

# Materialized View Selection for Multidimensional Datasets\*

Amit Shukla  
amit@cs.wisc.edu

Prasad M. Deshpande  
pmd@cs.wisc.edu  
Computer Sciences Department  
University of Wisconsin - Madison  
Madison, WI 53706

Jeffrey F. Naughton  
naughton@cs.wisc.edu

## Abstract

To fulfill the requirement of fast interactive multidimensional data analysis, database systems precompute aggregate views on some subsets of dimensions and their corresponding hierarchies. However, the problem of what to precompute is difficult and intriguing. The leading existing algorithm, BPUS, has a running time that is polynomial in the number of views and is guaranteed to be within  $(0.63 - f)$  of optimal, where  $f$  is the fraction of available space consumed by the largest aggregate. Unfortunately, BPUS can be impractically slow, and in some instances may miss good solutions due to the coarse granularity at which it makes its decisions of what to precompute. In view of this, we study the structure of the pre-computation problem and show that under certain broad conditions on the multidimensional data, an even simpler and faster algorithm, PBS, achieves the same  $(0.63 - f)$  bound. Our empirical study of the behavior of PBS shows that even when this condition does not hold, PBS picks a surprisingly good set of aggregates for precomputation. Furthermore, BPUS and other previous work has assumed that all aggregates are either precomputed in their entirety or not at all. We show that if one relaxes this and allows aggregates to be partially precomputed, not only is it possible to find solutions that are better than those found by previous algorithms, in some cases it is even possible to find solutions that are better than the solution that is 'optimal' by the previous definition.

This research is supported by a gift from NCR Corp., and by ARPA through Rome Air Force Laboratory contract F30602-97-2-0247

*Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.*

Proceedings of the 24th VLDB Conference  
New York, USA, 1998

## 1 Introduction

Multidimensional data analysis, as supported by OLAP systems, requires the computation of many aggregate functions over large amounts of data. To meet the performance demands imposed by these applications, virtually all OLAP products resort to some degree of precomputation of these aggregates. The more that is precomputed, the faster queries can be answered; however, it is often difficult to determine which are the best aggregates to be precomputed given a fixed amount of space. Thus, the database administrator tries to fill available space with precomputed aggregates in order to minimize the average query response time of the system. An important problem a DBA faces is determining the amount of space that should be allocated for precomputation. A graph of the average query response time corresponding to different amounts of space allocated for precomputation is shown in Figure 1. Such a graph can be used to make an intelligent decision of the amount of space for precomputation such that the performance gains from adding more space are diminishing. A DBA can compute the slope of the graph at different points and determine the point at which diminishing returns from precomputation outweigh the cost of additional disk space.

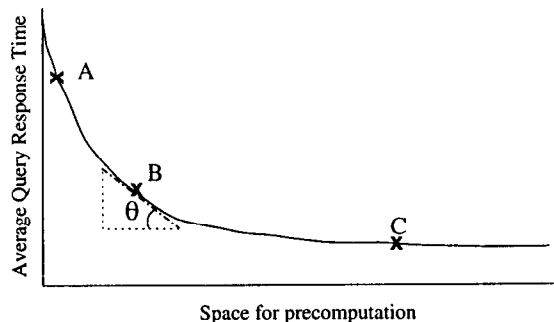


Figure 1: A graph of space vs average query cost for a lattice.

Constructing this graph is non-trivial. The problem has been shown to be computationally intractable. Harinarayan et al. [HRU96] have proposed an elegant heuristic algorithm, BPUS, to approximate the optimal

solution. Using BPUS, one can determine the best set of aggregates for the corresponding amount of precomputation space, and then evaluate the benefit of these aggregates. While BPUS is much faster than an exhaustive search, in general it will take several days to several months to generate the graph, rendering this approach infeasible, which in turn means the DBA will have to resort to guesses as to what is a good amount of space to dedicate for precomputation.

In this paper, we propose a simple and fast heuristic algorithm, PBS, to select aggregates for precomputation. PBS runs several orders of magnitude faster than BPUS, and is fast enough to make the exploration of the time-space tradeoff feasible during system configuration. However, PBS, like previous solutions, is a heuristic algorithm, so a main contribution of this paper is an exploration of its performance. We examine the conditions under which PBS selects views having a fixed bound with respect to the optimal set of views. Due to its speed, PBS can be used by DBAs to determine how much space should be allocated for precomputation. Next we examine the materialized view selection problem when subsets of aggregates can be computed using chunks [DRSN98], and show with an example that the benefit of the views selected by PBS using chunks can be greater than the benefit of the optimal set of views selected without chunk based precomputation. Then we show how BPUS and PBS can be adapted to use chunk based precomputation. This results in improved performance bounds for both algorithms. We begin with an example to motivate the problem.

### 1.1 An Example

Consider a table of sales with the schema

```
Sales(ProductId, StoreId, TimeId, Sales)
```

with the intuitive meaning that each tuple represents the sales of some product sold in some store at some time. We will use this schema as an example throughout the paper. There are a number of queries that can be asked of this data. For example, one may wish to know sales by product; or sales by store; or sales by product and store; or sales by store and time; and so forth. Each of these queries represents an aggregate computation. For example, sales by product and store in SQL is just:

```
SELECT ProductId, StoreId, SUM (Sales)
FROM Sales
GROUP BY ProductId, StoreId
```

If the sales table is large, this query will be slow. However, if this aggregate is precomputed, the query can be answered with a simple scan of the precomputed aggregate. In addition, the precomputation of this aggregate also benefits queries on aggregates derived from it. For example, the following query, which asks for the aggregated sales grouped by the product can be answered using the above aggregate.

```
SELECT ProductId, SUM (Sales)
FROM Sales
GROUP BY ProductId
```

Therefore, the task the DBA faces is to choose a set of queries to precompute and store.

### 1.2 Related Work

A useful way to describe the full precomputation problem is to use the framework proposed by Gray *et al.* [GBLP96] using the *cube* operator. The cube operator is the  $n$ -dimensional generalization of the SQL group-by operator. The cube on  $n$  attributes computes the group-by aggregates for each possible subset of these dimensions. In our example, this is: {}, {ProductId}, {StoreId}, {TimeId}, {ProductId, StoreId}, {ProductId, TimeId}, {TimeId, StoreId}, {ProductId, StoreId, TimeId}.

As we mentioned, [HRU96] proposes a greedy algorithm, BPUS, to find a set of aggregates to materialize. BPUS attempts to maximize the benefit of the set of aggregates picked. They prove that if the largest aggregate view occupies a fraction  $f$  of the space available for precomputation, then the aggregates picked by BPUS have a benefit at least  $(0.63 - f)$  times the benefit of the optimal set of views for the same amount of space. Other related work includes [GHRU97], where the authors consider the selection of views and indexes together. [Gupt97] presents a theoretical framework for the view-selection problem, and proposes a general algorithm and several heuristics, while [Ull96] surveys techniques proposed for determining what aggregates should be precomputed.

### 1.3 Paper Organization

Section 2 describes the lattice framework and cost model for the aggregate selection problem. In Section 3, we describe a fast aggregate selection algorithm along with bounds on its performance. Section 4 presents subset caching using chunks and shows how it affects the aggregate selection problem. We carry out an experimental evaluation of the different precomputation algorithms in Section 5, and present insights into the problem of aggregate selection. Section 6 presents our conclusions.

## 2 Problem Formulation

### 2.1 Lattice Framework for Multidimensional Datasets

Queries on multidimensional datasets can be modeled by the data cube operator. For distributive function such as sum, min, max, etc., some aggregates can be computed from another aggregate. In the example schema of Section 1.1, the aggregate on {ProductId, StoreId} can be used to answer a query on {ProductId}. This relation between aggregate views can be used to place them within a lattice framework as proposed in [HRU96] and [BPT97]. Aggregates are vertices of an  $n$ -dimensional cube. The following properties define a hypercube lattice  $\mathcal{L}$  of aggregates.

- (a) There exists a partial order  $\preceq$  between aggregate views in the lattice. For aggregate views  $u$  and  $v$ ,  $v \preceq u$  if and only if  $v$  can be answered using the results of  $u$  by itself.
- (b) There is a base view in the lattice, upon which every view is dependent. The base view is the database.

- (c) There is a completely aggregated view “ALL”, which can be computed from any other view in the lattice.

The aggregate selection problem is equivalent to selecting vertices from the underlying hypercube lattice. For example, the lattice  $\mathcal{L}$  in Figure 2 represents the cube of the schema described in Section 1.1. The three dimensions ProductId, StoreId, TimeId are represented by P, S, T respectively, and an aggregate view is labeled using the names of the attributes it is aggregated on. For example, view PS is aggregated on attributes ProductId and StoreId. In Figure 2, if an edge connects two views, then the *higher* view can be used to precompute the other view. For example, there is an edge between PS and P. This means that PS can be used to compute P. If there is no precomputation, a query on P (ProductId) has to be answered using the base data, PST (Sales table).

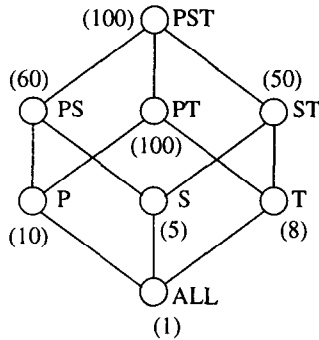


Figure 2: The hypercube lattice corresponding to the example in Section 1.1. The numbers are aggregate sizes in tuples.

One can define some functions on hypercube lattices. For an aggregate  $v$ , parent, children and descendant are defined as follows:

$$\begin{aligned} \text{parent}(v) &= \{u \mid v \prec u; \nexists w, v \prec w, w \prec u\} \\ \text{children}(v) &= \{u \mid u \prec v; \nexists w, u \prec w, w \prec v\} \\ \text{descendants}(v) &= \{u \mid u \preceq v\} \end{aligned}$$

## 2.2 Cost Model

We use the cost model proposed by [HRU96], in which the cost of answering a query (time of execution) is assumed to be equal to the number of tuples in the aggregate used to answer the query. To justify this cost model, they considered the different situations that can occur when answering a query from an aggregate. Namely, the presence or absence of an index on the aggregate, and whether the query asks for an entire aggregate or a subset of an aggregate. An experimental validation of this cost model is provided in [HRU96]. They found that there is an almost linear relationship between size and running time of a query. In summary, we assume that the cost of answering a query  $q$  is equal to the number of tuples read to return the answer.

## 2.3 The Benefit Metric

Informally, the *benefit of an aggregate view  $v$*  is computed by adding up the savings in query cost for each view  $w$  (including  $v$ ) over answering it from the base

view. If a set  $\mathcal{S}$  of aggregate views is chosen for materialization, the benefit of  $\mathcal{S}$  is the sum of the benefits of all views in  $\mathcal{S}$ . The same metric is used by [HRU96].

**Definition 2.1.** Let  $\mathcal{S}$  be the set of aggregates selected for precomputation. The least cost view in  $\mathcal{S}$  which can be used to answer a query on  $v$  is denoted by  $L(v)$ .

Let  $C(v)$  be the cost of computing another view from  $v$ . Looking back to our cost model,  $C(v)$  is the number of tuples in  $v$ .

**Definition 2.2.** The benefit of  $u$  with respect to  $\mathcal{S}$ ,  $B(u, \mathcal{S})$ , is defined as

1. For each aggregate view  $v \preceq u$ ,  $B_v$  is defined as:
  - 1.1 Let  $w$  be the least cost view in  $\mathcal{S}$  such that  $v \preceq w$ . ( $w = L(v)$ )
  - 1.2 If  $C(u) < C(w)$ , then  $B_v = C(w) - C(u)$ , else  $B_v = 0$
2.  $B(u, \mathcal{S}) = \sum_{v \preceq u} B_v$

In short, for each view  $v$  that is a descendant of  $u$ , we check to see if computing  $v$  from  $u$  is cheaper than computing  $v$  from any other view in the set  $\mathcal{S}$ . If this is the case, then precomputing  $u$  benefits  $v$ . Since all aggregates can be computed from the (unaggregated) base data, and  $\mathcal{S}$  contains the base data, in step 1.1 we can always find a least cost aggregate view  $w$  (the base data in the worst case). Our goal is to maximize the benefit of a set of aggregates given a fixed amount of space. This leads us to define another metric derived from benefit. In the rest of the paper, we will use benefit per unit space rather than just benefit since we assume that space is a constraint.

**Definition 2.3.** The benefit per unit space of a view  $u$  is defined as:

$$B_s(u, \mathcal{S}) = \frac{B(u, \mathcal{S})}{|u|} = \frac{1}{|u|} \sum (C(L(v)) - C(u)),$$

where  $v \prec u$ ,  $C(L(v)) > C(u)$ ,  $|u| = C(u) = \text{size of } u$ .

## 2.4 Average Query Cost

The effect of maximizing benefit per unit space of a set of aggregates is not intuitively obvious. In particular, it is hard to understand the improvement in query response time when the benefit increases by some amount. To overcome this drawback of benefit, we define a new metric *Average Query Cost*, which is analogous to query response time. An improvement in average query cost is equivalent to a corresponding improvement in average query response time. We use this characteristic in the experimental evaluation, where average query cost is used as the metric to plot graphs and explore query response time as space available for precomputation is increased.

Consider a lattice  $\mathcal{L}$  with  $n$  views,  $v_1, \dots, v_n$ . There are  $n$  different *templates* for queries, one for each view:  $Q_1, Q_2, \dots, Q_n$ . Let there be a set  $\mathcal{S}$  of aggregate views precomputed, so that a query on view  $v_i$  can be most cheaply answered from a view  $L(v_i) = w_i \in \mathcal{S}$ . Let queries on  $\mathcal{L}$  occur with probabilities  $p_1, p_2, \dots, p_n$ .

**Definition 2.4.** The average query cost is defined as:

$$\sum_{i=1}^n p_i C(w_i), \quad (1)$$

where  $C(w_i)$  is the cost of answering a query  $Q_i$  on a view  $v_i$ .

## 2.5 Reconciling these two metrics

We have two different metrics for the goodness of a precomputation. The first is maximizing the benefit per unit space of a precomputation, and the second is minimizing the average query cost. In this section we show that optimizing either of these two metrics leads to the same solution. First, we have to account for query probabilities in the benefit computation. We do this by modifying step 2 of the benefit computation in Definition 2.2 to:

$$2. \mathcal{B}_s(v, S) = \frac{1}{|v|} \sum_{u \preceq v} p_u \cdot \mathcal{B}_u$$

This modified benefit formula is used in [HRU96] to maximize the benefit of a set of aggregates ( $|D|$  is the size of the Database,  $D$ ):

$$\sum_{v \in \mathcal{L}} p_v \cdot (|D| - C(L(v))) = |D| - \sum_{v \in \mathcal{L}} p_v \cdot C(L(v)) \quad (2)$$

We prove in [SDN98] that maximizing the benefit (Equation 2) is the same as minimizing the average query cost (Equation 1).

## 2.6 Precomputation of Aggregates

There is a tradeoff between the amount of space allocated for precomputation, and the average query cost (query response time); more space implies a smaller average cost of queries (faster response time). Precomputing a subset of the aggregates involves intelligently picking a subset of the aggregates using a limited amount of space. For example, consider a lattice in which all aggregates have an equal probability of being queried. Figure 3 shows the average cost of queries on the lattice as the amount of space allocated for precomputation is increased. The average query cost corresponds to the optimal set of aggregates for a given amount of space. A small amount of precomputation dramatically improves the average query cost. The improvement in average query cost decreases as more space is allocated for precomputation.

To find the optimal set of views to precompute, one can enumerate all possible combinations of aggregate views and find the one which results in the minimum average query cost or the maximum benefit. Finding the optimal set of aggregates in this manner has a complexity of  $O(2^n)$ , where  $n$  is the number of aggregates in the schema. If  $d$  is the number of dimensions in the schema, and there are no hierarchies, then  $n = 2^d$ , so the cost of finding the optimal combination grows as  $2^{2^d}$ . For 6 dimension and no hierarchies, the cost is of the order of  $2^{64}$ ! Clearly, computing the optimal set of aggregates exhaustively is not feasible; this is in fact an intractable problem (NP-hard).

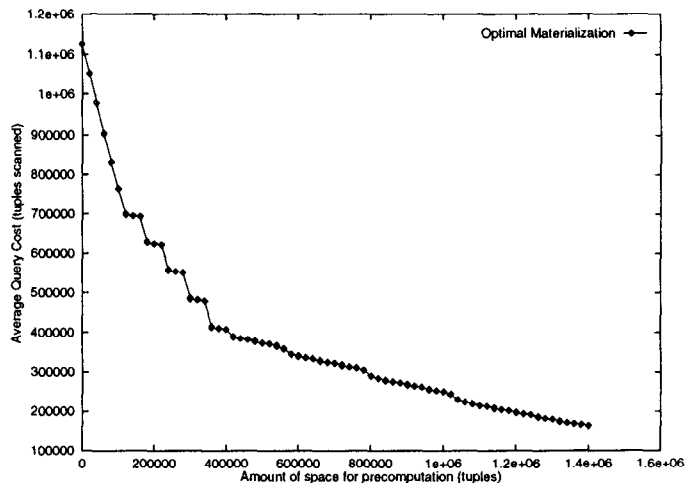


Figure 3: Average Query Cost of a 4 dimensional lattice as the amount of space available for precomputation is increased. The average query cost corresponds to the optimal set of aggregates for a given amount of space.

In view of this, [HRU96] presents an algorithm, BPUS, that uses the benefit per unit space of an aggregate. The inputs to BPUS are: space – the amount of space available for precomputation, and  $\mathcal{A}$ , a set initially containing all aggregates in the lattice, except the database. The output is  $\mathcal{S}$ , the set of aggregates to be precomputed. The algorithm assumes that queries are uniformly distributed.

---

### Algorithm BPUS

```

WHILE (space > 0) DO
  w = aggregate with max. benefit per unit space in  $\mathcal{A}$ 
  IF (space - |w| > 0) THEN
    space = space - |w|
     $\mathcal{S} = \mathcal{S} \cup w$ 
     $\mathcal{A} = \mathcal{A} - w$ 
  ELSE
    space = 0

```

$\mathcal{S}$  is the set of aggregates picked by BPUS

---

The authors proved that the benefit of the aggregates selected by BPUS is no worse than  $(0.63 - f)$  times the optimal benefit; if no aggregate view occupies more than some fraction  $f$  of the total space available for precomputation. There is no bound if  $f > 0.63$ . For example, if some view occupies 50% of the space available then all one knows is that the result achieves at least 13% of the optimal benefit. The order of BPUS is  $O(k \cdot n^2)$ , where  $k$  is the number of aggregates selected for precomputation, and  $n$  is the number of vertices in the lattice. We have observed experimentally that 10% to 50% of the aggregates are picked before improvements in response time diminish as additional space is allocated for precomputation. If 10% of the aggregates are selected for precomputation,  $k = n/10$ , and the complexity of BPUS becomes  $O(n^3)$ . Though the complexity is polynomial, for a real dataset it may result in too much time spent in making a decision of what aggregates should be precomputed as we see next.

### 3 Fast Aggregate Selection

A major drawback of the BPUS algorithm is its running time. To measure the execution time of BPUS, we implemented it with complexity  $O(k \cdot n^2)$ , and used a six dimensional schema with a five level hierarchy on each dimension, with hierarchy sizes (100, 50, 25, 5, 2). The first number (100) is the cardinality of the dimension, and the other numbers are (in order) the distinct values in its hierarchy. The corresponding datacube lattice has 46656 aggregates. The database had 10M tuples, and the space allocated for precomputation was 1.6 billion tuples, which is about 1% of the cube size. The machine it was run on was a 200 MHz dual processor Ultra Sparc I running Solaris 2.5.2, with 256 MB of main memory. For this schema, our implementation of BPUS took 1193 secs to pick one aggregate. If 10% of the aggregates are picked, BPUS will pick 4665 aggregates, and take about 64 days to determine the set of aggregates to be precomputed. If one tries to plot a graph such as in Figure 1, we estimate that it will take around 640 days to do so using BPUS. Clearly, this is a large amount of time that a user has to wait before he knows how much space should be used, or what aggregates should be precomputed. With this in mind, our goal is to design an efficient algorithm that doesn't sacrifice accuracy for speed. We designed a simple algorithm, PBS, with an  $O(n \log n)$  worst case execution time, and proven performance bounds for PBS under certain conditions on hypercube lattices. On the above 6 dimensional schema, PBS run in less than 1 second, compared to 64 days for BPUS. In addition, in Section 5, we experimentally evaluate PBS and show that it performs well even when the conditions on the lattice are relaxed. PBS can be used to plot a graph such as Figure 1, which a DBA can use to compute the slope of the graph at different points and determine the point at which diminishing returns from precomputation outweigh the cost of additional disk space.

#### 3.1 PBS

PBS, which stands for *Pick By Size*, picks aggregates for precomputation in increasing order of their size. The inputs to PBS are: space – the amount of space available for precomputation and  $\mathcal{A}$ , a set initially containing all aggregates in the lattice. The output is  $\mathcal{S}$ , the set of aggregates to be materialized. PBS assumes that queries on all aggregates are equally likely.

---

#### Algorithm PBS

```

WHILE (space > 0) DO
  w = smallest( $\mathcal{A}$ )
  IF (space - |w| > 0) THEN
    space = space - |w|
     $\mathcal{S} = \mathcal{S} \cup w$ 
     $\mathcal{A} = \mathcal{A} - w$ 
  ELSE
    space = 0
 $\mathcal{S}$  is the set of aggregates picked by PBS

```

---

In the algorithm,  $\text{smallest}(\mathcal{A})$  is a function which returns the view having the smallest size in set  $\mathcal{A}$ . The

order of PBS is  $O(n \log n)$ , which arises from the cost of sorting the aggregates by size. Next, we explore the conditions under which PBS guarantees the  $(0.63 - f)$  bound by proposing a subset of hypercube lattices which we will call SR-hypercube lattices.

#### 3.2 SR-Hypercube Lattices

Consider the subclass of hypercube lattices with an ordering between the sizes of aggregates and their parents. More specifically, for all aggregates  $v, w$  satisfying  $w = \text{parent}(v)$ , and  $k = |\text{children}(v)|$ ,

$$\frac{|v|}{|w|} \leq \frac{1}{1+k} \quad \text{when } |w| \neq |\mathcal{D}|$$

where  $\mathcal{D}$  is the database. We call such lattices *Size Restricted*, or SR-hypercube lattices. Intuitively, this means that if  $|\text{children}(w)| = 5$ , then the  $|w| \geq 6|v|$ . Though this appears to be a rather strong condition to impose on the structure of a lattice, it is in fact quite likely to be satisfied. For example, the hypercube lattice in Figure 2 is size restricted. To verify that a hypercube lattice is size restricted, one can use an aggregate size estimator which guarantees that the error in the estimate is bounded. One such estimator based on probabilistic counting is presented in [SDNR96]. Once the size estimates have been obtained, it is a simple matter to perform comparisons between aggregate sizes and determine whether the lattice is size restricted.

#### 3.3 PBS on SR-Hypercube Lattices

In this section, we prove a bound on the benefit of the set of aggregates picked by PBS in relation to the optimal set for SR-hypercube lattices. First we prove a bound for a subset of SR-hypercube lattices. Then we extend it to all SR-hypercube lattices.

**Theorem 3.1.** *Consider a SR-hypercube lattice where  $\forall w \in \mathcal{L}, w \neq \mathcal{D}, |w| \neq |\mathcal{D}|$ . The ratio of the benefits of PBS and optimal is at least  $(0.63 - f)$ ;  $f$  is the ratio of the largest aggregate size to the amount of space available for precomputation. Intuitively, this is the subset of SR-hypercube lattices in which aggregate sizes do not “saturate” and become equal to the database size.*

*Proof.* Harinarayan *et al.* [HRU96] prove that if  $f$  is the ratio of the size of the largest view to the size of the database, then BPUS will pick a set of aggregate views whose benefit is no less than  $(0.63 - f)$  of the optimal benefit. We prove the bound on PBS by showing that PBS and BPUS pick the same set of aggregates. The proof is by induction on the number of aggregates picked.

Consider a  $d$  dimensional SR-hypercube lattice where the ratio of the size of a view and its parent is  $\leq 1/(1+k)$ . The level number of an aggregate in the lattice is defined as the number of attributes it is aggregated on, and two aggregates with the same level number are said to be at the same level. The first aggregate picked by PBS is “ALL”, the smallest view in the lattice. As the basis of induction, let us examine the first aggregate view picked by BPUS. If  $i$  is the level number, and  $\ell_i$  is the ratio of the size of a view and the largest of its children, then the number of children that

Table 1: The initial benefits per unit space of views in a  $d$  dimensional SR-hypercube lattice.

Level	View size	# ben. views	Benefit per unit space
0	$= s$	$2^0$	$= \frac{ \mathcal{D}  - s}{s} \cdot 2^0$
1	$\geq sl_1$	$2^1$	$\leq \frac{ \mathcal{D}  - sl_1}{sl_1} \cdot 2^1$
2	$\geq sl_1 l_2$	$2^2$	$\leq \frac{ \mathcal{D}  - sl_1 l_2}{sl_1 l_2} \cdot 2^2$
$\vdots$			
$d-1$	$\geq s \prod_{i=1}^{d-1} l_i$	$2^{d-1}$	$\leq \frac{ \mathcal{D}  - s \prod_{i=1}^{d-1} l_i}{s \prod_{i=1}^{d-1} l_i} \cdot 2^{d-1}$

a node at level  $i$  has is greater than or equal to  $i$ . Therefore,  $l_i \geq 1 + i$ . When no views are selected for precomputation, the benefits per unit space of views are listed in Table 1. Assuming that  $\mathcal{D} \gg s \prod_{i=1}^{d-1} l_i$ ,  $0 < k < d$ , the maximum values of the benefits per unit space are:

$$\frac{\mathcal{D}}{s}, \frac{2^1}{l_1}, \frac{\mathcal{D}}{s}, \frac{2^2}{l_1 l_2}, \frac{\mathcal{D}}{s}, \dots, \frac{2^{d-1}}{\prod_{i=1}^{d-1} l_i}, \frac{\mathcal{D}}{s}$$

Since  $l_i \geq 2$ , the smallest view has the highest benefit per unit space initially, and is picked by BPUS. Hence, the first view picked by both PBS and BPUS is the smallest in size.

Let us assume that aggregates are picked by size until  $r$  have been picked. The  $(r+1)^{\text{th}}$  aggregate picked by PBS is the smallest in size among the remaining unpicked aggregates. Let us examine the structure of the lattice and find the  $(r+1)^{\text{th}}$  view that will be picked by BPUS. Since aggregate views have been picked by size so far, and the size of an aggregate  $v$  is greater than the size of any of its children, children of  $v$  are picked before  $v$ . This results in a *precomputation frontier* (Figure 4). Aggregate views on this frontier have all their children already selected for precomputation.

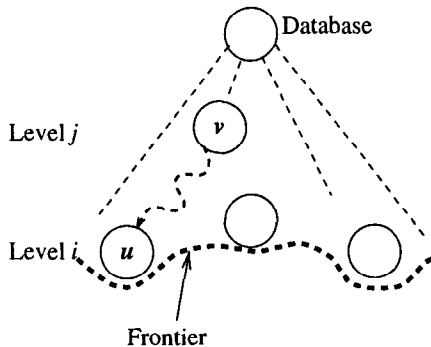


Figure 4: Shows the *precomputation frontier* for an SR-hypercube lattice.

We now show that only views on the frontier are considered at this point by BPUS for precomputation because views inside the frontier have a smaller benefit per unit space than some view on the frontier. Consider a view  $v$  at level  $j$  in the lattice. It has a descendant  $u$  at level  $i$  in the lattice, such that  $u$  is on the frontier

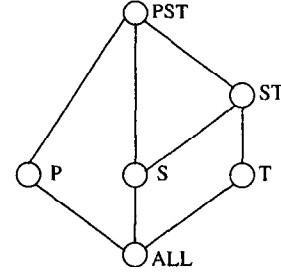


Figure 5: The example lattice from Section 1.1, where aggregates PS and PT have been pruned since they are equal in size to the database.

(Figure 4). The benefit per unit space of  $v$  is

$$\leq \frac{|\mathcal{D}|}{|u|} \cdot \frac{j(j-1)\dots(i+1)+1}{(i+1+1)(i+2+1)\dots(j+1)} \leq \frac{|\mathcal{D}|}{|u|}$$

since the maximum cardinality of  $|\text{descendants}(v)|$  is  $(j(j-1)\dots(i+1)+1)$ , and the size of  $v$  is at least  $|u| \cdot (i+1+1)(i+2+1)\dots(j+1)$ . The benefit per unit space of  $u$  is  $|\mathcal{D}|/|u|$ . Since  $u$  has a higher benefit per unit space than  $v$ , BPUS will not pick  $v$  at this point. Thus, only views on the frontier can be considered for precomputation at any point. To show that the next aggregate picked will be the smallest remaining unpicked aggregate, we have to show that among views on the frontier, the view with the smallest size will now be picked. A view  $u$  on the frontier has no children that have not yet been selected for precomputation. The benefit per unit space of  $u$  is  $(|\mathcal{D}| - |u|)/|u|$ . This means that the view with the smallest size has the largest benefit per unit space, and will be chosen by BPUS. PBS will also pick the view with the smallest size. Since BPUS and PBS pick the same set of aggregates, the benefit per unit space of the set of aggregates picked by PBS will be at least  $(0.63 - f)$  times the optimal benefit.  $\square$

Now we extend the above proof to prove a bound for all SR-hypercube lattices. In particular, we consider the case when the size ratio is satisfied until some level in the lattice and then aggregate views have the same size as the database  $\mathcal{D}$ .

**Theorem 3.2.** *PBS is within  $(0.63 - f)$  for a SR-hypercube lattice.*

*Proof.* Consider an SR-hypercube lattice. All aggregate views which have the same size as the database ( $|w|/|v| = 1$ ) have zero benefit ( $|\mathcal{D}| - |v| = 0$ ). This means that any query which can be answered by scanning  $v$  or  $w$  can be answered at equal cost by scanning the database. Therefore, we can prune the hypercube lattice so that such aggregates are removed. Figure 5 shows the pruned lattice corresponding to the lattice in Figure 2. In the new lattice, the condition stated in Theorem 3.1 is true. That is,  $w = \text{parent}(v)$ ;  $|v|/|w| \leq 1/(1+k)$ , where  $k = |\text{children}(w)|$ . From Theorem 3.1 it follows that the benefit of PBS is no less than  $(0.63 - f)$  of the optimal benefit.  $\square$

PBS may not achieve this bound for non SR-hypercube lattices. This is shown with an example in [SDN98].

### 3.4 Non-uniform Query Distributions

PBS assumed that all aggregates have an equal probability of being queried. We propose a variation of PBS, called PBS-U, in which a user can assign probabilities to aggregates. Domain specific knowledge of the schema and workload can be used to assign higher probabilities of being queried to *some* aggregates. We extend PBS so that the user can associate each aggregate with a probability of being queried, which corresponds to the frequency with which the aggregate is expected to be queried.

**Definition 3.1.** *The probability weighted size of an aggregate is equal to the ratio of the size of the aggregate to the probability of its occurrence.*

PBS-U picks aggregates in order of their probability weighted size:  $|v|/p_v$ , where  $|v|$  is the size of aggregate view  $v$ , and  $p_v$  is the frequency with which  $v$  is queried. Aggregates which are more frequently queried have their probability weighted size reduced, increasing their likelihood of being picked. PBS-U takes as input: space – the amount of space available for precomputation and  $\mathcal{A}$ , a set initially containing all views in the lattice. The output is  $\mathcal{S}$ , the set of views to be materialized;  $\text{wt-smallest}(\mathcal{A})$  is a function which returns the view having the smallest probability weighted size in set  $\mathcal{A}$ .

---

#### Algorithm PBS-U

```

WHILE (space > 0) DO
   $w = \text{wt-smallest}(\mathcal{A})$ 
  IF (space -  $|w| > 0$ ) THEN
    space = space -  $|w|$ 
     $\mathcal{S} = \mathcal{S} \cup w$ 
     $\mathcal{A} = \mathcal{A} - w$ 
  ELSE
    space = 0
S is the set of aggregates picked by PBS-U

```

---

If the lattice formed by considering probability weighted sizes of aggregates instead of their sizes is an SR-hypercube lattice, then it follows from Theorem 3.2 that PBS-U has the same bound as PBS. The intuition is that with probabilities of occurrence, the benefit per unit space of  $u$  is:  $p_u \cdot \frac{\text{benefit}}{|u|} = \frac{\text{benefit}}{|u|/p_u}$ . Considering the lattice as size restricted on  $|u|/p_u$  instead of just  $|u|$ , a similar proof as Theorem 3.2 holds.

## 4 Chunk Based Precomputation

### 4.1 Introduction to Chunks

The idea of chunks for dynamic query caching first appeared in [DRSN98]. In this paper we adapt it for static precomputation. The idea of chunks was motivated by *MOLAP* (Multidimensional OLAP) systems, which use multi-dimensional arrays to represent data. Instead of storing a large array in simple row major or column major order, they are broken down into chunks and stored in a chunked format [SS94, ZDN97]. The distinct values for each dimension are divided into ranges and the chunks are created based on this division. Figure 6

shows how the multidimensional space can be broken up into chunks. Our observation was that chunks are very suitable as a unit of precomputation. Chunks capture the notion of semantic regions and divide the entire space into uniform semantic regions. In a chunk-based

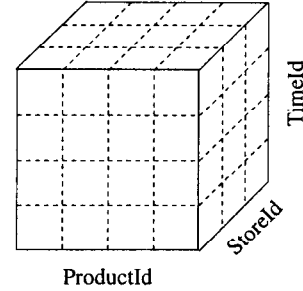


Figure 6: Chunking the Multidimensional Space for the schema from Section 1.1.

precomputation scheme, aggregates are broken up into chunks, and a chunk becomes the unit of precomputation. If the database supports multidimensional arrays, a chunk based aggregate can be implemented as a multi-dimensional array. On the other hand, if the system is fully relational, a chunked file organization as described by [DRSN98] can be used. An additional advantage of chunk based precomputation is an efficient implementation of the chunk based dynamic caching scheme proposed in [DRSN98].

When a query is asked, the set of chunks needed to answer it are determined. Then depending on what has been selected for precomputation, this set of chunks is divided into two partitions. The first partition consists of chunks that are precomputed, and the results just have to be looked up. The second partition consists of chunks that have to be computed from other aggregate chunks. For example, in Figure 7, a query could ask for chunks 0 and 1 of {ProductId}. If chunk 0 of {ProductId} is precomputed, then the first partition consists of chunk 0 of {ProductId}, while the second partition consists of chunk 1 of {ProductId}. Chunk 1 of {ProductId} has to be computed from other aggregate chunks (ancestors of {ProductId} in the lattice). From Figure 7, we see that chunk 1 of {ProductId} can be obtained by aggregating chunks 1, 5, 9, 13 of {ProductId, StoreId}. By using either a multidimensional array or a chunked file representation, direct access to these chunks of {ProductId, StoreId} is possible. This enables us to compute only missing chunks to determine the result of a query, rather than computing the entire query result.

### 4.2 The Motivation for using Chunks

Let us look at some examples to see how the ability to precompute parts of aggregates influences the average query cost and benefit per unit space. Figure 8 shows the subset of the lattice describing the hypercube for the Sales schema restricted to vertices with ProductId or StoreId.

**Example 4.1.** *Consider the hypercube lattice of Figure 8. Figure 9 shows a graph with the amount of space available for precomputation on the x-axis, and the av-*

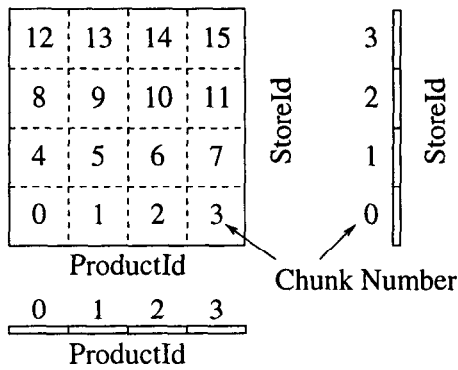


Figure 7: Chunks at different levels. The grid on the left corresponds to the aggregate (ProductId, StoreId).

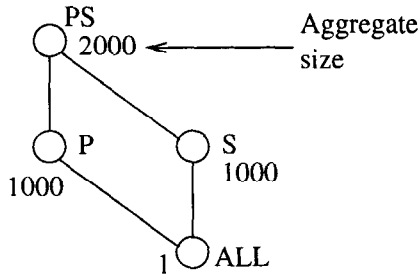


Figure 8: Data cube of {ProductId, StoreId}.

erage query cost on the y-axis. If chunk based precomputation is used, one can obtain a much lower average query cost as compared to BPUS or PBS.

The next example illustrates how the use of chunks can yield a better benefit per unit space than the optimal precomputation of whole aggregates.

**Example 4.2.** Consider the hypercube lattice of Figure 8. If the amount of space available for precomputation in 501 tuples, the optimal algorithm which picks whole group bys will be able to fit only ALL, resulting in a total benefit of 1999. If we use chunks, with a 25 tuples per chunk, We can fit ALL as well as 500 tuples (20 chunks) of P. This results in a benefit of 2499, which is larger than the optimal benefit when picking whole aggregates.

From the above examples we can see that the use of

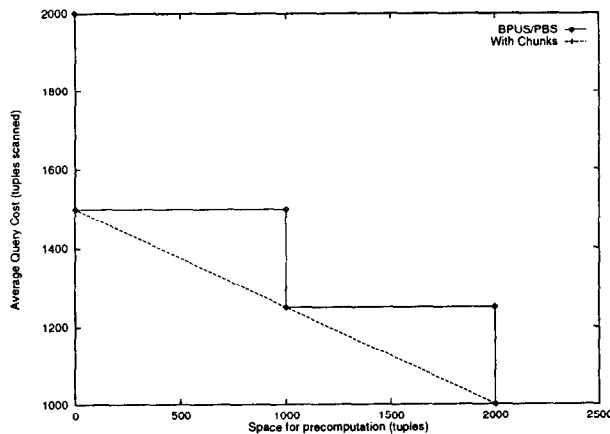


Figure 9: Graph corresponding to the lattice in Figure 8

chunking to precompute subsets of aggregates makes it possible for one to design algorithms with a lower average query cost than algorithms which assume that the granularity of precomputation is whole aggregates.

### 4.3 Cost Model for Chunks

In this section we look at the benefit per unit space of a chunk of an aggregate. Since chunk based precomputation can be thought of a semantic index on the data, the linear cost model applies to chunks. Assigning probabilities to chunks results in a huge increase in the size of the lattice and renders all proposed algorithms expensive and inefficient. Therefore, we assume that all chunks of an aggregate have the same probability of being queried as the aggregate. We start with two examples that examine the benefit per unit space of a chunk.

**Example 4.3.** In this example we examine how precomputing a chunk can have some benefit. Assume in the lattice of Figure 2 that all chunks of PS are precomputed. To answer a query on ALL, we have to compute chunk 0 of ALL. This can be done by scanning chunks 0-15 of PS (Figure 7), which is cheaper than scanning the database PST. Let us analyze what happens when chunk 0 of S (which is a descendant of PS) is precomputed. Chunk 0 of S and chunks 4-15 of PS can be aggregated to compute ALL. Hence precomputing a chunk of S saves us the scan of chunks 0-3 of PS.

**Example 4.4.** Consider the lattice of Figure 2 and assume that all chunks of S are precomputed and nothing else is precomputed. To answer a query on ALL, we have to scan chunks 0-3 of S (Figure 7). In this scenario, let us analyze the benefit of a chunk of P. Since S is not an ancestor of P in the lattice, P doesn't benefit the computation of ALL even if the size of P is smaller than the size of S.

The observations of the above examples are summarized in the following lemmas.

**Lemma 4.1.** Let  $S$  be the set of aggregates selected for precomputation. If  $w \in S$ ,  $w = L(v)$  and  $v \prec u \prec w$  then the computation of a chunk of  $u$  benefits  $v$ .

*Proof.* Since  $v \prec u \prec w$ , precomputing a chunk of  $u$  saves a scan of the corresponding set of chunks of  $w$  in answering a query on  $v$ .  $\square$

**Lemma 4.2.** Let  $S$  be the set of aggregates selected for precomputation. If  $w \in S$ ,  $w = L(v)$  and  $u \not\prec w$  then the computation of a chunk of  $u$  does not benefit  $v$ .

*Proof.* Since  $u \not\prec w$ , precomputing a chunk of  $u$  does not benefit a query on  $v$  being answered from  $w$ .  $\square$

We now formalize the notion of the benefit per unit space of a chunk in the following theorem.

**Theorem 4.1.** For a hypercube lattice with a set  $S$  of aggregates precomputed,  $B_s(u_c, S) \leq B_s(u, S)$ , where  $u \in S$ ,  $u_c$  is a chunk of  $u$ . In other words, the benefit per unit space of a chunk of  $u$  is  $\leq$  the benefit per unit space of  $u$ .



*Proof.* We have seen in Lemmas 4.1 and 4.2 that depending on the aggregate used to answer a query, a chunk can have a zero or non-zero benefit. A chunk of  $u$  benefits an aggregate  $v$  only when  $v \prec u$ ,  $u \prec L(v)$ ,  $\mathcal{C}(L(v)) > \mathcal{C}(u)$ . Let us assume that this holds and let  $w = L(v)$ , then the benefit per unit space of  $u$  to  $v$  is

$$\frac{1}{|u|} \cdot (\mathcal{C}(w) - \mathcal{C}(u)) \quad (3)$$

Let us compute the benefit per unit space of a chunk  $u_c$ .

$$\frac{1}{|u_c|} \cdot (\mathcal{C}(w') - \mathcal{C}(u_c)) \quad (4)$$

where  $w'$  is the set of chunks of  $w$  which don't have been scanned to compute  $v$  if  $u_c$  is precomputed. Let  $\mathcal{N}(u)$ ,  $\mathcal{N}(w)$  be the number of chunks of  $u$  and  $w$  respectively. Then,

$$|u_c| = \mathcal{C}(u_c) = \frac{\mathcal{C}(u)}{\mathcal{N}(u)} \text{ and } |w'| = \mathcal{C}(w') = \frac{\mathcal{N}(w)}{\mathcal{N}(u)} \cdot \frac{\mathcal{C}(w)}{\mathcal{N}(w)}$$

from the chunk decomposition of aggregates. Using this, Equation 4 becomes

$$\frac{1}{|u|} \cdot (\mathcal{C}(w) - \mathcal{C}(u)) \quad (5)$$

This is the same as the benefit per unit space of  $u$  to  $v$  (Equation 3). We can now rewrite the definition of benefit per unit space (Definition 2.3) to account for chunks:

$$\mathcal{B}_s(u_c, \mathcal{S}) = \frac{1}{|u|} \sum (\mathcal{C}(L(v)) - \mathcal{C}(u)), \quad (6)$$

$v \prec u$ ,  $\mathcal{C}(L(v)) > \mathcal{C}(u)$ ,  $u \prec L(v)$ . The difference between Definition 2.3 and Equation 6 is in the additional condition  $u \prec L(v)$  which says that  $L(v)$  should be an ancestor of  $u$  for a chunk to have a non-zero benefit per unit space. Therefore, the benefit per unit space of a chunk of an aggregate is less than or equal to the benefit of the entire chunk.  $\square$

#### 4.4 Algorithms using Chunk based Precomputation

At a first glance, it looks like the ability to precompute subsets of aggregates makes the precomputation problem even harder. Not only do aggregates have to be selected, a decision of which subset of the aggregate to precompute also has to be made. We demonstrate that this ability has not made the problem harder, and show how BPUS and PBS can be modified to work in this situation.

When aggregates are decomposable into chunks, the resulting lattice is an AND-OR View Graph as defined by Gupta [Gupt97]. Gupta proposes algorithms for AND-OR view graphs that are exponential. Since the number of chunks can be very large (depending on the data), exponential algorithms are clearly unpractical. We design polynomial algorithms that are based on BPUS and PBS.

We proved in Theorem 4.1 that the benefit per unit space of a chunk of an aggregate  $v$  is less than or equal to the benefit per unit space of  $v$ . Since BPUS is a greedy algorithm, picking the aggregate with the maximum benefit per unit space at each step, it will pick whole aggregates rather than chunked subsets of an aggregate. We use this observation to define BPUS-C, which uses chunks to improve the benefit per unit space of a precomputation. The inputs to BPUS-C are: space – the amount of space available for precomputation, and  $\mathcal{A}$ , a set initially containing all aggregates in the lattice, except the database. The output is  $\mathcal{S}$ , the set of aggregates to be precomputed.

#### Algorithm BPUS-C

Run Algorithm BPUS on the lattice

$w =$  aggregate with max. benefit per unit space in  $\mathcal{A}$

Add  $k$  chunks of  $w$  to  $\mathcal{S}$ ,  $k \cdot |w_c| \leq$  available space

$\mathcal{S}$  is the set of aggregates picked by BPUS-C

For a general hypercube lattice, it is hard to quantify the exact benefit of  $\mathcal{S}$ . BPUS has a  $(0.63 - f)$  bound and BPUS-C picks a subset of an aggregate not chosen by BPUS, in addition to picking the whole aggregates chosen by BPUS. Therefore, the benefit of  $\mathcal{S}$  produced by BPUS-C is greater than or equal to that of BPUS. To obtain the  $(0.63 - f)$  performance bound for SR-hypercube lattices, PBS-C has to duplicate the operations of PBS. PBS-C takes the same inputs as BPUS-C and produces the same output as BPUS-C. PBS-C differs from BPUS-C in the first step. Instead of running BPUS on the lattice, PBS-C runs PBS on the lattice. For SR-hypercube lattices, we can obtain a tighter bound on the benefit with respect to the benefit of the optimal precomputation.

**Theorem 4.2.** *For a SR-hypercube lattice, let  $f'$  be the ratio of the size of the largest chunk size and the space available for precomputation. Then the benefit of PBS-C and BPUS-C is  $(0.63 - f')$  times the optimal benefit.*

*Proof.* When BPUS or PBS is run on a SR-hypercube lattice, the cheapest way to precompute an aggregate is either from itself (if it has been selected for precomputation), or from the database,  $\mathcal{D}$  (if it hasn't been selected). Then the following holds:

$$v \prec u \wedge \mathcal{C}(L(v)) > \mathcal{C}(u) \Rightarrow \mathcal{D} = L(v) \wedge u \prec L(v)$$

Examining Equation 6, this means that the benefit per unit space of a chunk of  $u$  is *always* equal to the benefit per unit space of  $u$ . In this situation, we can still pick whole aggregates, and when a whole aggregate doesn't fit, we can precompute as many chunks as possible of the aggregate with largest benefit per unit space. Since  $u_c$  and  $u$  have equal benefit per unit space, the performance bound can be quantified. The  $f$ , the ratio of the size of the largest aggregate to the space becomes  $f'$ , the ratio of the size of the largest chunk size to the space. PBS-C and BPUS-C will pick the same aggregates for a SR-hypercube lattice. Therefore, the benefit of aggregates picked by both PBS-C and BPUS-C is at least  $(0.63 - f')$  times the optimal benefit.  $\square$

In summary, the ability to precompute chunk based subsets of aggregates improves the benefit of a set of aggregates precomputed with respect to the optimal set (of whole aggregates) for a given amount of space. For example, if the largest aggregate has a size of 50MB, the unit of precomputation has a size of 4KB, and the space available for precomputation is 200MB, then  $f = 50/200 = 0.25$ . So the bound of PBS and BPUS is  $(0.63 - 0.25) = 0.38$ . On the other hand, PBS-C and BPUS-C modified to use aggregate subset caching have a bound of  $(0.63 - 4/204800) = 0.62998$ , or almost 0.63 for a SR-hypercube lattice.

## 5 Experimental Evaluation

In our experiments, we study the average query cost of a precomputation as the amount of space available for precomputation is increased. We performed experiments on datasets generated from real-life data distributions, and synthetic data. We assume that queries on any aggregate are equally likely, and use analytical formulas presented in [SDNR96, RS97] to estimate the size of aggregates formed by the data cube operator. For example, consider a relation R having attributes A, B, C and D. Suppose we want to estimate the size of the group by on attributes A and B. If the number of distinct values of A is  $n_A$  and that of B is  $n_B$ , then the number of elements in  $A \times B$  is  $n_e = n_A n_B$ . Let  $|\mathcal{D}|$  be the number of tuples in the database. Using these values and an assumption that tuples are uniformly distributed, the number of elements in the group by on A and B is:  $n_e - n_e(1 - 1/n_e)^{|\mathcal{D}|}$ . This is similar to relational group by size estimation.

### 5.1 Experiments on distributions found in Real-datasets

We ran experiments on datasets generated from data distributions found in four real-life datasets. The data distributions appeared in [AAD+96]. They are derived from sales transactions of department stores and mail-order companies. The number that appears next to an attribute represents the number of distinct values. We now describe the datasets.

**Dataset R1** Contains data with 5.5 million tuples, for a mail order company. Each transaction has four attributes: customer id (213972), order date (2589), product id (15836), and the catalog used for ordering (214).

**Dataset R2** Contains data with 7.5 million tuples, describing grocery purchases of customers from a supermarket. There are five attributes in each transaction: date of purchase (1092), shopper type (195), store code (415), the state in which the store is located (46), the product group of the item purchased (118).

**Dataset R3** This is data with 9 million tuples from a department store. Each transaction has five attributes: the store id (17), date of purchase (15), the UPC of the product (85161), the department number (44), and the SKU number (63895).

**Dataset R4** Contains data with 3 million tuples from a department store. Each transaction has a total

of six attributes: the store number (4), the date of purchase (15), item number (26412), the business center (6), the merchandising group (22496), and a sequence number (255).

Using the dimension sizes in datasets R1, R2 and R3, we estimated aggregate sizes and found that the resulting lattice is size restricted. Figures 10, 11, 12 illustrate how the average query cost varies with the space available for precomputation. The data cubes for all three schemas are SR-hypercube lattices, hence PBS and BPUS pick the same set of aggregates. PBS-C has a lower average query cost than PBS because it picks a subset of an aggregate when PBS and BPUS cannot pick any aggregate (Figure 13). Figure 11 has been truncated along the x-axis to show the effect of chunk based precomputation in greater detail.

Dataset R4 has 6 dimensions, and one of the dimensions (store number) has only 4 distinct values. This means the data cube of R4 is not a SR-hypercube lattice. In this case, we plot the relative error of PBS and PBS-C with respect to BPUS. If  $AQC(PBS)$  is the average query cost of the set of aggregates picked by PBS, then the relative error between PBS and BPUS is:  $(AQC(PBS) - AQC(BPUS)) / AQC(BPUS)$ . Figure 14 shows the relative errors of PBS and PBS-C with respect to BPUS. The error of PBS is quite small ( $\leq 0.08\%$ ). PBS-C does better than BPUS because it has the ability to precompute subsets of an aggregate.

### 5.2 Synthetic datasets

Real datasets do not give us the flexibility to vary the increase in size between an aggregate and its parent. So, we ran some experiments to study how the average query cost of the set of aggregates picked by PBS varies with respect to the optimal set as the lattice deviates from the size restrictions. The dataset had four dimensions, and 1.2 million tuples. In the initial configuration, aggregates formed a SR-hypercube lattice. Then, we increased the fraction between an aggregate and its parent by small increments of 0.20. For example, if  $u = \text{parent}(v)$ , and  $|v|/|u| = \ell$ , then  $\ell$  is successively increased to  $\ell + .10$ ,  $\ell + .30$ ,  $\ell + .50$ . Figure 15 shows the results of this experiment. The relative error of PBS-C with respect to optimal is  $AQC(PBS-C) - AQC(OPT) / AQC(OPT)$ . Initially, when the offset is increased, the error increases, but then starts decreasing. A negative error means that PBS-C can do better than the optimal set resulting from picking full aggregates. We have compared PBS-C with the optimal algorithm which picks whole aggregates since optimally deciding which subsets of aggregates to pick in addition to which aggregate to pick is too computationally expensive. We found that the error of PBS-C doesn't exceed 30%. This makes it practically useful for general hypercube lattices.

## 6 Conclusions

Precomputing aggregates on some subsets of dimensions and their corresponding hierarchies can substantially reduce the response time of a query. However, the decision of what to precompute is not easy. Algorithms have been proposed to solve this problem, and

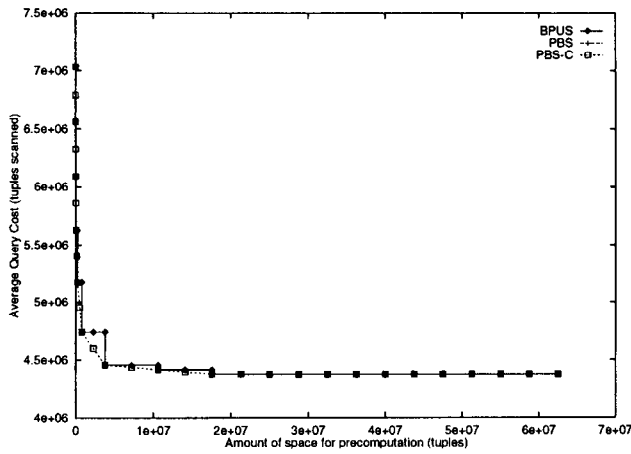


Figure 10: The average query cost as space is varied for dataset R1

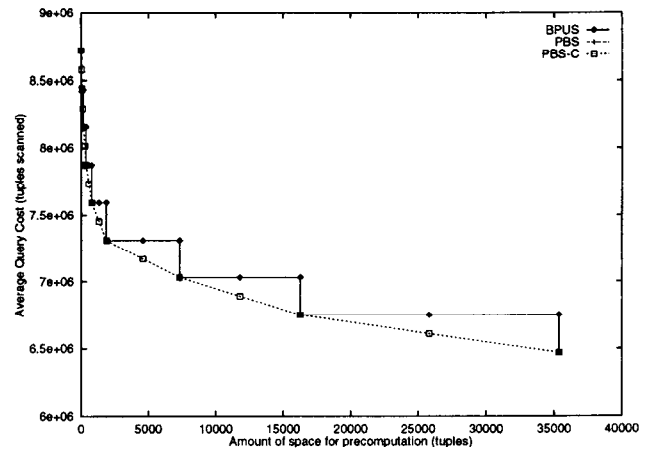


Figure 11: The average query cost as space is varied for dataset R2.

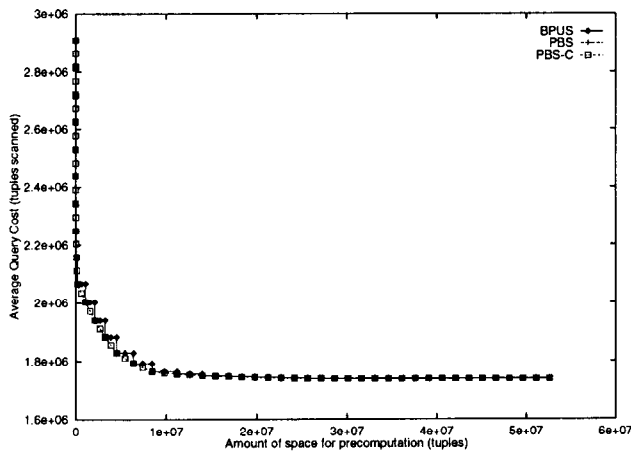


Figure 12: The average query cost as space is varied for dataset R3.

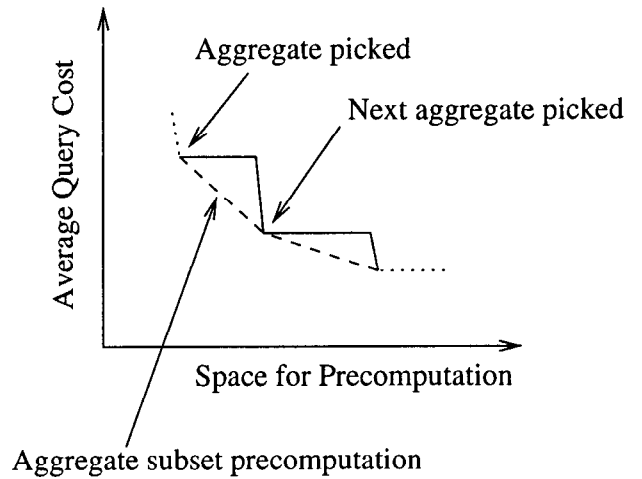


Figure 13: Illustrates when PBS-C is better than PBS as the amount of space is varied.

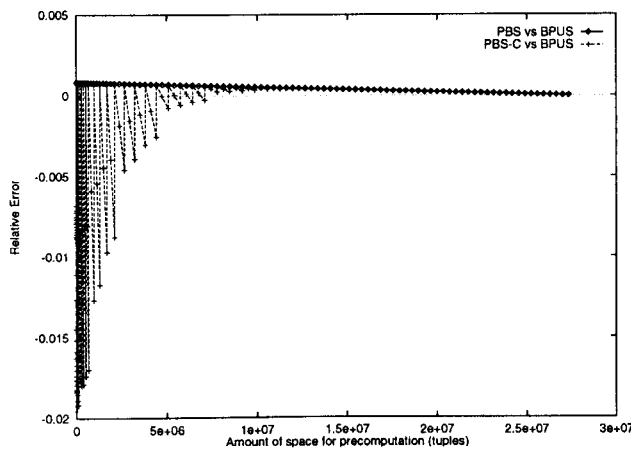


Figure 14: The relative error of PBS and PBS-C with respect to BPUS as space is varied for dataset R4.

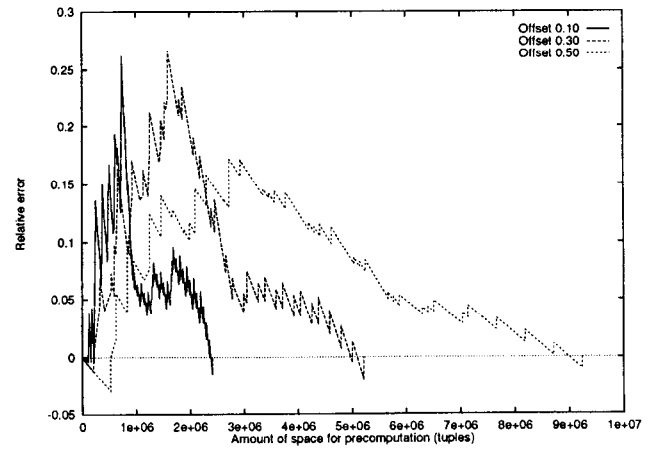


Figure 15: The relative for the synthetic dataset.

the most accurate algorithm, BPUS, guarantees that the set of aggregates it picks will not have a benefit worse than  $(0.63 - f)$  times optimal, where  $f$  is the fraction of available space used by the largest view.

Depending on the size of the lattice, BPUS could take from several days to several months to find a set of aggregates to be materialized. We designed a fast algorithm PBS with complexity  $O(n \log n)$ , where  $n$  is the number of vertices in the lattice. We proved that for SR-hypercube lattices, a broad class of hypercube lattices, PBS achieves  $(0.63 - f)$  times the optimal benefit. We showed that the execution times of BPUS and PBS can vary by several orders of magnitude, 64 days vs 0.37 seconds. We introduced chunk based precomputation and showed how using chunks for aggregate subset precomputation can make the benefit larger than the "optimal" benefit when picking whole aggregates. We designed a benefit based cost model for chunks, and extended BPUS and PBS to use chunk based precomputation. We showed that for SR-hypercube lattices, BPUS and PBS achieve a benefit that is not less than  $(0.63 - f')$  of optimal, where  $f'$  is the ratio of the largest chunk size to the space available for precomputation.

In the experimental evaluation, we performed experiments on data generated from distributions found in real-life, and synthetic datasets to see how the average query cost of aggregates picked by PBS and PBS-C varies with the amount of space available for precomputation. We found that the lattice embedded by three of the four real-life datasets is size restricted, corroborating the assertion that SR-hypercube lattices occur commonly. In the remaining real-life dataset, we showed that the difference in the average query cost of PBS-C relative to that of BPUS is small. To study the effect of changing the restriction on aggregate sizes, we generated a 4 dimensional synthetic data and varied aggregate sizes to worsen the performance of PBS-C. We found that the average query cost of PBS-C was never worse than 30% of optimal.

To enable a DBA to determine how much space should be allocated for precomputation, it is useful to plot a graph of the average query cost vs. the space available for precomputation (Figure 10). Either PBS or PBS-C can be used for this purpose. To conclude, we discuss which algorithm is appropriate for a given lattice. The algorithm of choice will depend on the existence of chunks. If chunking is supported, then the chunked version, BPUS-C or PBS-C should be used. If chunk are not supported, then the non-chunked version can be used. If we assume support for chunks, then for a hypercube lattice, the only algorithm known to date to have a provable bound on the benefit is BPUS. A major drawback is the execution time of BPUS. A simple approach in this case is to use PBS-C. While PBS-C performed well in our empirical study, there is no known theoretical bound on the benefit of the set of aggregates picked by PBS-C for a general hypercube lattice. If it is important to find the set of views to be precomputed quickly, and the user can tolerate a potentially slightly higher average query cost, then PBS-C is ideal. For SR-hypercube lattices, PBS-C guarantees a  $(0.63 - f')$  error bound, and should be used.

## References

- [AAD+96] S. Agarwal, R. Agrawal, P.M. Deshpande, A. Gupta, J.F. Naughton, R. Ramakrishnan, S. Sarawagi. On the Computation of Multidimensional Aggregates, *Proc. of the 22nd Int. VLDB Conf.*, 506–521, 1996.
- [BPT97] E. Baralis, S. Paraboschi, E. Teniente. Materialized View Selection in a Multidimensional Database, *Proc. of the 23rd Int. VLDB Conf.*, 1997.
- [DRSN98] P.M. Deshpande, K. Ramasamy, A. Shukla, J.F. Naughton. Caching Multidimensional Queries Using Chunks. *Proc. ACM SIGMOD Int. Conf. on Man. of Data*, 1998.
- [GBLP96] J. Gray, A. Bosworth, A. Layman, H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals, *Proc. of the 12th Int. Conf. on Data Engg.*, pp 152-159, 1996.
- [GHRU97] H. Gupta, V. Harinarayan, A. Rajaraman, J.D. Ullman. Index Selection for OLAP. *Proc. of the 13th ICDE*, 208–219, 1997.
- [Gupt97] H. Gupta. Selection of Views to Materialize in a Data Warehouse. *Proc. of the Sixth ICDT*, 98–112, 1997.
- [HRU96] V. Harinarayan, A. Rajaraman, J.D. Ullman. Implementing Data Cubes Efficiently, *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 205–227, 1996.
- [RS97] K.A. Ross, D. Srivastava. Fast Computation of Sparse Datacubes. *Proc. of the 23rd Int. VLDB Conf.*, 116–125, 1997.
- [SS94] S. Sarawagi and M. Stonebraker. Efficient Organization of Large Multidimensional Arrays. *Proc. of the 11th Int. Conf. on Data Engg.*, 1994.
- [SDNR96] A. Shukla, P.M. Deshpande, J.F. Naughton, K. Ramasamy, Storage Estimation for Multidimensional Aggregates in the Presence of Hierarchies, *Proc. of the 22nd Int. VLDB Conf.*, 522–531, 1996.
- [SDN98] A. Shukla, P.M. Deshpande, J.F. Naughton. Materialized View Selection for Multidimensional Datasets. Available from [www.cs.wisc.edu/~samit](http://www.cs.wisc.edu/~samit).
- [Ull96] J.D. Ullman, Efficient Implementation of Data Cubes Via Materialized Views A survey of the field for the 1996 KDD conference.
- [ZDN97] Y. Zhao, P.M. Deshpande, J.F. Naughton. An Array-Based Algorithm for Simultaneous Multidimensional Aggregates, *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 159–170, 1997.