Materialized View Selection for Multi-cube Data Models

Amit Shukla, Prasad M. Deshpande, and Jeffrey F. Naughton

{amit,pmd,naughton}@cs.wisc.edu
University of Wisconsin - Madison, Madison WI 53706, USA

Abstract. OLAP applications use precomputation of aggregate data to improve query response time. While this problem has been well-studied in the recent database literature, to our knowledge all previous work has focussed on the special case in which all aggregates are computed from a single cube (in a star schema, this corresponds to there being a single fact table). This is unfortunate, because many real world applications require aggregates over multiple fact tables. In this paper, we attempt to fill this lack of discussion about the issues arising in multi-cube data models by analyzing these issues. Then we examine performance issues by studying the precomputation problem for multi-cube systems. We show that this problem is significantly more complex than the single cube precomputation problem, and that algorithms and cost models developed for single cube precomputation must be extended to deal well with the multi-cube case. Our results from a prototype implementation show that for multi-cube workloads substantial performance improvements can be realized by using the multi-cube algorithms.

1 Introduction

Online Analytical Processing (OLAP) systems use the *multidimensional model*, which expands the row and column approach of the relational model into multiple categories of data called dimensions. Dimensions such as time, product, line item, and geography categorize and summarize facts, like unit sales. We refer to this summarization as *aggregation*. Array arithmetic can be used to efficiently access cells and slices of the data. This results in support for complex analytical queries and applications. Other functionality supported by OLAP servers include functions that analyze, forecast, model, and answer "what if" questions about the data. They also have built-in functions for mathematical, financial, statistical, and time-series manipulation.

OLAP systems require fast interactive multidimensional data analysis of aggregates. To fulfill this requirement, database systems frequently statically precompute aggregate views on some subset of dimensions and their corresponding hierarchies. Virtually all OLAP products resort to some degree of precomputation of these aggregates. In order to understand the issues involved in precomputation, let us first look at how multidimensional data providers structure their data model. The most common approach used by OLAP products is called the *multi-cube* structure [8]. In this approach, the application designer segments the database into a set of multidimensional structures each of which is composed of a subset of the overall number of dimensions in the database. Products such as Oracle Express, Microstrategy DSS Suite, Informix Metacube and Microsoft OLAP services

C. Zaniolo et al. (Eds.): EDBT 2000, LNCS 1777, pp. 269–284, 2000.

all use the multi-cube approach. Unfortunately, to date, the research community has virtually ignored the precomputation problem over multi-cubes, concentrating instead on the simpler single–cube model. To our knowledge, this is the first paper to address the precomputation problem in the context of multi-cube domains. Thus, queries which access multiple cubes are not taken into account when making a decision of what group bys to precompute.

The goal of this paper is twofold. First, to understand the ramifications of having a multi-cube data model, and second to understand the precomputation problem when there are queries which access multiple cubes. We examine existing techniques, and propose new techniques which can be utilized to solve this problem. We show that the multi-cube aggregate selection problem is significantly more complex than the single cube computation problem, and that algorithms and cost models developed for single cube precomputation must be extended to deal well with the multi-cube case. Our results show that for multi-cube workloads substantial performance improvements can be realized by using multi-cube algorithms instead of previously proposed single cube algorithms.

1.1 An Example Schema

In this paper, for clarity of exposition we will assume a Relational approach to OLAP. This means that each "cube" of a multi-cube model corresponds to a fact table. However, the material presented in this paper is not restricted to the relation model.

Consider a schema which consists of three dimensions, `CustID`, `ProdID`, and `TimeID`. They identify a customer, a product, and the time (in months). The schema has three fact tables. The first is `Sales`, a row of which captures the dollar sales and unit sales of a particular product by a certain store in some month. The second fact table is `ProdCost`, and it captures the cost of products on a month by month basis. The third table captures the shipping cost of products to various customers, and is called `ShipCost`.

`Sales`(ProdID, CustID, TimeID, Sales, UnitsSold)
`ProdCost`(ProdID, TimeID, PCost)
`ShipCost`(CustID, TimeID, SCost)

A user can specify "derived" or computed metrics, which are formed as a combination of other metrics. If the component metrics belong to different fact tables, then a join is required to generate the derived metric. For example, a "derived metric" called *Profit* = (Sales - UnitsSold * (PCost + SCost)), which is obtained from the natural join of the ProdCost and Sales tables along the ProdID, TimeID dimensions. Any queries which involve the Profit metric will require a join to be performed, unless the join is precomputed. For example, the following query requires a join:

```
SELECT Sales.CustID, SUM(Sales - UnitsSold * (PCost + SCost))
FROM ProdCost, ShipCost, Sales
WHERE ProdCost.ProdID = Sales.ProdID  AND ProdCost.TimeID = Sales.TimeID
AND ShipCost.CustID = Sales.CustID  AND ProdCost.TimeID = Sales.TimeID
GROUP BY Sales.CustID
```

1.2 Related Work

To find a set of aggregates to materialize, [7] proposes a greedy algorithm that attempts to maximize the benefit per unit space. They prove that if the largest aggregate view occupies a fraction f of the space available for precomputation,

then the aggregates picked by the greedy algorithm have a benefit at least $(0.63-f)$ times the benefit of the optimal set of views for the same amount of space. The greedy algorithm restricts itself to a single cube data model.

Other related work includes [10] where the authors explore efficient algorithms for aggregate selection for single cube schemas. In [5], the authors consider the selection of views and indexes together. [6] presents a theoretical framework for the view-selection problem, and proposes a general algorithm and several heuristics. [13] surveys techniques proposed for determining what aggregates should be precomputed. When lattices are so large that even scanning them once is expensive, a different approach to precomputation is needed. [2] examine lattices with 10^{13} aggregate views. For such lattices, they provide heuristics to determine what views should be precomputed based a set of views the user supplies. All these papers ([13,5,6,2]) examine the aggregate selection problem in the context of a single cube. Finally, Shukla discusses aggregate selection algorithms in detail in [11].

1.3 Paper Organization

In section 2 we describe the lattice framework and cost model for the aggregate selection problem. We also describe an existing aggregate selection algorithm. Section 3 discusses the issues that arise when joining multiple cubes and precomputing their joins. Section 4 presents greedy algorithms for aggregate selection for multi-cube schemas. 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 Previous Work on Precomputation

In this section we first discuss the lattice framework for multidimensional datasets. This framework was first proposed by Harinarayan et al. [7]. Next we present the cost model for single cube schemas proposed by [7], and used in subsequent research [13,5,2,10]. In order to handle multi-cube schemas, we extend the single cube cost model to account for join costs required to compute derived metrics. Shukla et al. [10] had proposed an average query cost metric which makes visualization of the "goodness" of the aggregate set selected for precomputation easier. We repeat a description of average query cost in Section 2.4.

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 group bys can be computed from the precomputed result of another group by. In the example Sales table of Section 1.1, the aggregate on (ProdID, CustID) can be used to answer a query on (ProdID). This relation between aggregate views can be used to place them within a lattice framework as proposed in [7]. 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 1 represents the cube of the schema described in Section 1.1. The three dimensions ProdID, CustID, TimeID are represented by P, C, T respectively, and an aggregate view is labeled using the names of the attributes it is aggregated on. For example, view PC is aggregated on attributes ProdID and CustID. In Figure 1, 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 PC and P. This means that PC can be used to compute P. If there is no precomputation, a query on P (ProdID) has to be answered using the base data, PCT (Sales table). When there are multiple cubes, we have a collection of lattices from which aggregates have to be picked for precomputation.

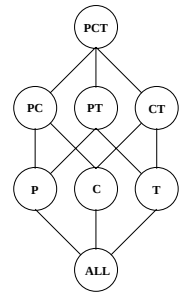

Fig. 1. The hypercube lattice corresponding to the example in Section 1.1.

2.2 The Cost Model

We use the cost model proposed by [7], 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. An experimental validation of this cost model is provided in [7], where the authors used experiments on TPC-D Benchmark data. 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 $\mathcal{S}$. This is similar to the metric used by [7].

We now formally define the benefit of an aggregate view. If $\mathcal{S}$ is a set of aggregates that have already been selected for precomputation, the *benefit* of a view v is concerned with how materializing v improves the cost of computing other views, including itself. Let $\mathcal{C}(v)$ be the cost of computing another view from v. Looking back to our cost model, $\mathcal{C}(v)$ is the number of tuples in v. The benefit of v with respect to the $\mathcal{S}$, $\mathcal{B}(v, \mathcal{S})$, is defined below.

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

In short, for each view u that is a descendant of v, we check to see if computing u from v is cheaper than computing u from any other view in the set $\mathcal{S}$. If this is the case, then precomputing v benefits u. Since all aggregates can be computed from the (unaggregated) base data, in step 1.1 we can always find a least cost aggregate view w (the base data in the worst case).

Definition 1. *The benefit per unit space of a view v is defined as:*

$$\mathcal{B}_s(v, \mathcal{S}) = \frac{\mathcal{B}(v, \mathcal{S})}{|v|}, \text{ where } |v| \text{ is the size of } v$$

2.4 Average Query Cost

Next, consider a set of lattices $\mathcal{L}$ with n views, $v_1, \ldots, v_n$. There are n different *templates* for queries, one for each view: $\mathcal{Q}_1, \mathcal{Q}_2, \ldots, \mathcal{Q}_n$. Let there be a set $\mathcal{S}$ of aggregate views precomputed, so that a query of view v_i can be most cheaply answered from a view u_i, $u_i \in \mathcal{S}$. Let queries on $\mathcal{S}$ occur with probabilities $p_1, p_2, \ldots, p_n$, then the *average query cost* is defined as:

$$\frac{1}{n} \cdot \sum_{i=1}^{n} p_i \mathcal{C}(u_i), \tag{1}$$

where $\mathcal{C}(u_i)$ is the cost of answering a query $\mathcal{Q}_i$ on a view v_i. In [10], we show that maximizing the benefit of a set $\mathcal{S}$ of aggregate views is the same as minimizing their average query cost.

2.5 Aggregate Selection

PickAggregates is a greedy algorithm proposed by Ullman etal. [7] to select aggregates for precomputation based on the above benefit model. The inputs to PickAggregates are: space – the amount of space available for precomputation, and $\mathcal{A}$, a set initially containing all aggregates in the lattice, except the base table. The output is $\mathcal{S}$, the set of aggregates to be precomputed. The algorithm is as follows:

Algorithm PickAggregates
WHILE (space > 0) **DO**
    w = aggregate having the maximum 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
    Update the benefit of affected nodes in the lattice
$\mathcal{S}$ is the set of aggregates picked by BPUS

The notion of benefit used by the above algorithm was described in Section 2.3. PickAggregates attempts to maximize the benefit of the set of aggregates picked. The authors 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.

3 Issues in Multi-cube Models

In order to discuss precomputation for multi-cube data models, it is necessary to define the semantics of queries over multi-cube models. For concreteness, in this section we discuss the issues that arise in queries over multi-cube models, and state the conventions we will follow when dealing with queries over multi-cubes. We start by looking at the semantics of multidimensional queries.

3.1 Multidimensional Query Semantics

The multidimensional model expands the row and column approach of the relational model into multiple categories of data called dimensions. Dimensions such as time, product, line item, and geography categorize and summarize facts, like unit sales. Users can ask analytical queries using a multidimensional query tool. A multidimensional query consists of the dimensions of interest (such as time, product), the fact to be summarized (like unit sales), and conditions (such as time equals january 1999). If the query accesses multiple cubes, then the data model implicitly specifies the procedure to join the two cubes. Therefore, in this paper we adopt standard multidimensional semantics [12], which state that one should join on all the dimensions common to the cubes being joined.

Definition 2. *The columns used to join two fact tables are called their "join dimensions". The join dimensions between two tables include all their common dimensions.*

For example, from the schema of section 1.1, the join dimensions between `Sales` and `ProdCost` are ProdID, TimeID. Based on this definition of join dimensions, we can define our notation for uniquely identifying a group by.

Definition 3. *Let D_i represent a dimension, F_j represent a distributive aggregation function such as sum, min, max, and M_k represent a metric. Then the results of the query*get $D_1, \ldots, D_n, F_1(M_1), \ldots, F_r(M_r)$, *can be represented compactly as* $(D_1, \ldots, D_n)_{F_1(M_1),\ldots,F_r(M_r)}$

For example using this definition, the query: get ProdID, CustID, SUM(Sales) will be represented by (ProdID, CustID)$_{\text{SUM(Sales)}}$. In order to simplify the notation, we assume that if the aggregation function is not specified, then it is SUM. Therefore, the above query can be rewritten as (ProdID, CustID)$_{\text{Sales}}$. A corollary of this assumption by multidimensional query tools is that multidimensional queries are restricted to a subset of SQL in that one cannot express different ways of joining two fact tables - one always has to join on all the common dimensions.

3.2 Precomputed Joins

Let us look at the benefits of precomputing aggregates that result from joining multiple cubes. Our goal is to investigate the approach of computing the join of all the cubes. Precomputing the join of all the cubes creates a single large table

that is in effect the "universal relation" [14], of which the multiple fact tables are projections. We scrutinize this approach to understand if it will enable us to apply single-table aggregate selection algorithms immediately. We show that this approach is fraught with hazards.

Consider the tables from the schema of Section 1.1. At a first glance, it seems as if we can extend the idea of precomputation of aggregates formed from a single fact table to precomputing aggregates formed from joins between fact tables. If we perform an *equi-join* between the two tables ShipCost and Sales,

```
SELECT Sales.CustID, Sales.ProdID, Sales.TimeID, SUM (Sales), SUM (SCost)
FROM Sales, ShipCost
WHERE Sales.CustID = ShipCost.CustID
AND Sales.TimeID = ShipCost.TimeID
GROUP BY Sales.CustID, Sales.ProdID, Sales.TimeID
```

to obtain a new table T_e with the schema T_e(CustID, ProdID, TimeID, S_Sales, S_SCost). This new table T_e can be precomputed and materialized in the database. However, T_e can be used only by queries which require a join of Sales and ShipCost. That is, T_e cannot be used to answer queries which access only one of the joined tables. This leads us to examine whether we can use full outer joins between tables to solve this problem.

When we perform the natural join between two tables ShipCost and Sales, only tuples from ShipCost that have matching tuples in Sales – and vice versa – appear in the result. Hence, tuples without a "related tuple" are eliminated from the result. *Outer joins* [3] were proposed to be used when one wants to retain all tuples from both tables being joined. In our example, an outer join between ShipCost and Sales would include tuples from both tables that join with each other as well as tuples that belong to only ShipCost, and only Sales, whether or not they have matching tuples in the other relation. Now, suppose that we perform a full outer join between the ShipCost and Sales tables.

```
SELECT Sales.CustID, Sales.ProdID, Sales.TimeID, SUM (Sales), SUM (SCost)
FROM Sales, ShipCost
WHERE Sales.ProdID(+) = ShipCost.ProdID(+)
AND Sales.TimeID(+) = ShipCost.TimeID(+)
GROUP BY Sales.CustID, Sales.ProdID, Sales.TimeID
```

In the above SQL, the (+) notation is used to denote an outer join. The outer join results in a new table T_o with the schema T_o(CustID, ProdID, TimeID, S_Sales, S_SCost). The queries $(\text{TimeID})_{\text{SCost}}$ and $(\text{TimeID})_{\text{Sales}}$ can be answered using the following SQL statements:

```
SELECT TimeID, SUM (SCost)
FROM T_o
WHERE SCost is NOT NULL
GROUP BY TimeID
```

```
SELECT TimeID, SUM (Sales)
FROM T_o
WHERE Sales is NOT NULL
GROUP BY TimeID
```

On the other hand, to answer a query on $(\text{CustID, TimeID})_{\text{Sales}}$ requires the SQL.

```
SELECT CustID, TimeID, SUM (Sales)
FROM T_o
WHERE Sales is NOT NULL
AND SCost is NOT NULL
GROUP BY CustID, TimeID
```

For one derived metric, there are three possible aggregates which can be queried. Let us look at what happens when we join in a third table, ProdCost to T_o using an outer join. The aggregates which can be queried are: $(\text{TimeID})_{\text{Sales}}$, $(\text{TimeID})_{\text{Sales,SCost}}$, $(\text{TimeID})_{\text{PCost}}$, $(\text{TimeID})_{\text{SCost,PCost}}$, $(\text{TimeID})_{\text{SCost,Sales}}$, and $(\text{TimeID})_{\text{PCost,Sales}}$, $(\text{TimeID})_{\text{SCost,PCost,Sales}}$. If we try to precompute aggregates from T_o, it is not clear which aggregate should be precomputed. The table used by $(\text{TimeID})_{\text{Sales}}$ cannot be used to answer $(\text{TimeID})_{\text{Sales,SCost}}$. Hence both aggregates have to considered separately by an aggregate selection algorithm for precomputation. This negates the advantage of having a single lattice. Besides, this goes against the data model design, which splits this one table into multiple tables for reasons of efficient storage, and to avoid the various anomalies associated with denormalized schemas. Clearly, it makes sense to precompute the natural join and compute the derived metric. In the above example, we would precompute $T_{\bowtie} = T_e$(CustID, ProdID, TimeID, Profit), where Profit = (Sales - UnitsSold∗(SCost - PCost)). This avoids the repetition of SCost for each ProdID, and repeating the PCost for each CustID.

From this discussion it is clear that the single universal relation like table approach is fraught with difficulty. This single table must be materialized from a full multi-way outer join. It is likely to be very large, and to contain a lot of redundant information. Finally, mapping multidimensional queries to this single table is likely to be inefficient because each aggregate requires a *slightly* different NULL value filtering, and the NULL filtering itself must be performed. Accordingly, we look for alternative approaches to speeding up OLAP queries using precomputation in section 4.

3.3 Multi-cube Join Quirks

Let us look at the various issues that arise when multiple-cubes are joined, and their join is precomputed. The basis of star schema data modelling is that the dimensions determine the measures and there are no other dependencies. That is, the metrics in a table are functionally dependent on the dimensions. If the base data consists of *aggregate views* of some base schema, then we cannot synthesize a lost dimension using joins. Let us assume that the following functional dependencies exist: $CPT \rightarrow Sales$; $PT \rightarrow PCost$, where C, P, and T stand for CustID, ProdID, and TimeID respectively. But if the tables in the database are actually:

SalesNoTime(ProdID, CustID, Sales, UnitsSold)
ProdCost(ProdID, TimeID, PCost)

Then, one cannot join the two tables to get $(\text{TimeID})_{\text{Sales}}$. We call such an aggregate a "*phantom*" table since it can be obtained from the original data, but not from the derived view.

Another interesting quirk arises when performing a join between two tables; one cannot aggregate out a join column before performing the join. This leads to a lossy join in the sense that we lose the ability to distinguish which tuple should be in the result. For example, in the following schema,

Sales(ProdID, CustID, TimeID, Sales, UnitsSold)
CustCost(ProdID, CustID, TimeID, Cost)

with the functional dependencies $CPT \rightarrow Sales$, $PTC \rightarrow Cost$, if the join columns are ProdID (P), CustID (C), and TimeID (T), then we cannot join the aggregate $(\text{CustID, ProdID})_{\text{Sales}}$ with $(\text{ProdID, CustID, TimeID})_{\text{Cost}}$ to compute the result of a query. This importance of the need for this discussion will become clearer when we consider join benefits of aggregate nodes in a lattice.

3.4 The Subset Assumption

Now we discuss the subset assumption, and show how query semantics can be ambiguous without it. Consider a query which joins the `ProdCost` and `ShipCost` tables from the schema described in Section 1.1. The functional dependencies are: PT → PCost, CT → SCost. Neither of these two fact tables is contained in the other. This leads to interesting queries such as $(\text{Customer})_{\text{PCost}}$ which are ambiguous since they can be obtained using either of the following two SQL statements.

```
SELECT ShipCost.Customer,  SUM (PCost)
FROM ShipCost, ProdCost
WHERE ShipCost.Time = ProdCost.Time
GROUP BY ShipCost.Customer
```

```
SELECT ShipCost.Customer, T1.Sum_PCost
FROM ShipCost, (SELECT SUM(PCost) AS Sum_PCost  FROM ProdCost) T1
```

The first SQL query joins the two tables on their join dimensions, which leads to an answer which may not have any meaning. The second SQL query computes the total product cost and repeats it for each customer. This might be more accurate in capturing what the user wants since the product cost is not dependent on the Customer attribute. We define "dimensional containment", which makes it easier to define what joins will result in meaningful results.

Definition 4. *When the dimensions of one fact table T_1 are a subset of the dimensions of another fact table T_2, then T_2 is said to dimensionally contain T_1.*

Since query semantics are ambiguous without it, we assume that when two or more tables are joined, one of the tables dimensionally contains the others. For example, from Section 1.1, the Sales table dimensionally contains both ShipCost and ProdCost. Tables dimensionally contained by the Sales table can be joined through it to answer queries. For example, a query such as $(\text{CustID})_{\text{PCost}}$ can be unambiguously answered using the following SQL:

```
SELECT Sales.CustID,  SUM (ProdCost.PCost)
FROM Sales, ProdCost
WHERE Sales.Product = ProdCost.Product
AND Sales.Time = ProdCost.Time
GROUP BY Sales.CustID
```

Next we examine the precomputation problem for multi-cube systems. We start with a framework for multidimensional datasets.

4 New Aggregate Selection Techniques for Multi-cubes

In this section we first extend the cost model to account for the benefits of aggregates arising from the existence of multiple cubes. Then, we propose aggregate selection algorithms based on the new benefit model.

4.1 Benefits across Multiple Cubes

Let us look at the lattices $\mathcal{L}_1$, $\mathcal{L}_2$ for two cubes with schemas $\text{T}_1(\text{A,B,M}_1)$ and $\text{T}_2(\text{A,B,C,M}_2)$, where A, B, C are the dimensions, and M_1, M_2 are the measures. To obtain an aggregate containing a derived metric composed of metrics from both cubes, we have to join the two tables. In the multi-cube scenario, each precomputed aggregate can potentially have a join benefit in addition to the simple benefit

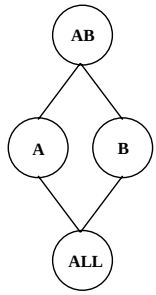

Fig. 2. Lattice $\mathcal{L}_1$

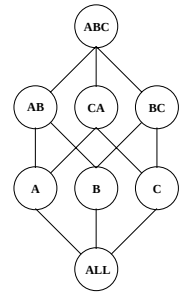

Fig. 3. Lattice $\mathcal{L}_2$

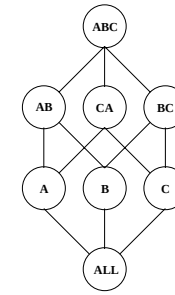

Fig. 4. Lattice $\mathcal{L}_{\bowtie}$

described in section 2.3. The benefit of the aggregate $\mathcal{L}_2$(A,B) is larger because any aggregate which required the join of $\mathcal{L}_1$(A,B) with $\mathcal{L}_2$(A,B,C) can now be obtained by joining $\mathcal{L}_1$(A,B) with $\mathcal{L}_2$(A,B). Thus, the precomputation of $\mathcal{L}_2$(A,B) has a benefit to its children in $\mathcal{L}_2$, and a benefit to the aggregates having derived metrics which result from a join. Let us quantify this join benefit.

We assume that joins are performed using a hash join based algorithm. So, if nodes $\mathcal{L}_1$(A,B) and $\mathcal{L}_2$(A,B) are joined to answer the query, then we assume that each node is scanned and partitioned on the join dimensions, and written to disk. Then these partition are read in to perform the join. Therefore, counting only I/O costs (as we have been doing so far), the join cost is approximately equal to $2(|\mathcal{L}_1(A,B)| + |\mathcal{L}_2(A,B)|)$. The derived metric results in a virtual lattice $\mathcal{L}_{\bowtie}$ (see Figures 2, 3, and 4). The aggregate $\mathcal{L}_2$(A,B) benefits $\mathcal{L}_{\bowtie}$(A,B) and all its descendants. The cost savings from precomputing $\mathcal{L}_2$(A,B) (ie, the benefit of $\mathcal{L}_2$(A,B)) are computed as follows: $2(|\mathcal{L}_1(A,B)| + |\mathcal{L}_2(A,B,C)|) - 2(|\mathcal{L}_1(A,B)| + |\mathcal{L}_2(A,B)|)$ which is equal to $2(|\mathcal{L}_2(A,B,C)| - |\mathcal{L}_2(A,B)|)$. The benefit per unit space of $\mathcal{L}_2$(A,B) increases by

$$\mathcal{J}_s(\mathcal{L}_2(A,B),\mathcal{S}) = \frac{2(|\mathcal{L}_2(A,B,C)| - |\mathcal{L}_2(A,B)|)}{|\mathcal{L}_2(A,B)|} * (\#\text{ benefited nodes in other lat.})$$

where $\mathcal{J}_s$ denotes the join benefit of $\mathcal{L}_2(A,B)$, and $\mathcal{S}$ is the set of aggregates already selected for precomputation. In addition, the number of nodes in $\mathcal{L}_1$ benefited by the precomputation of $\mathcal{L}_2(A,B)$ is 4 (namely, $\mathcal{L}_1(A,B)$, $\mathcal{L}_1(A)$, $\mathcal{L}_1(B)$, and $\mathcal{L}_1(ALL)$. From the discussion in section 3.3 about multi-cube join semantics, one should note however that the number of benefitted join nodes can be zero. For example, neither $\mathcal{L}_1(A)$ nor $\mathcal{L}_2(A)$ have any join benefit since joining either node with a node in the other lattice does not lead to a semantically valid answer. Thus we can define the total benefit of $\mathcal{L}_2(A,B)$ as: $\mathcal{B}_s(\mathcal{L}_2(A,B),\mathcal{S}) + \mathcal{J}_s(\mathcal{L}_2(A,B),\mathcal{S})$ We define the *principal* lattice as the lattice that dimensionally dominates all the other tables in the join. For example, $\mathcal{L}_2$ is the principal lattice for the derived metric obtained by joining $\mathcal{L}_2(A,B,C)$ with $\mathcal{L}_1(A,B)$ (see Figure 3). Aggregates from lattice $\mathcal{L}_1$ cannot be used in a join to obtain any nodes in $\mathcal{L}_{\bowtie}$ as it violates the rule that one cannot aggregate out a join column before performing the join (see section 3.3). Thus, only aggregates from the principal lattice can be used, and the dimensions of the least aggregated node of the principal node must contain the join dimensions. For example, the least aggregated node of $\mathcal{L}_2$ than can be used is (A,B).

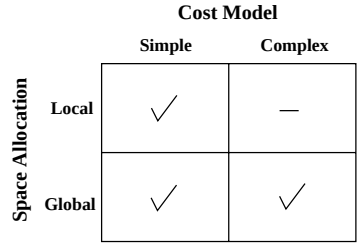

Fig. 5. The different multi-cube algorithm strategies

4.2 Aggregate Selection for Multi-cubes

Now we present three algorithms that pick aggregates for schemas having multi-cube data models. There are two different parameters that we vary to obtain different algorithms for aggregate selection. The first is the space allocation strategy, which can be either *local* or *global*. Local space allocation means that the available space for precomputation is divided up among the cubes, and aggregate selection algorithms are then run on each cube. Global space allocation means that space is not divided among the cubes, and aggregate selection algorithms pick the best aggregates from all cubes simultaneously. The second parameter is the cost model. The cost model can be *simple* (as described in section 2.3, which doesn't consider derived metrics and virtual cubes, or *complex* (as described in section 4.1, which accounts for the benefit arising from the joins required by derived metrics. The complex cost model was described in Section 4.1. The algorithms which result from varying these two parameters can be classified using the grid in Figure 5.

We will refer to the four algorithms by combining the type of cost model followed by the space allocation strategy. For example, *ComplexGlobal* refers to the algorithm which uses the complex cost model and a global space allocation strategy. The algorithm which results from using the complex cost model and a local space allocation strategy doesn't make sense since the cost model captures benefits across cubes, but the space allocation is local to each cube. Therefore, we don't discuss the *ComplexLocal* algorithm. We will study the SimpleLocal, SimpleGlobal, and ComplexGlobal algorithms. The experimental studies will compare the Simple and Complex cost models to quantify the improvement in query response time obtained by using the complex cost model.

The algorithm SimpleLocal works by dividing the space among the cubes, and then executing the algorithm PickAggregates on each cube. The inputs to PickAggregates are: space – the amount of space available for precomputation, and $\mathcal{A}$, a set initially containing all aggregates in the lattice, except the base table. The output is $\mathcal{S}$, the set of aggregates to be precomputed. The algorithm is as follows:

Algorithm PickAggregates

WHILE (space > 0) **DO**
  w = aggregate having the maximum simple 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
  Update the benefit of affected nodes in the lattice
$\mathcal{S}$ is the set of aggregates picked by BPUS

SimpleGlobal differs from SimpleLocal in that the set of aggregates it has to choose from is the union of the sets of aggregates from each cube. Therefore, for SimpleGlobal, $\mathcal{A}$, is a set initially containing all aggregates in all the lattices, except the base tables of those lattices. ComplexGlobal differs from SimpleGlobal only in the second step of the algorithm.

---

w = aggregate having the maximum complex benefit per unit space in $\mathcal{A}$

The series of steps required to update the benefits of the aggregates in the lattices for ComplexGlobal is interesting. Thus a brief description is presented next.

Algorithm UpdateBenefits

IF the aggregate is picked from a virtual lattice
  Update aggregates in the virtual lattice
  A node in the Principal lattice has a reduced benefit
ELSE
  Update aggregates in the lattice
  Update the cheapest parent for any affected Virtual lattices
    also update the benefit of the previous cheapest parent.

Let us examine these steps in a little more detail. At each step the greedy algorithm picks the aggregate with the highest benefit. If the aggregate v with the maximum benefit is from a virtual lattice, then we have to first update all descendants of v to see if they can now be computed less expensively using v. The benefit of all the ancestors of v also has to be reduced since they no longer benefit the computation of v or any of its descendants which were updated. Now the benefits of the nodes which benefited v in the principal lattice have to be reduced because they no longer benefit v or its descendants.

On the other hand, if the aggregate u is picked from a non-virtual cube, then its descendants have to be updated since it might be cheaper to compute them using u. Ancestors of u also have to be updated to reduce their benefit (if any) to u. If u belongs to the cube $\mathcal{P}$, then aggregates in a virtual cube $\mathcal{V}$, for which $\mathcal{P}$ is the *principal* lattice, also have to be updated. Any aggregates v in $\mathcal{V}$ for which it is now cheaper to use u should be updated to reflect this information. In addition, the benefit of the previous aggregate used to compute v should be reduced if it is cheaper to compute v using u.

We can show that the greedy algorithm never performs too badly. In fact, if f is the ratio of the size of the largest aggregate to the amount of space available for precomputation, it can be shown that the benefit of the greedy algorithm is at least $(0.63 - f)$ of the benefit of the optimal algorithm. (0.63 arises from $(e-1)/e$, where e is the base of the natural logarithm). The proof is very similar to that presented for the greedy algorithm in [7], so we do not present it here. The algorithm *PBS* proposed in [10] can be used for lattices which are SR-hypercube (see [10]). In addition, the lattice must not be the principal lattice for some derived metric.

5 Experimental Evaluation

In this section we quantify the improvement in average query cost (average query response time) that can be achieved by the use of the complex cost model. We

Fact Table	Size (Tuples)	Component Dimensions	Metrics
Budget	250,000	Prod, Cust, Scenario, Time	UnitSales, DollarSales
Inv	50,000	Prod, Cust, Channel, Time	Inventory
ProdCost	221,000	Prod, Scenario, Time	ProductCost
Sales	146,000	Prod, Cust, Channel, Scenario, Time	UnitSales, DollarSales
ShipCost	64,000	Cust, Scenario, Time	ShippingCost

Table 1. The APB-1 benchmark schema

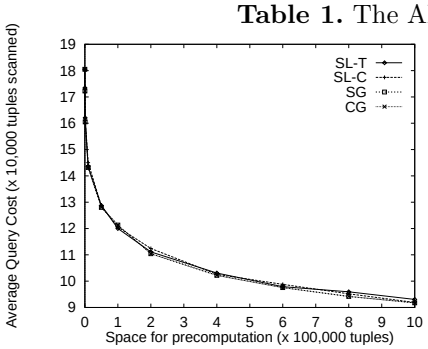

Fig. 6. The average query cost as space is varied for experiment 1 (2 tables, no derived metrics).

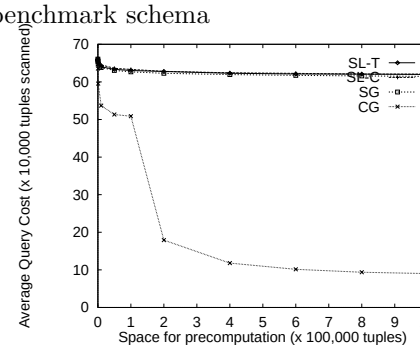

Fig. 7. The average query cost as space is varied for experiment 2 (3 tables, 1 derived metric).

used the schema for the APB-1 benchmark [1]. APB-1 is the industry standard benchmark defined by the OLAP council [1], which has most of the major OLAP vendors as members. The APB-1 benchmark database consists of five dimensions, Product, Customer, Channel, Scenario, and Time. The Product, Customer, and Time dimensions have 6, 2, and 3 level hierarchies defined on them respectively, while the channel and scenario dimensions don't have any hierarchies (only 1 level). The dimensions and their levels followed by the number of distinct values in parenthesis are described next. The Product dimension has 6 levels: Code (9000), Class (900), Group (100), Family (20), Line (7), and Division (2). The Customer dimension has 2 levels: Store (900) and Region (100). The Channel and Scenario dimensions have 1 level each (no hierarchy), and 10, 3 distinct values respectively. The Time dimension has three levels, Month (24), Quarter (8), and Year (2). There are five fact tables, Budget, Inv, ProdCost, Sales, and ShipCost. The Budget table contains the budgeted (scenario = 'Budget') UnitSales, and DollarSales of a product to a customer in a given month. The inventory is stored in the Inv table by product, customer, sales channel, and time. The ProdCost table stores product costs by product, scenario, and time, while ShipCost stores the shipping cost by customer, scenario and time. Lastly, the Sales table contains the actual sales of a product to a customer using some channel in some month. The five fact tables, their dimensions, and their sizes (in tuples) are shown in Table 1. We used the analytical formulas presented in [9] to estimate the size of aggregates formed by the 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 fact table. 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 what is done in relational group by size estimation. All the experiments vary the amount of space used for precomputation, and plot the average query cost as a function of the amount of space.

The two derived metrics are *Profit*, and *Sales increase*. Profit is computed by joining the Sales, ProdCost and ShipCost tables along their common dimensions, with Profit = DollarSales - UnitsSold * (ProductCost + ShippingCost). Sales increase is computed by joining the Sales and the Budget tables along their common dimensions, and subtracting Budget.DollarSales from Sales.DollarSales.

We ran four experiments on the APB schema by restricting the precomputation to specific tables, and specific derived metrics. We also restricted the measurement of the average query cost to those tables. The graphs show the change in average query cost as the space available for precomputation is increased. We explored two different strategies for splitting space for the SimpleLocal (SL) algorithm, SL-T where the space is split in the ratio of the fact table sizes, and SL-C where the split is in the ratio of the cube sizes. The experiments are:

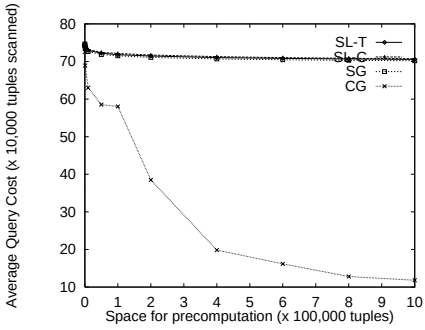

Fig. 8. The average query cost as space is varied for experiment 3 (4 tables, 2 derived metrics).

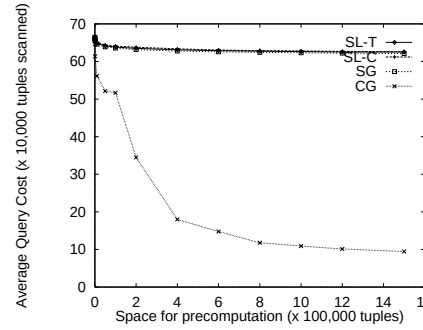

Fig. 9. The average query cost as space is varied for experiment 4 (5 tables, 2 derived metrics).

- Expt 1: The schema contains the Sales, Budget tables, and no derived metrics. This experiment studies the effect of multiple fact table without any derived metrics. Since there are no derived metrics, SimpleGlobal and ComplexGlobal pick the same set of aggregates. The graph is shown in Figure 6.
- Expt 2: The schema contains the Sales, ShipCost, ProdCost tables, and one derived metric (Profit). In this experiment we study the effect of a derived metrics formed from three fact tables. The graph is shown in Figure 7.
- Expt 3: The schemas consists of the Sales, ShipCost, ProdCost, Budget tables, and two derived metrics (Profit and Sales increase). In this experiment we study the effect of having two derived metrics. The graph is shown in Figure 8.
- Expt 4: This schema consists of the entire APB-1 database, and contains all five fact tables, and two derived metrics (Profit and Sales increase). The graph is shown in Figure 9.

In plotting these graphs we have assumed that queries uniformly access all aggregates in all lattices. The graphs show that as the space is increased, the average

query cost reduces rather rapidly at first, and then the rate of decrease of average query cost reduces significantly. We can see that ComplexGlobal outperforms any of the other algorithms because it precomputes nodes in the lattice resulting from the virtual fact table having the derived metrics. For example, let us look at Figure 7. The sudden reduction in average query cost when the space changes from 100,000 to 200,000 tuples occurs because a detailed level aggregate of the virtual lattice now fits and is picked for precomputation. This results in a large decrease in the average query cost since queries on a large number of aggregates in the virtual lattice can now be computed using the detailed level aggregate instead of performing a join. The reduction in the average query cost is not as dramatic in Figures 8 and 9 because there are two virtual lattices, and there are actually two drops in the average query cost corresponding to the picking of detailed level aggregates from the two virtual lattices. As we can see from Figure 9, the average query cost of the set of aggregates picked using the simple cost model can be four times the average query cost of the set of aggregates picked using ComplexGlobal.

6 Conclusions

Precomputing aggregates on some subsets of dimensions and their corresponding hierarchies can substantially reduce the response time of a query. While this problem has been well-studied in the recent database literature, to our knowledge we are the first to study aggregate selection for multi-cube data models which compute aggregates over multiple cubes. In this paper, we analyzed the multi-cube precomputation problem in detail. We showed that this problem is significantly more complex than the single cube precomputation problem, and that algorithms and cost models developed for single cube precomputation must be extended to deal well with the multi-cube case. We proposed three different algorithms, SimpleLocal, SimpleGlobal, and ComplexGlobal which pick aggregates for precomputation from multi-cube schemas. Our results from a prototype implementation show that for multi-cube workloads substantial performance improvements can be realized by using multi-cube algorithms instead of the previously proposed single cube algorithms. In particular, the complex cost model considers join costs, leading to a much better set of aggregates picked for aggregation.

To conclude, we discuss which algorithm is appropriate for a given schema. The algorithm of choice will depend on the existence of derived metrics which require fact table joins to be performed. In the absense of derived metrics, the complex cost model reduces to the simple cost model. Further, if the lattice is a SR-hypercube lattice [10], the algorithm PBS proposed by [10] can be used. For non-SR-hypercube lattices, either SimpleLocal or SimpleGlobal can be used. If the schema contains derived metrics, ComplexGlobal should be used.

References

1. APB-1 Benchmark, Release II, November 1998. Available from `http://www.olapcouncil.org/research/bmarkly.htm`.
2. E. Baralis, S. Paraboschi, E. Teniente. Materialized View Selection in a Multidimensional Database, *Proc. of the 23rd Int. VLDB Conf.*, 1997.
3. R. Elmasri, S. Navathe, *Fundamentals of Database Systems*, The Benjamin/Cummings Publishing Company, Inc., 1989.

4. 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.
5. H. Gupta, V. Harinarayan, A. Rajaraman, J.D. Ullman. Index Selection for OLAP. *Proc. of the 13th ICDE*, 208–219, 1997.
6. H. Gupta. Selection of Views to Materialize in a Data Warehouse. *Proc. of the Sixth ICDT*, 98–112, 1997.
7. V. Harinarayan, A. Rajaraman, J.D. Ullman. Implementing Data Cubes Efficiently, *Proc. ACM SIGMOD Int. Conf. on Man. of Data*, 205–227, 1996.
8. Nigel Pendse and Richard Creeth, The Olap Report. Information available from `http://www.olapreport.com/`.
9. 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.
10. A. Shukla, P.M. Deshpande, J.F. Naughton, Materialized View Selection for Multidimensional Datasets, *Proc. of the 24th Int. VLDB Conf.*, 1998.
11. A. Shukla, Materialized View Selection for Multidimensional Datasets, *Ph.D. Dissertation, University of Wisconsin - Madison*, 1999.
12. Erik Thomsen, *Olap Solutions : Building Multidimensional Information Systems*, John Wiley & Sons, 1997.
13. J.D. Ullman, Efficient Implementation of Data Cubes Via Materialized Views A survey of the field for the 1996 KDD conference.
14. J.D. Ullman, Principles of Database and Knowledge–base Systems, Volume II, Computer Science Press, 1988.