBE on attribute set X is defined as the union of cuboids on all subset of attributes of X.

The technique for computing a CUBE depends on the aggregation function. In this paper we assume that the function is *distributive.* Functions like *add, min* are distributive. For these functions super-aggregates can be computed from the aggregates. Thus if a set X is the disjoint union of sets Y and Z, then $sum(X) = sum(Y) + sum(Z)$. Such relations do not hold for a non-distributive function like *median.*

Computing the CUBE requires that we compute all the cuboids that together form the CUBE. The base cuboid has to be computed from the original relation; this can be done using any of the standard group-by techniques like sorting or hashing [8, 9, 10]. The other cuboids can be computed from the base cuboid due to the distributive nature of the aggregation. For example, in the retail application relation, *sum of sales by (product, customer)* can be obtained by using *sum of sales by (product, year, customer).* There are a variety of choices for scheduling the computations of the cuboids and also for the actual method of computing the group-by. Different combinations of these result in different techniques for computing the CUBE.

2.1 Scheduling the cuboid computations

- **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 $X \subset Y$. We also use the heuristic of computing a cuboid with $k-1$ attributes only from a cuboid with k attributes, since cuboid size is likely to increase with additional attributes, and it is in general better to compute using a smaller cuboid. 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 1.

  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.

- **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 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. This can significantly reduce the number of I/Os required. The details of this scheme are explained in Section 3.

2.2 Computing the Group-bys using Sorting

In relational query processing, there are various methods for computing a group-by, such as sorting or hashing. These methods can be used to compute one cuboid from another. 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.

- 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 3.

Using the above observations we can cut down on the number of sorts required. In fact only a sort of the base cuboid will be needed. From a base cuboid on k attributes we compute all cuboids on $k-1$ attributes and using these, all cuboids with $k-2$ attributes and so on.

For many matching tasks, it has been shown that hashing is a reasonable alternative to sorting [9]. It may be possible to apply the overlap method for hashing as well. However, we concentrate on Sorting based methods in this paper.

3 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.

3.1 Basic Definitions

Before we explain our scheme for CUBE implementation in detail, we define some terms which will be used frequently.

3.1.1 Sorted Runs and Partitions

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 $B = ((1,1,2),(1,1,3),(1,2,2),(2,1,3),(2,3,2),(3,3,1))$. Let S be the cuboid on the first and third attribute. i.e., $S = ((1,2),(1,3),(2,3),(2,2),(3,1))$. The sorted runs for S are $((1,2),(1,3))$, $((1,2))$, $((2,3))$, $((2,2))$ and $((3,1))$.

A *partition* of the cuboid S in B is a union of sorted runs such that the first $l-1$ columns of all the tuples of the sorted runs have the same value. Again, in the above example, the partitions for S in B will be $((1,2),(1,3))$, $((2,2),(2,3))$ and $((3,1))$.

This definition of a partition is motivated by the following observations, which will be useful as explained later.

- A cuboid can be considered as a union of pairwise disjoint partitions.
- All tuples of one partition are either less or greater than all tuples of any other partition in the sort order of the cuboid. This means that each partition can be computed independently of the others.

3.2 Overview of the Overlap Method

The algorithm begins by sorting the base cuboid. All other cuboids can be directly computed in sorted order without any further sorting. 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 1.

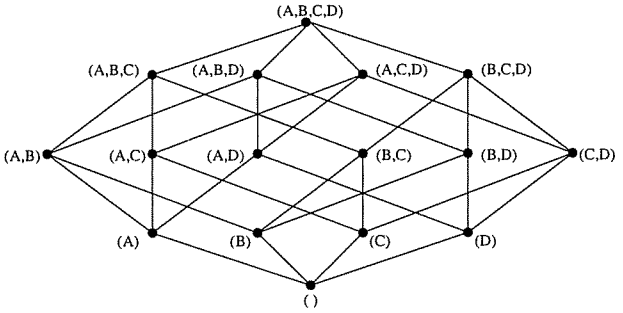


Figure 1: Sort orders enforced on the cuboids

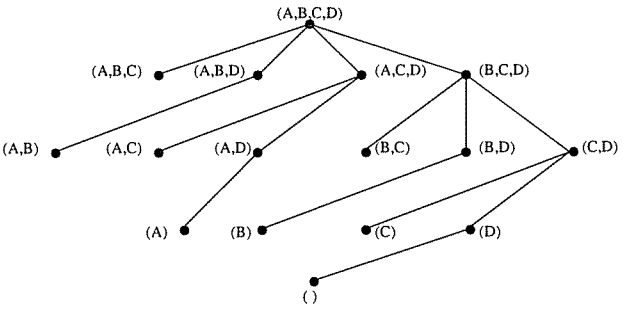


Figure 2: Cuboid Tree obtained from the cuboid DAG

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. The idea underlying this reduction is that while computing a cuboid from another sorted cuboid we just need memory sufficient to hold a partition of the cuboid. (Because all tuples of one partition are greater than all tuples of the previous partition, each partition can be computed independently of the others.) 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 whereas the partition size for (A, B, D) is bounded by the number of distinct values of D. Space enough to hold these partitions is sufficient for this computation. Thus computation of many cuboids can be overlapped in the available memory effectively reducing the number of scans.

There are three issues to be considered in the overlapped computation of cuboids:

1. Choosing a parent to compute a cuboid.
2. Computing a cuboid from its parent.
3. Choosing a set of cuboids for overlapped computation.

We discuss these issues in depth in the following section.

3.3 Details

Choosing a Parent to Compute a Cuboid

Each cuboid in the cuboid DAG has more than one parent from which it could be computed. By making a decision about which of these cuboids will be used, we convert the DAG to a rooted tree where 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 1 is as shown in Figure 2.

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 is the same as (A, C) order and the partition size is 1.

The partition sizes can be estimated from estimates of the cuboid sizes. Let us assume that some estimator provides us with the sizes of all the cuboids involved in a CUBE. If such information is not available, then it can be estimated by using the uniform distribution assumption given the number of distinct values of each attribute in the CUBE and the size of the base cuboid [11]. The size of a partition of a cuboid $(A_1, A_2, \ldots, A_{l-1}, A_{l+1}, \ldots, A_k)$ being computed from the cuboid $(A_1, A_2, \ldots, A_k)$ is approximately equal to

$$\frac{\text{Size of Cuboid } (A_1, A_2, \ldots, A_{l-1}, A_{l+1}, \ldots, A_k)}{\text{Size of cuboid } (A_1, A_2, \ldots, A_{l-1})} \tag{1}$$

This formula is obtained by using the uniform distribution assumption.

This gives us the following heuristic to minimize partition sizes: 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. The tree in Figure 2 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).

Computing a Cuboid From its Parent

Every cuboid (say S) other than the base cuboid is computed from its parent in the cuboid tree (say B). There are two options for computing S from B depending on the memory available for this computation. If we have sufficient memory to fit the largest partition of S in B in memory, we can compute the entire cuboid S in one pass over B. However, if we do not have sufficient memory to do this, we can write out sorted runs of S in the first pass and then merge these sorted runs to obtain the cuboid in a further pass. Writing out the sorted runs requires just one page of memory. In the former case we define the **state** of the computation to be *Partition* and in the latter case, we define the state to be *SortRun*. The algorithm for computing a cuboid is specified below :

Input: Memory allocated for this computation M, the state of the computation.

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 is the same as 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.

Here we assume that the memory allocated for the computation is larger than the size of largest partition. Later we describe how to handle the case when the partition does not fit in memory by allocating pages dynamically.

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 is the same as 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 is not the same as 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.

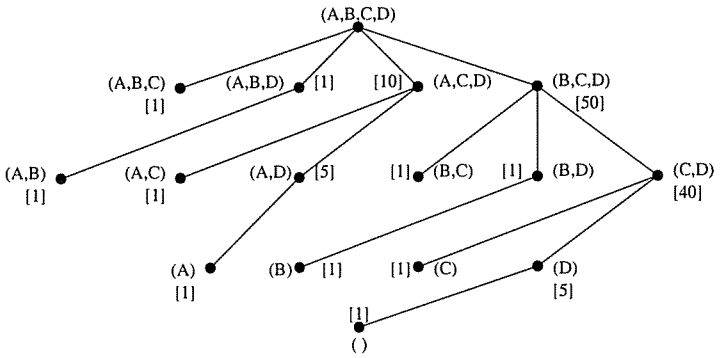
Choosing a Set of Cuboids for Overlapped Computation

We have to choose a set of cuboids that can be computed concurrently within the memory constraints. Out of these some will be in *Partition* state and require memory equal to the size of its partition. The remaining cuboids are in *SortRun* state and need just one page of memory each. We assume that we have estimates of sizes of the cuboids. The partition sizes can be computed from these (Equation 1).

Given any subtree of a cuboid tree and an initial amount 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.



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

Figure 3: Estimates of Partition Sizes

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 show that finding an overall optimal allocation scheme for our cuboid tree is NP-hard (Section 4). 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.

3.4 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 3. 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 4. In the second 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.

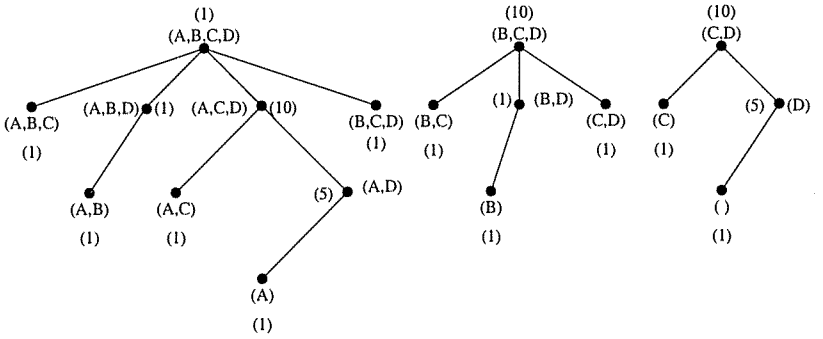


Figure 4: Steps of the algorithm

4 Finding an Optimal Allocation Strategy is NP-hard

4.1 Cost of Computation

We consider the number of page I/Os as a measure of the cost of a scheme. We have a tree of cuboids to be computed. For each cuboid we have an estimate of its size (say S) and its partition size (say P). The cost of writing out the cuboids is common to all the schemes. So we can ignore this while comparing the schemes. For cuboids being computed in the *SortRun* state there is an additional cost of writing out the sorted runs. For each of the subtrees computed in a pass, there is a cost of scanning the root of that subtree. Let A be the set of cuboids computed in *SortRun* state and B be the set of cuboids forming the roots of the subtrees generated by the scheme. Thus the total cost of any scheme is given by :

$$Cost = \sum_{i \in A} S_i + \sum_{i \in B} S_i$$

The optimal memory allocation scheme should try to minimize this cost.

Theorem : The optimization problem stated as follows is NP-hard:
Given: A cuboid tree with estimates of cuboid sizes and partition sizes, the memory available $M \in Z^+$.
Find: An allocation scheme that satisfies the constraints C1 and C2 and has the minimum cost.

4.2 Reduction from Modified Knapsack to Memory Allocation Problem

We prove NP-hardness by showing a reduction from a modified knapsack problem, in which the number of items is a power of two. The modified knapsack problem is also NP-hard (see Appendix A).

The difficulty in showing the optimal allocation problem to be NP-hard is that it is a multi-pass algorithm. It has to do allocation for a subtree of cuboids in each pass. The overall cost of computation depends on the decisions made in every pass. To show NP-hardness, we restrict the problem to instances in which the cost depends only on the decisions made in the first pass. The optimal algorithm should compute a subtree in the first pass and generate a set of subtrees. It should be possible to compute each of these generated subtrees in a single pass, i.e., these subtrees should not be split further. In this case, the total cost is determined just by the subtrees generated in the first pass. The number of possible generated subtrees should be of the same order as the number of cuboids to be computed, in order to prove that the problem is NP-hard in the number of cuboids.

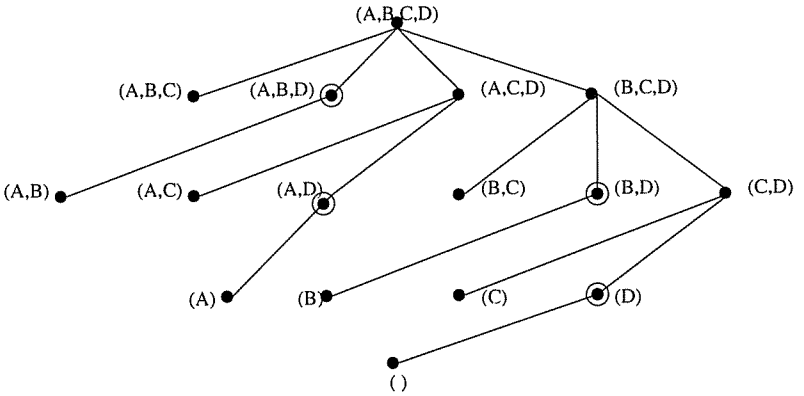


Figure 5: Nodes having One Child

Consider the nodes that have exactly one child in the cuboid tree shown in Figure 5. These nodes are marked in the figure. It turns out these nodes are plentiful in the tree. A cuboid tree for a CUBE with n attributes will have 2^n nodes, out of which 2^{n-2} nodes will have exactly one child. This can be shown by simple induction using the recursive structure of the tree.

Consider a modified knapsack problem which has N items where N is a power of 2, i.e. $N = 2^n$ for some n. We consider a cuboid tree for a CUBE on $n+2$ attributes. This tree will have $2^{n+2} = 4N$ nodes. The number of nodes in this tree having exactly one child is thus $2^n = N$. Let us denote the set of these nodes as Y, the set of children of these nodes as Y' and all the remaining nodes as Z. We now set up a one-one correspondence between the N items in the Knapsack problem and the N nodes in Y. If u is the item corresponding to the cuboid i, set $P_i = s(u)$ and $S_i = v(u)$, where P_i is the partition size and S_i is the cuboid size.

The mapping is intuitive since P_i is the memory needed to compute compute the cuboid in *Partition* state and is equivalent to the size taken up by the item if it included in the knapsack. Similarly S_i is a measure of the value of computing the cuboid in *Partition* state and corresponds to the value of the item in the knapsack case.

For nodes in Y' choose $P = 1$. Choice of S is immaterial. For each of the other nodes in Z choose S such that:

$$S > \sum_{i \in Y} S_i \tag{2}$$

The choice of P for these nodes is immaterial. Let the total memory available for allocation be:

$$M = \sum_{i \in Z} P_i + |Y| + B$$

This completes the instance of the memory allocation problem.

Now consider the optimal allocation for this instance. The memory is enough to fit a partition of each of the nodes in Z in memory and to allocate 1 page for writing out sorted runs for nodes in Y. The big sizes for nodes in Z (Equation 2) gives them higher priority . The optimal allocation will always allocate memory for their partitions before considering nodes in Y. Similarly the optimal allocation should allocate at least 1 page for each of the nodes in Y, so that at least sorted runs can to be written out for these nodes. Otherwise there would be additional scans required on nodes in Z and the solution will not be optimal.

Consider the decision to be made at this stage. The amount of memory left to be allocated is B. For each of the nodes i in Y we need $P_i - 1$ pages to hold its partition (1 page has already been allocated) and 1 page for its

child. Thus we need P_i pages to compute the node i and its child. If we don't give this amount of memory, sorted runs for i will be written out. They will be merged in a further pass during which the child will also be computed. The additional cost would be $S_i(to\ write\ out\ the\ runs) + S_i(to\ read\ and\ merge)$. Thus to minimize this cost, we have to allocate memory to cuboids i in Y such that $\sum_i S_i$ is maximized, i.e. get the maximum benefit out of it. This is precisely the solution to the instance of the knapsack problem. Thus we can find a solution to the knapsack instance iff we find a solution to this instance of the memory allocation problem.

In generating this instance there was only a linear increase in size (from N to $4 * N$), and the reduction is therefore polynomial-time. Since the modified knapsack problem is NP-hard the optimal allocation problem is also NP-hard in the number of cuboids.

5 Some Important Issues

5.1 Incorrect estimation

As we have already seen, the scheme relies on the estimates of the cuboid and partition sizes. However, in practice good estimates are hard to achieve and hence a partition might not fit in the memory allocated. Thus rather than statically allocating memory to a cuboid we may allocate memory dynamically. The estimates are used just to decide the state of each cuboid and actual allocation is done as and when needed.

5.2 Limiting the number of sorted runs

It is necessary to limit the number of sorted runs in order to avoid multiple levels of merging in the merging phase. We exploit the fact that all sorted runs of a partition are either greater than or less than all sorted runs of any other partition. Thus by directly appending sorted runs from different partitions to give longer sorted runs, we can limit the number of sorted runs to the maximum number of sorted runs in any partition. The number of sorted runs in any partition is bounded by the number of distinct values of the missing attribute.

Thus while computing cuboid $\{A, B, D\}$ from $\{A, B, C, D\}$ the maximum number of sorted runs will be equal to the number of distinct values of C.

5.3 Choosing an Initial Sort Order

Cuboids which form the rightmost children of their parents have the biggest partition sizes. Partition sizes increase from the left to right. Thus more memory is needed for the right cuboids to be computed in memory. The sort order should be such that the smaller cuboids are the rightmost children of their parents in the cuboid tree, so that they may fit in memory. As we show in the experiments section, this is important for skewed data where there may be a wide variation in the sizes of the cuboids. Thus if cuboid ABC is much smaller than cuboid BCD the sort order (D,C,B,A) will be better than sort order (A,B,C,D), since for (D,C,B,A) order cuboid ABC forms the rightmost child of the root.

Another option is to choose a sort order for the base cuboid such that the number of distinct values of the first column is less than the number of distinct values of the second column and so on. This will help in reducing the number of sorted runs for the rightmost cuboids whose partitions are least likely to fit in memory. This is important when some dimensions have a large number of possible distinct values. For example, if A has the least

number of distinct values, then sort order (A, B, C, D) is better. BCD is the rightmost cuboid and most likely sorted runs for it will be written out. The number of sorted runs is bounded by the number of distinct values of the missing column A. Thus it is good to have A with the least number of distinct values.

6 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 4.3.1. 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.

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.

6.1 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.

6.1.1 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 extremely low, 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 6.1.3 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 8 shows the performance of the three algorithms for uniform data. Figures 6 and 7 are for non-uniform data. We generated non-uniform data in two ways:

- **Zipf distributions**
  The values of each attribute were chosen from their domain using a Zipf distribution. We tried out different degrees of skewness in the data. For this experiment, values for attributes A and B were chosen with Zipf factor of 2, C with a factor of 1, and D and E with a factor of 0 (uniform distribution).
- **Sparse and Dense dimensions**
  A set of dimensions is *sparse* if only a small fraction of all possible combinations of values for these dimensions is actually present in the relation. A set of dimensions is *dense* if for each combination of the sparse

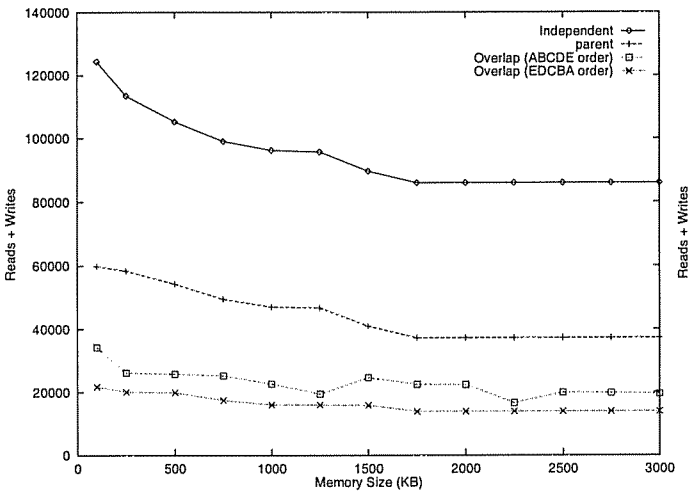


Figure 6: Non-uniform Data : Zipf Distribution : Input Size 2.4M, CUBE Size 10M

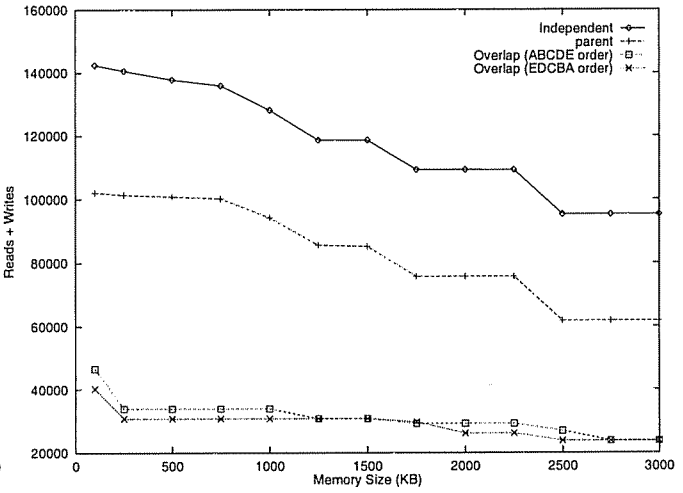


Figure 7: Non-uniform Data : Sparse and Dense Dimensions : Input Size 2.4M, CUBE Size 19.4M

dimensions, a significant fraction of the total number of combinations of values in these dimensions is present. Real data often has sparse and dense dimensions. In our experiments, (A,B) comprise the sparse dimensions and (C,D,E) comprise the dense dimensions.

The graphs in Figures 6, 7 and 8 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 8. For example, the I/O performance at memory size 1500K is worse than that at 1250K for our algorithm. This only shows that the breadth-first heuristic that we are using for memory allocation is not always optimal.

As mentioned earlier, choosing a proper sort order for the base cuboid is important. This can be seen for non-uniform data, where we tried out two sort orders 'ABCDE' and 'EDCBA' .

For Zipf-distributed data, the sort order 'EDCBA', which keeps the skewed attributes to the right, is better than sort order 'ABCDE'. This is because cuboids having attributes with skewed distributions are typically smaller. The 'EDCBA' sort order keeps these cuboids to the right in the tree and is therefore better, as explained in Section 5.3. For data with sparse/dense dimensions, again sort order 'ECDBA' is better. The reason is similar: cuboids having sparse attributes are smaller and these smaller cuboids are towards the right for the sort order 'EDCBA'.

6.1.2 Number of Attributes

In this experiment, we varied the number of attributes on which the CUBE is to be computed from 3 to 7. The input relation had 100,000 tuples (3.2M) and the memory was 320K. Each tuple was 32 bytes wide. The number of distinct values of the attributes were different in each case varying from 40 to 120. The graph in Figure 9 plots the sum of reads and writes for the three methods.

The graph shows that as the number of attributes is increased, the performance difference between the *Overlap* and the *Independent* (or *Parent* method) gets magnified. The number of cuboids is exponential in the number of attributes on which the CUBE is to be computed. Thus the number of scans and sorts in the *Independent* and *Parent* method increase exponentially with the number of attributes. The *Overlap* method tries to compute as

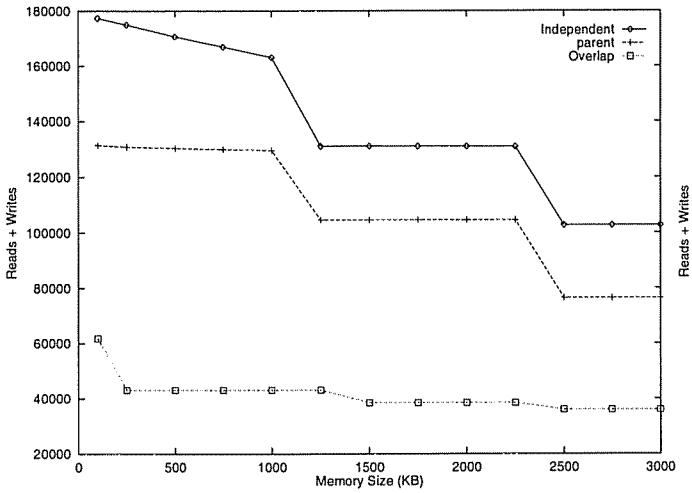


Figure 8: Uniform Data : Varying Memory : Input Size 2.4M, CUBE Size 27.1M

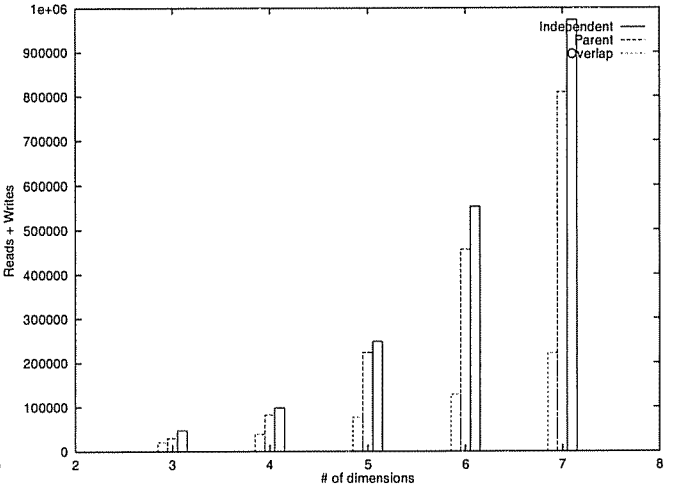


Figure 9: Uniform Data : Varying Number of Attributes : Input Size 3.2M, Memory Size 320K

many cuboids together as possible. Thus it does much better than the *Independent* and *Parent* method as the number of attributes is increased.

6.1.3 Scaleup Experiments

The relation size was varied from 100,000 (2.4M) to 1,000,000 tuples (24 M) to check how our method scales for larger input sizes with proportionately larger memory sizes. The memory used for each case was about 10% of the relation size. The graph in Figure 10 shows that the performance characteristics of the algorithms we consider are unchanged when the data sets are scaled to more realistic levels.

6.1.4 Computing only a subset of the cuboids

Very often we may not want to compute the entire CUBE, but only a few cuboids. This may be due to storage considerations or other reasons. For example, in [2] the authors consider the problem of deciding which cuboids are to be computed given some amount of disk storage. Our algorithm can be easily adapted to handle this case, where a only a given set of cuboids is to be computed: essentially, the tree of cuboids should be pruned to include only the cuboids that are to be computed. Figure 11 compares the three methods when only 1 cuboid, a quarter, half or all of the cuboids are to be computed. The three methods have identical performance when only one cuboid is to be computed. The difference in performance increases as the number of cuboids to be computed increases.

6.2 Comparison of Different Allocation Options

As explained earlier there a number of ways in which memory can be allocated to different cuboids for computation. Finding the optimal allocation was shown to be NP-hard. We implemented the optimal allocation algorithm and compared it with the heuristic of breadth-first memory allocation. The algorithm for finding the optimal allocation is exponential in the number of nodes in the cuboid tree, which itself is exponential in the number of attributes.

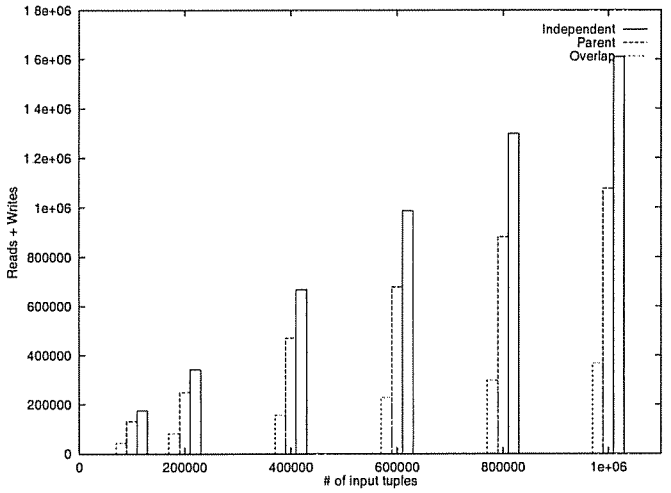


Figure 10: Scale up : I/Os

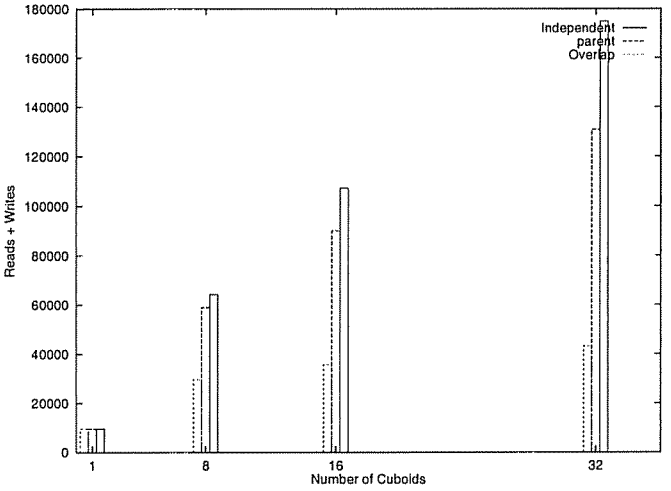


Figure 11: Computing a subset of Cuboids: Input Size 2.4M, CUBE size 27.1M, Memory 250K

Finding the optimal allocation takes an unreasonable amount of time for five or more attributes. Hence, we could only carry out this comparison for CUBEs on 4 attributes.

Pure BF does not perform well compared to the optimal for some cases. We identified some of the reasons and incorporated some more heuristics to prune the allocation generated by BF. The graph in Figure 12 compares the performance of the three methods (Optimal, BF, BF with pruning). The input relation has 100,000 tuples and the memory is varied from 250 to 3000 pages of $1K$ each. It can be seen that after pruning, BF performs very close to optimal. It should be noted that even though BF with pruning and the optimal methods match in this graph, BF with pruning is not always optimal. It is likely to be suboptimal for bigger cuboid trees (while computing CUBE on more attributes). However we couldn't present those figures since running the optimal for more than 4 attributes is not feasible. The heuristics yield performance quite close to the optimal. Thus for practical cases, finding the optimal allocation does not seem to be worth the effort. Also, finding the optimal allocation is not even feasible for CUBEs on five or more attributes.

6.3 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.

6.3.1 Varying Memory

Figure 13 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 13, 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. Our method is thus close to the optimal method of computing the cube.

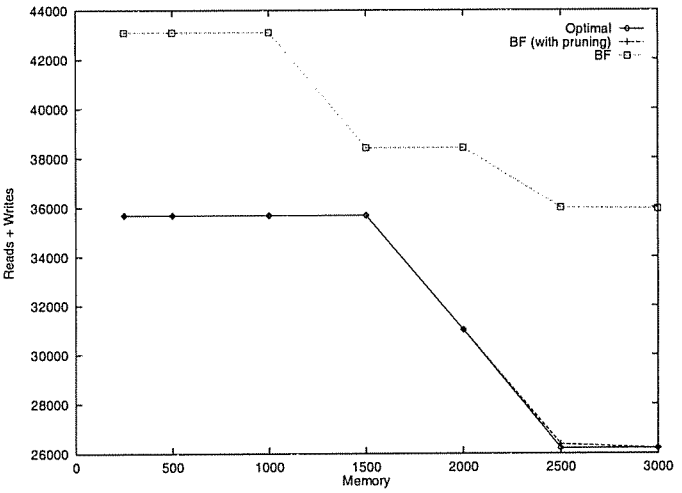


Figure 12: Comparison of Optimal and BF allocation

6.3.2 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 Figure 14. 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 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.

7 Related Work

There has been some concurrent work on the problem of computing the CUBE [14, 15]. In this section, we compare and contrast our approach to theirs. They have identifed a set of common optimizations that can be applied while computing the CUBE. Specifically, they name these as "Smallest-parent", "Cache- results", "Amortize-scans", "Share-sorts", "Share-partitions". They propose a sort-based method (called *PipeSort*) and a hash-based method (called *PipeHash*) which use these optimizations.

The *PipeSort* method takes into account the sizes of the group-by while creating a tree from the cuboid DAG. It considers the problem of choosing a parent as a minimum cost matching problem in order to reduce the cost of scanning and sorting. The method sets up simple pipelines and uses a different sort order for each pipeline. It thus makes use of the optimizations "Smallest-parent", "Cache-results", "Amortize-scans" and "Share-sorts".

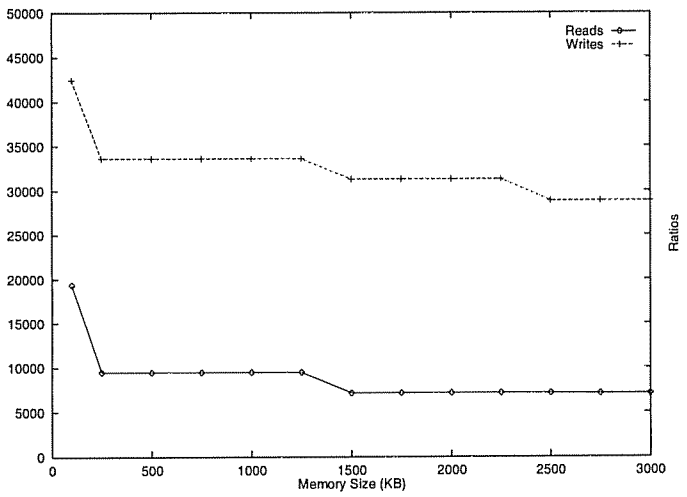


Figure 13: Varying memory : Relation : 2.4M; **CUBE** size : 27.1M

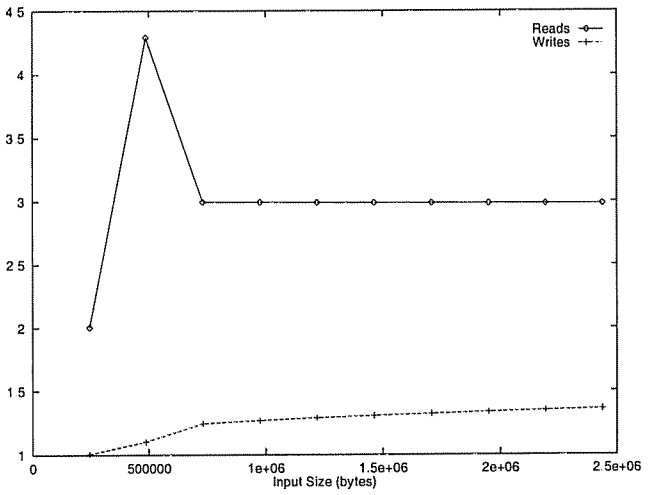


Figure 14: Varying relation sizes: Memory : 500K

The *PipeHash* method partitions the input data on some set of attributes. It then computes all the group-bys having these set of attributes using hashing. The data is partitioned such that for each partition, the hash tables for all the group-bys being computed fit in memory and can thus be computed simultaneously. This method uses the optimizations "Smallest-parent", "Cache-results", "Amortize- scans" and "Share-partitions".

The *Overlap* method we propose doesn't use the optimization of "Smallest-parent" directly. It gives more importance to matching sort orders while choosing a parent. The sort order of the root decides the sort order for all the other cuboids. By choosing the sort order appropriately most of the benefit of "Smallest-parent" can be achieved. The *Overlap* method uses more complex (branching) pipelines, which lead to more overlap. Thus "Cache-results" and "Amortize-scans" are used much more effectively than in the *PipeSort* method. Besides, the *Overlap* method makes use of partially matching sort orders, thus using "Share-sorts" very effectively.

We implemented the *PipeSort* method and the *PipeHash* method to compare their performance with the *Overlap* method. The implementation represents our interpretation of their methods; since they are not fully specified in the paper. They may vary in details from the actual methods. We performed three kinds of experiments. Unless otherwise mentioned the memory used in all these experiments was 0.5MB and the data was uniformly distributed. All graphs compare the execution times.

1. **Varying density :** (Figures 15, 16, 17 and 18) There are two ways of varying the density of the input data:
   - Varying the number of distinct values of a dimension. The input data had 4 dimensions 3 of which had 40 distinct values. The number of distinct values of the fourth dimensions was varied from 40 to 1000 to give densities ranging from 1% to 25%. The number of tuples was fixed at 640,000.
   - Varying the number of tuples. The number of distinct values of the dimensions were kept constant (at 40, 40, 40, 100) and the number of tuples was varied from 64,000 to 1.6 million.
2. **Varying the number of dimensions:** (Figures 19 and 20). The data density was kept constant at 1% and the number of dimensions was varied from 3 to 5. The number of input tuples was 640000. The three inputs had the following composition of distinct values – 40*400*4000, 40*40*40*1000 and 10*40*40*40*100.

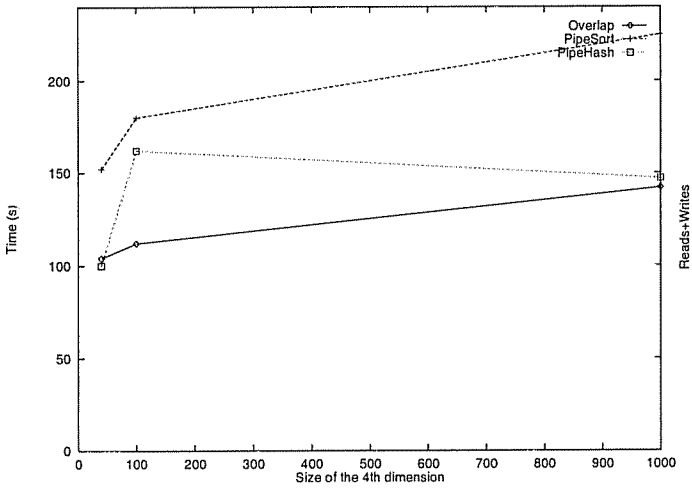


Figure 15: Varying Size of a Dimension : Execution Times

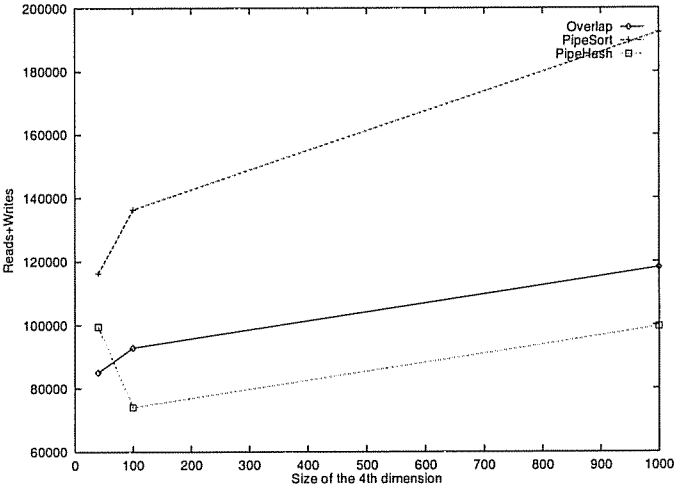


Figure 16: Varying Size of a Dimension : I/Os

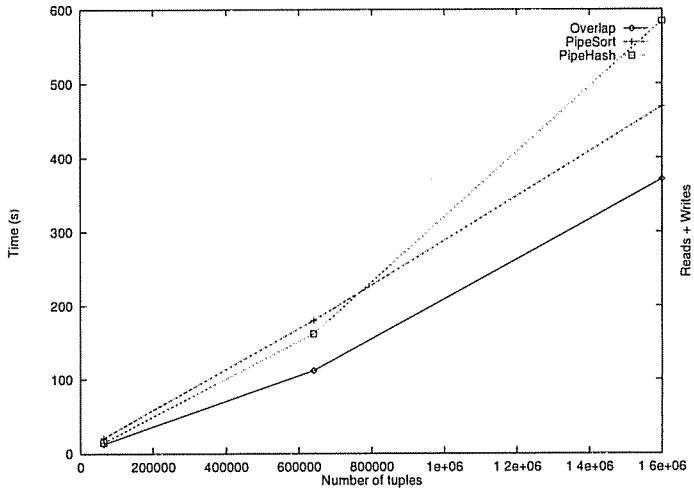


Figure 17: Varying Number of Tuples : Execution Times

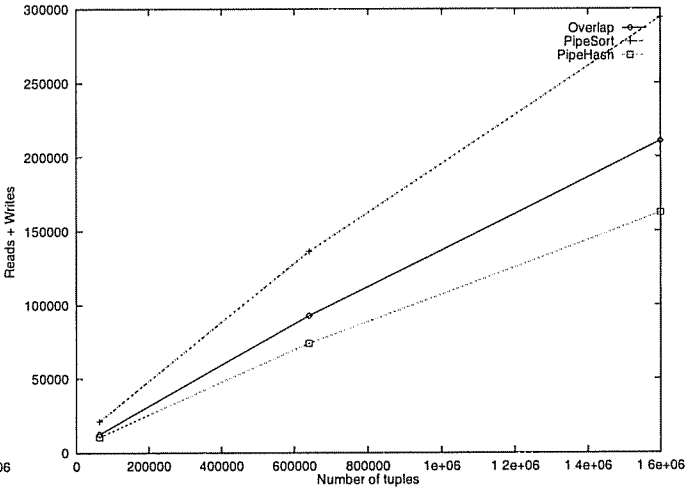


Figure 18: Varying Number of Tuples : I/Os

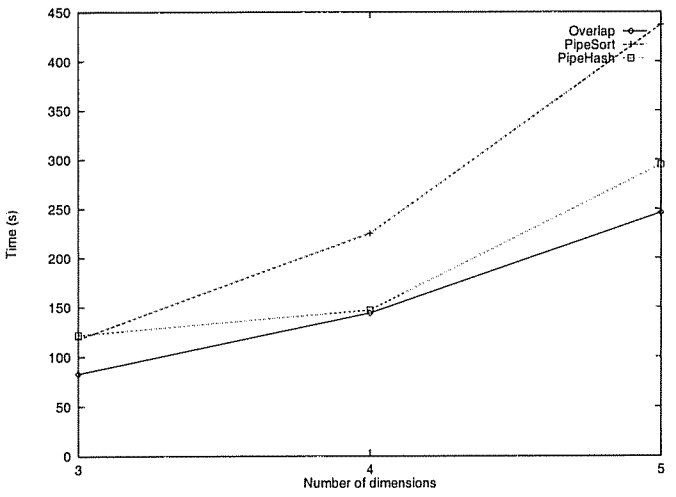


Figure 19: Varying Number of Dimensions : Execution Times

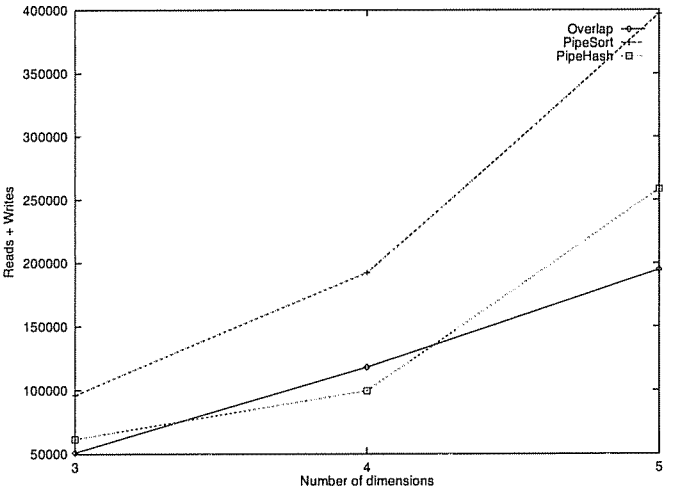


Figure 20: Varying Number of Dimensions : I/Os

Distribution	*Overlap*		*PipeSort*		*PipeHash*	
	Time(s)	I/Os	Time(s)	I/Os	Time(s)	I/Os
Uniform	434	359818	775	708935	587	380540
Non-uniform	317	267573	556	522742	586	300127

Table 1: Different Data Distributions

3. **Varying the Data Distribution:** (Table 1) Two data sets were tried. Each had five dimensions A,B,C,D,E with 20, 20, 20, 100 and 1000 distinct values. The input had a million tuples of 24 bytes each (total input size = 24MB). In one data set the data was uniformly distributed. In the other case, the data was non-uniform. The values for the first three dimensions were chosen with a skew factor of 2, 1 and 1 respectively. The memory used was 2.4MB (10% of input size) in each case.

The results show that *Overlap* performs consistently better than the *PipeSort*. It seems that *PipeSort* should do better in the case of non-uniform data, since it considers the cuboid sizes ("Smallest-parent" optimization). In case of non-uniform data, there is a wide variation in the sizes of the cuboids, so the "Smallest-parent" optimization should be important. The *Overlap* method does not use the "Smallest-parent"optimization. However by choosing a appropriate sort order (of EDCBA for the non-uniform data) most of the benefit of "Smallest-parent" can be achieved and *Overlap* still performs better. To account for the consistent better performance of *Overlap* let us consider the working of the *PipeSort* method for the five dimension case. It has to set up a separate pipeline each for ABCD, ABCE, ABDE, ACDE and BCDE. It will use different sort orders for these. Thus there are at least 5 scans and 4 sorts (4 sorts because one of these will have an order matching with the parent and will not need a sort) of the group-by ABCDE. Typically the group-bys at the higher level in the tree are much bigger than those at the lower level. Thus ABCDE will bigger than ABCD, ABC, AB or A. The *PipeSort* method has to do multiple scans and sorts of this big group-by ABCDE. It can be seen from the experiments that the number of I/Os for *PipeSort* is much greater than that for *Overlap*. The *Overlap* method uses partial sort orders and more overlap. Thus sorting costs are reduced. Also, ABCDE has to be scanned fewer times due to better use of "Amortize-Scans".

In most of the cases, *Overlap* performs better than *PipeHash*. But there are some cases where *PipeHash* is better. The experiments show that the number of I/Os for *PipeHash* is comparable to that of *Overlap*. However,

Memory	*Overlap*		*PipeHash*	
	Time(s)	I/Os	Time(s)	I/Os
0.5MB	246	194717	299	258462
1MB	221	184596	314	199041
2MB	222	183698	201	197469

Table 2: Varying Memory

the *PipeHash* method is sensitive to the number of partitions of the input data created. When the data is partitioned, all the partitions get created simultaneously. Thus, if there are many partitions, the pages allocated for each partition are scattered on the disk rather than being contiguous. As the memory available is increased, the number of partitions is reduced. A lesser number of partitions leads to fewer random I/Os and fewer disk seeks. The I/O cost for *PipeHash* reduces significantly. As a result *PipeHash* might perform better as the memory available is increased. This can be seen for the 5 dimensional case when the memory available is increased from 0.5MB to 2MB (Table 2).

8 Conclusions and Future Work

8.1 Future Work

There are several directions for future research. Our algorithm would benefit from accurate estimates of the cuboid sizes. Accurate estimation is an interesting research issue and we have some results on it [13]. We also need to study how the algorithm performs in the presence of incorrect estimates.

The algorithm we have presented is not the only way to compute the CUBE, of course. For example, one interesting approach is to sort the base cuboid in two different sort orders. For example, the cuboid $\{A, B, C, D\}$ could be sorted in sort orders (A, B, C, D) and (D, C, B, A). This will split the cuboid DAG into two trees. The cuboids which are worst in terms of the partition size for one sort order will be the best for the other order. The performance implications of such an approach merits further investigation.

8.2 Summary of Contributions

In this paper we have examined various schemes to implement the CUBE operator. Sorting-based methods have a significant advantage because a cuboid sorted in a particular order can be used to compute other cuboids which may have a sort order different from its sort order. Also, sorting enables us to pipeline the computation of the various cuboids, thus saving 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 minimizing the sorts, 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. It also performs well in the common case where only a subset of the cuboids are to be computed.
- 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.

References

[1] Jim Gray, Adam Bosworth, Andrew Layman and Hamid Pirahesh, "Data Cube: A Relational Operator Generalizing Group-By, Cross-Tab and Sub-Totals", to appear in IEEE transactions on Knowledge and Data Engineering.

[2] Venky Harinarayan, Anand Rajaraman and Jeff Ullman, "Implementing Data Cubes Efficiently", ACM SIGMOD, 1996.

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

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

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

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

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

[8] Robert Epstein, "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.

[9] Goetz Graefe, "Query Evaluation Techniques for Large Databases", *ACM Computing Surveys, 25(2):73-170*, June 1993.

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

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

[12] Michael R. Garey and David S. Johnson, *Computers and Intractability*, W. H. Freeman and Company, pp 45-76,65,96,247; 1979.

[13] Amit Shukla, Prasad M. Deshpande, Jeffrey F. Naughton and Karthik Ramasamy, *Storage Estimation for Multidimensional Aggregates in the Presence of Hierarchies*, submitted to VLDB '96

[14] Sunita Sarawagi, Rakesh Agrawal and Ashish Gupta, *On Computing the Data Cube*, submitted to VLDB '96

[15] Sunita Sarawagi, Rakesh Agarwal and Ashish Gupta, *On Computiing the Data Cube*, Research Report RJ 10026, IBM Almaden Research Center, San Jose, California, 1996.

A Modified Knapsack Problem

Consider the 0/1 Knapsack problem which is known to be NP-hard [12]:
INSTANCE: A finite set U, a size $s(u) \in Z^+$ and a value $v(u) \in Z^+$ for each $u \in U$, a size constraint $B \in Z^+$.
Find: a subset $U' \subseteq U$ such that $\sum_{u \in U'} s(u) \leq B$ and $\sum_{u \in U'} v(u)$ is maximum.

We generate an instance of the knapsack problem in which the number of elements in the set U is a power of 2. Suppose we need to add m elements to the set U to make the total number of elements a power of 2. First construct an instance in which the value of each item is scaled by $2m$. Thus the new instance is:
NEW INSTANCE 1: A set $U_1 = U$. For each $u \in U_1$, $s_1(u) = s(u)$ and $v_1(u) = 2m * v(u)$. The size constraint $B_1 = B$.

It is clear that U' is a solution of this new instance iff it is a solution of the original instance. All the values are scaled equally, hence the set with the maximum value for one instance still remains the optimal for the other instance.

We then generate another instance from this by adding m elements each with their size and weight set as 1.

NEW INSTANCE 2: A set $U_2 = U_1 \cup X$, where X has m elements. For each $u \in X$, $s_2(u) = 1$ and $v_2(u) = 1$. For each $u \in U_1$, $s_2(u) = s_1(u)$ and $v_2(u) = v_1(u)$. The size constraint $B_2 = B_1$.

If U' is a solution of instance 1, then $U' \cup X'$ is a solution of instance 2, where $X' \subseteq X$ and the number of elements in X' is $B_2 - \sum_{u \in U'} s_2(u)$. Basically the remaining capacity of the knapsack is filled by elements in X.

Similarly if $U' \cup X'$ is a solution of instance 2, where $U' \subseteq U_1$ and $X' \subseteq X$ then U' is a solution of instance 1, i.e. the extra elements are dropped.

Both these statements are valid because the value of each element in U_1 is a multiple of $2m$. Hence the value of any subset of U_1 is a multiple of $2m$. This means that the difference in the value between the optimal solution and any other set satisfying the size constraint is a multiple of $2m$, i.e. the difference is greater than m. Thus a non-optimal set in instance 1 cannot become a optimal set in instance 2 by adding elements in X.

Note that $m < |U|$. Thus the number of elements increases by a factor less than 2. Both these reductions are polynomial time. Hence the 0/1 Knapsack problem restricted to cases where the number of items is a power of 2 is NP-hard.