Simultaneous Optimization and Evaluation of Multiple Dimensional Queries

Yihong Zhao Prasad M. Deshpande Jeffrey F. Naughton Amit Shukla*

Computer Sciences Department
University of Wisconsin, Madison

{zhao, pmd, naughton, samit }@cs.wisc.edu

Abstract

Database researchers have made significant progress on several research issues related to multidimensional data analysis, including the development of fast cubing algorithms, efficient schemes for creating and maintaining precomputed group-bys, and the design of efficient storage structures for multidimensional data. However, to date there has been little or no work on multidimensional query optimization. Recently, Microsoft has proposed "OLE DB for OLAP" as a standard multidimensional interface for databases. OLE DB for OLAP defines Multi-Dimensional Expressions (MDX), which have the interesting and challenging feature of allowing clients to ask several related dimensional queries in a single MDX expression. In this paper, we present three algorithms to optimize multiple related dimensional queries. Two of the algorithms focus on how to generate a global plan from several related local plans. The third algorithm focuses on generating a good global plan without first generating local plans. We also present three new query evaluation primitives that allow related query plans to share portions of their evaluation. Our initial performance results suggest that the exploitation of common subtask evaluation and global optimization can yield substantial performance improvements when relational database systems are used as data sources for multidimensional analysis.

1 Introduction

In the past few years, database researchers have made great progress on OLAP issues such as cubing algorithms, techniques for effectively creating and maintaining materialized group-bys, and multi-dimensional storage structures and indexing. However, to our knowledge there has been very little attention devoted to optimizing multiple simultaneous OLAP queries. In addition to its intrinsic interest, the problem of optimizing multiple simultaneous OLAP queries is likely to be of considerable practical importance due to recent developments in the industry.

Specifically, Miscrosoft recently released its proposed "OLE DB for OLAP" standard for interfaces to multidimensional data sources [MS]. This standard, or one heavily influenced by this standard, is likely to become widely supported. One of the most interesting aspects of this standard is that it defines "Multi-Dimensional Expressions" (MDX), which provide a framework in which a user can very naturally ask several related OLAP queries in a single MDX query. This aspect of MDX reflects the fact that OLAP-style analysis very often gives rise to simultaneous related queries. MDX is intended to be a uniform front end for a variety of data sources, including multidimensional database systems and relational database systems. In this paper we consider the evaluation of MDX expressions by relational database systems.

Of course, the fact that the queries are expressed in a single MDX expression does not mean that the data source must evaluate them as a single unit; a data source can always evaluate the queries one after another without regard for the relationships between them. However, as we show in this paper, doing so typically misses an opportunity for substantial performance gains that could be achieved by optimizing and executing these queries as a unit. While our results apply directly to MDX, they would be equally applicable to other language frameworks that allow the expression of multiple dimensional queries.

While similar problems have long been studied in the context of global query optimization (see, for example, [PS88], [S88], and [SS94]), the multidimensional nature of the simultaneous queries found in MDX expressions present both new opportunities and new challenges. The opportunities arise primarily because of the restricted nature of the queries in question — the queries in MDX expressions typically look (in relational terms) like a select-star-join followed by an aggregation at some level in dimension hierarchies. The restricted domain of the queries facilitates the identification and exploitation of their common subtasks.

The first contribution of this paper is the design of several new query operators that allow multiple related star join queries to share common subtasks, even if each plan uses a different star join method. The new operators are shared scan hash-based star join, shared scan hash-based and join index-based star join, and shared join index-based star join. In the performance study, we show that these three query operators improve the performance of multiple related dimensional queries.

*This research was supported by a gift from the NCR Corp., and by ARPA through Rome Air Force Laboratory cntract F30602-97-2-0247.

A significant challenge in multiple dimensional query optimization and evaluation arises due to the way multidimensional data sources attempt to speed up query evaluation using precomputation [GHQ95, HRU96, CR96]. Simply put, in general, for any dimensional query there will be a number of distinct precomputed aggregates that can be used as the data table from which to evaluate the query. Choosing the best aggregate to use is simple in the context of a single dimensional query; however, choosing the correct set of tables to use for a set of dimensional queries is non-trivial.

The second contribution of our paper is the development of algorithms that choose which aggregate tables to use to evaluate a set of dimensional queries. These algorithms are **TPLO** (Two Phase Local Optimal), **ETPLG** (Extended Two Phase Local Greedy), and **GG** (Global Greedy). These algorithms differ in how aggressively they search for query plans that contain shared subtasks. We will discuss each of these in detail in the remainder of the paper and give examples of their performance implications. The evaluation plans generated by these algorithms make use of the new query operators introduced above in order to share common subtasks. The results obtained from our implementation suggest that common subtask sharing and global optimization can provide substantial performance improvements when relational database systems are used as data sources for multidimensional analysis.

We organize the paper as the following. In Section 2, we introduce MDX. In Section 3, we discuss the three shared operators. We present the TPLO algorithm in Section 4, the ETPLG algorithm in Section 5, and the GG algorithm in Section 6. We study and compare our three algorithms in Section 7. Finally, we conclude in Section 8.

2 Multi-Dimensional Expressions (MDX)

Due to space constraints, in this section we only discuss the MDX features which are relevant to the paper. The reader interested in more detail is referred to [MS]. We mainly focus on expressing several related OLAP queries in a single MDX expression. The Microsoft document [MS] includes the following example:

```
NEST ({Venkatrao, Netz},
      {USA_North.CHILDREN, USA_South, Japan})
  on COLUMNS
  {Qtr1.CHILDREN, Qtr2, Qtr3, Qtr4.CHILDREN }
  on ROWS
CONTEXT SalesCube
FILTER(Sales, [1991], Products.All)
```

`NEST, CONTEXT, COLUMNS, ROWS, CHILDREN, and FILTER` are all reserved MDX keywords. This query asks for the total sales for salesmen Venkatrao and Netz in all states of USA_North, USA_South, and Japan for all the months of the first quarter, the second quarter, the third quarter, and all the months of fourth quarter, for 1991.

In relational terms, the joins between the fact table and dimension tables are defined implicitly in an MDX expression. Therefore, we do not see the join attributes and join conditions. This is because an MDX expression does not define how an OLAP server processes a MDX query — MDX is supposed to be system independent, working for both Relational OLAP (ROLAP) and Multidimensional OLAP (MOLAP).

For the above MDX example, we assume that the database has five dimensions and one fact table (WholeSalesData.) The five dimensions are the Time, Store, Product, Sales person, and Measure. On the Time dimension, we have the Date → Month → Quarter → Year hierarchy. Ths Store dimension contains the Store → City → State → Region → Country hierarchy.

In terms of SQL statements, this MDX expression specifies six different group-by queries:

1. the total sales for Venkatrao and Netz in all states of USA_North for the 2^{nd} and 3^{rd} quarters in 1991
2. the total sales for Venkatrao and Netz in all states of USA_North for the Months of the 1^{st} and 4^{th} quarters in 1991
3. the total sales for Venkatrao and Netz in region USA_South for the 2^{nd} and 3^{rd} quarters in 1991
4. the total sales for Venkatrao and Netz in region USA_South for the Months of the 1^{st} and 4^{th} quarters in 1991
5. the total sales for Venkatrao and Netz in Country Japan for the 2^{nd} and 3^{rd} quarters in 1991
6. the total sales for Venkatrao and Netz in Country Japan for the Months of the 1^{st} and 4^{th} quarters in 1991

More succinctly, we have the six group-bys (SalesPerson, State, Quarter), (SalesPerson, State, Month), (SalesPerson, Region, Quarter), (SalesPerson, Region, Month), (SalesPerson, Country, Quarter), and (SalesPerson, Country, Month). In addition, for each group-by we have disjoint selection predicates. This means that we cannot use the "finding the Common Selection predicates" techniques that are so widely used in multi-query optimization algorithms for general SQL queries [S88].

In this paper, for simplicity of notation we interchange the MDX query with the target group-by corresponding to the query. For example, we use $A'B'C'$ both to refer to the target list of a group by and to the query that computes this group-by. For a relational data source storing its data in a star schema, each individual query of an MDX expression can be evaluated using a star join query followed by an aggregation for the target group-by. In addition, a typical query of MDX includes a selection predicate along each join dimension. In the rest of the paper, we use SQL and MDX interchangably to describe queries.

3 Operators for Merging Query Plans

The fundamental task in evaluating multiple related dimensional queries is identifying and exploiting common subtasks between the queries. In this section we propose several operators that are useful for this task in the context of relational implementations of dimensional data sets. Recall that the basic operation in such an environment is a star join. In this paper, we consider two star join methods: hash-based and index-based star join. For very selective star join queries, the join index-based star join method is a good alternative [OQ97]. For non-selective star join queries, the hash-based star join is a good solution [Su96].

3.1 Shared Scan for Hash-based Star Join

Before discussing this new operator, we describe by example how a generic pipelined right-deep star-join followed by an aggregation works in a relational system. Suppose that we have a fact table $F(A, B, dollars)$, and dimension tables $Adim(A, A')$ and $Bdim(B, B')$. Then consider evaluating the query

```
select A', B', Sum(dollars)
from Adim, Bdim, F
where Adim.A = F.A and Bdim.B = F.B
group by A', B'
```

The pipelined right-deep hash-based join begins by building hash tables on each of the (small) dimension tables $Adim$ and $Bdim$. Then, the large fact table F is streamed past these hash tables, probing them for matches in order to create joined tuples with the schema $(A', B', dollars)$. Finally, these joined tuples are passed to an aggregation operator to compute the group by on the attributes A' and B'. We assume that this aggregation is also done by hashing.

This pipelined right-deep query tree hash-based join method creates two different opportunities for sharing among query trees. Suppose we have two different queries sharing the same fact table. First, these query trees can share the scan of the base table. This technique is used in some commercial systems; for example, Teradata uses shared scans for multiple queries that happen to simultaneously be accessing the same table.

A more sophisticated opportunity for sharing arises if two query trees use the same set of dimension tables. Recall that the hash-based star join involves building a hash table on each dimension table, and then probing these hash tables with tuples of the fact table. If two queries use the same set of dimension tables, they can share hash tables, instead of redundantly building and probing several hash tables on the same dimension tables.

Consider the following schema. We have three dimension A, B, and C. Dimension A has the hierarchy $A \rightarrow A' \rightarrow A''$. Dimension B contains the hierarchy $B \rightarrow B' \rightarrow B''$ Dimension C includes the hierarchy $C \rightarrow C' \rightarrow C''$

In Figure 1, we show the hash-based star join query plan for computing the query $A'B'C''$ using the base table ABC. The query plans for the group-by $A'BC$ and $AB'C''$ are very similar to that of $A'B'C''$ except the aggregation step. The shared scan hash-based star join operator for the three group-bys is shown in Figure 2. Note that the scan of the base table ABC and the three join hash tables is shared by the three group-bys.

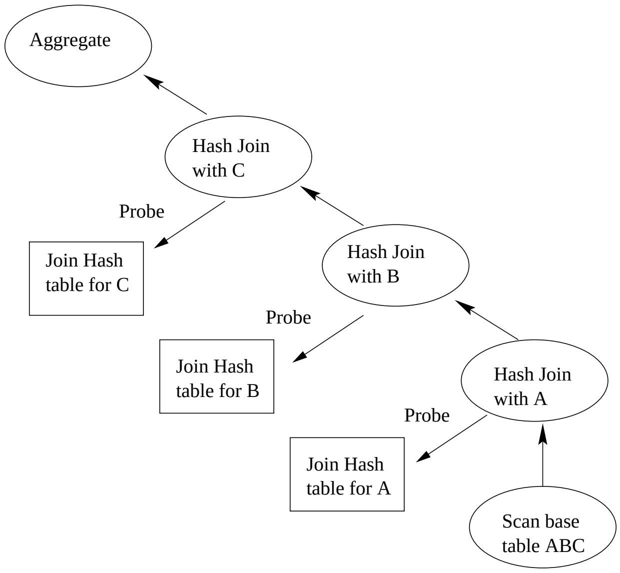

Figure 1: Query Plan for a single group-by

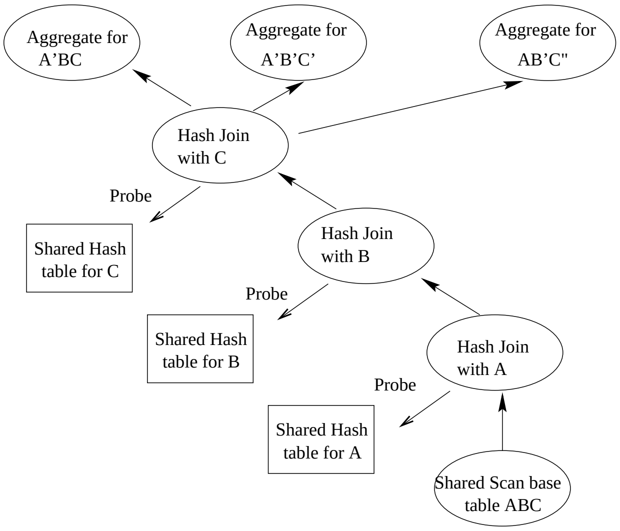

Figure 2: Query Plan for Shared Scan group-bys

After a tuple is joined with the dimension C, it is sent to the corresponding group-by's hash table for aggregation. When a tuple is fetched from the base table, we construct three result tuples for each group-by. Each tuple has to join with the dimension tables and project on the proper aggregation attributes of each group-by to the result tuple. For instance, the tuple for the group-by $A'B'C''$ has to probe the shared hash table A with its key attribute value and copy the corresponding hash entry's A' attribute to the result tuple. Then, it has to probe the hash table B and C, and copy the hash table entries' corresponding attributes B' and C'' to the result tuple. Finally, it uses the hash table for the group-by $A'B'C''$ to aggregate the query result.

3.2 Shared Index Join

This technique is useful for index-based star join query plans using the same base table. Suppose that we have the query

```
select A', B', Sum(dollars)
from Adim, Bdim, F
where Adim.A = F.A and Bdim.B = F.B
and Adim A' in {a1, a2}
and Bdim B' in {b2, b3}
group by A', B'
```

Also suppose that we have bitmap join indices mapping $Adim$'s A' attribute to tuples of F and $Bdim$'s B' attribute to tuples of F. The index-based star join proceeds by first reading the bitmaps from these two indices, and then probing F to extract the tuples that match. The idea of the new operator in this section is to let two such query plans over the same fact table share the fact table look ups. In order for the query plans to share these base table lookups, we first OR the result bitmaps for all the query plans, and then use the ORed bitmap to look up the corresponding tuples in the base table. Once the result tuples are retrieved from the base table, we send them to corresponding group-by hash tables for aggregations by consulting the group-by result bitmaps.

Figure 3 shows the query plan for the group-by $AB'C''$. First, we consider in more detail how a standard join index bitmap join works. Here, we assume that there is join bitmap index built on each attribute A, B, and C of the base table ABC. We also assume for this example, that the selectivity of the selection predicates of the group-by $A'B'C''$ is high, which causes the optimizer to use the bitmap index-based star join method. The join uses the follow steps:

1. retrieve bitmaps from the index of dimension A and OR them to generate the bitmap BP_A
2. retrieve bitmaps from the index of dimension B and OR them to get the bitmap BP_B
3. AND the two bitmaps BP_B and BP_A to get the bitmap BP_{AB}
4. retrieve bitmaps from the index of dimension C and OR them to get the bitmap BP_C
5. AND the bitmap BP_{AB} with the bitmap BP_C to generate the query result bitmap BP_{result}
6. use the bitmap BP_{result} to probe the fact table to get the results tuples
7. join the result tuples with the dimension hash tables and aggregate to produce the final result for the group-by

In Figure 4, we show how the shared bitmap star join operator works. We need to compute the queries having aggregation on group by $A'B''C''$, $A''B''$, and $A'B''C'$. The operator goes through the following steps:

1. build the join bitmaps for each query and OR the three join bitmaps to get one bitmap
2. use the bitmap to probe the base table ABC
3. use the tuples' position to split them into their corresponding group-bys. Given a tuple position, we test whether a query's bitmap's corresponding bit is set to 1. If so, we know the tuple belongs to this group-by. This task is done in each of the "Filter tuples" operators.

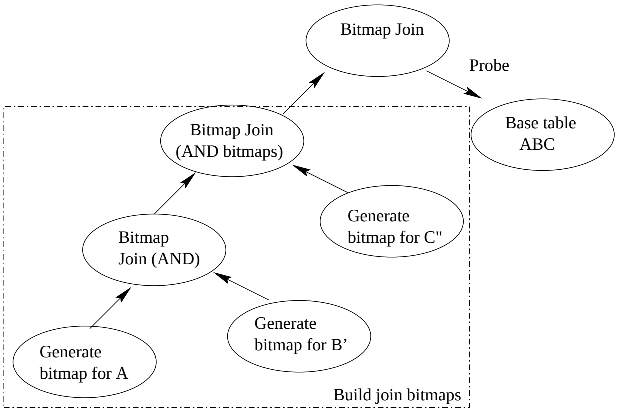

Figure 3: Query Plan for one Bitmap Star Join

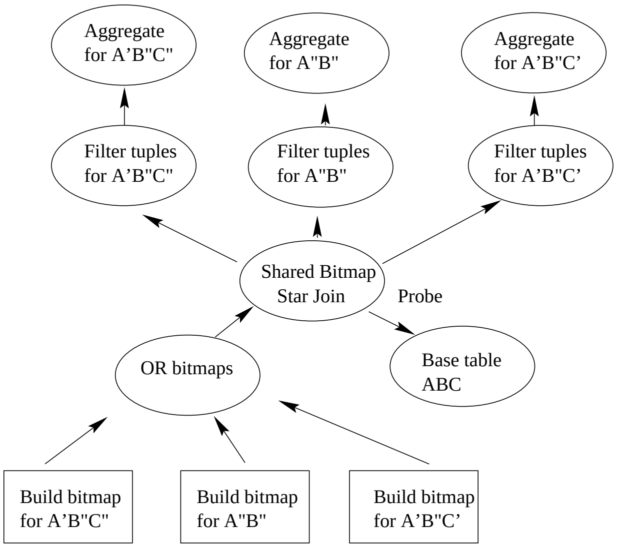

Figure 4: Query Plan for Shared Bitmap Star Join

4. aggregate the results for each query

3.3 Shared Scan for Hash-based and Index-based Star Join

The goal of this technique is to create sharing among the query plans using the same base table, but different star join methods. In general, we may have a situation where some of the plans perform a hash-based star join that needs to scan the base table; other queries use the star-join index based join (index join). Here, we assume that star-join indices can be either position based B-tree or bitmap indices. Since the query plans all have to read data from the base tables, we should merge the table scan query plans with index join plans to save the the cost of probing the base table for index-based star join plans. We know that a typical index join query has two phases: building the join bitmap and using the final bitmap to probe the base table. Here, we modify the probing of the base table to scanning it for the group-bys using index-based star join and use the result bitmap as the selection filter after the scan. This conversion allows us to share the scan of the base table among the hash-based and index based-join plans. We save the cost of

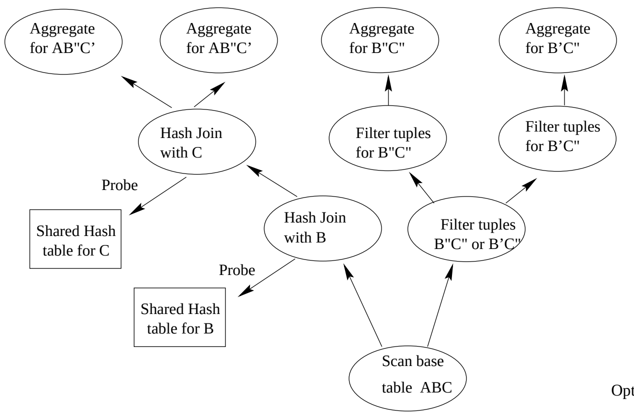

Figure 5: Shared Scan for Hash-based and Index-based Star Join

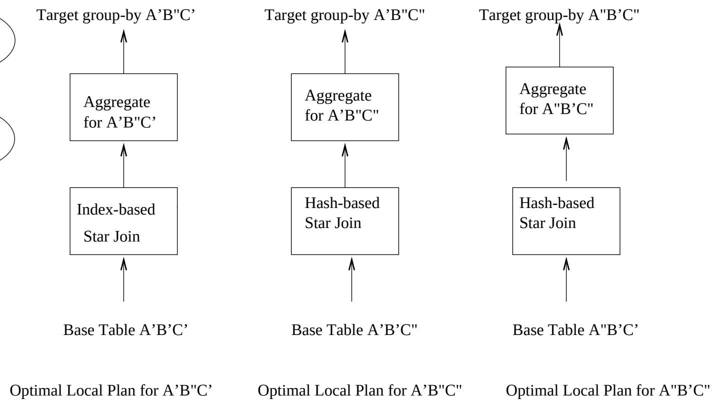

Figure 6: Phase One of Two Phase Local Optimal

probing the base table for an index-based query plan.

Figure 5 shows the operator of the shared scan for hash-based and index-based star joins. A scan of table ABC is shared by four different queries: $A'B''C'$, $AB''C'$, $B''C''$, and BC''. For each tuple read from the table ABC, we use the ORed bitmap for the query $B''C''$ and BC'' and tuple position to test whether the tuple should be aggregated for the group-by $B''C''$ and BC''. As tuples pass through the first filter, we use the the query $B''C''$ and BC'''s bitmaps to split those tuples and aggregate the corresponding tuples to each query. For the query using hash-based star join, the operator works exactly the same for shared scan hash-based star join operator.

4 Two Phase Local Optimal Algorithm

Section 3 introduced three operators for sharing subtasks among queries. In this and the two following sections we discuss optimization techniques that generate query plans for a set of queries. The goal is to produce a plan for a set of queries that allows the operators of Section 3 to be used to exploit common subtasks.

The Two Phase Local Optimal algorithm (TPLO) is perhaps the most natural and straight-forward way to approach the problem of simultaneous dimensional query optimization. TPLO breaks a MDX expression into several SQL queries. Virtually all database systems support OLAP queries by precomputing group bys; TPLO independently picks the best precomputed (materialized) group by for each of these component subqueries. Once the target "group by" for each component query has been determined, TPLO uses the SQL optimizer to generate the best plan for the queries. Finally, it generates a global plan by merging the common tasks among the query plans as much as possible, using the operators from Section 3.

We first divide the MDX expression into several SQL queries. MDX expressions use implicit joins among the fact table and dimension tables. One option that is always available is to use the lowest level fact table LL to compute each query. (LL is the base data, without any aggregation; for simplicity, we also consider LL to be a "materialized group by"). Using LL is a mistake if there is more highly aggregated materialized view that can be used to answer the query. Determining the materialized group-by to be used for a given query is done using a standard relational query optimizer by enumerating query plans for all candidate group bys and choosing the one with lowest estimated cost. We call each such plan the "optimal local plan" for the query.

For example, in Figure 6, suppose that for a given MDX expression we need to compute the group-bys $A'B''C'$, $A'B''C''$, and $A''B'C''$. We have a set of materialized group-by ABC, $A'B'C'$, $A'B'C''$, and $A''B'C'$. The TPLO would choose the materialized group-by $A'B'C'$, $A'B'C''$, and $A''B'C'$ in the optimal local plans to compute the query $A'B''C'$, $A'B''C''$, and $A''B'C''$.

Once we have the best query plan for each group-by of the MDX expression, we use the three techniques from Section 3 to merge the common tasks of the query plans, if there are any such common tasks. We call this phase two of TPLO. Returning to our example of Figure 6, we see that there are no such common subtasks. This is likely to happen with TPLO; this problem motivates the algorithms of the following sections.

5 Extended Two Phase Local Greedy Algorithm

The motivation of the Extended Two Phase Local Greedy (ETPLG) is to solve a drawback of the Two Phase Local Optimal (TPLO) algorithm. TPLO always uses the best local plan for each query of an MDX expression and merges plans at the query tree level to generate a global plan. The limitation of TPLO is that it cannot create the sharing of base tables if the optimal plans for the individual queries use different tables as their input. In some cases it may be better for individual queries to use sub-optimal base tables so that they can share tasks in their execution. We use a simple example to illustrate this.

Example 1: Suppose we need to compute the queries for the group-bys $A'B''C''$ and $A''B'C''$ of a MDX expression. The materialized group-bys include $A'B'C''$, and $A'B'C'$. TPLO generates the Plan 1 including $(A'B'C'' \Rightarrow A'B''C'')_{Hash-basedSJ}$ and $(A''B'C' \Rightarrow A''B'C'')_{Hash-basedSJ}$ in Figure 6. $(X \Rightarrow Y)_Z$ stands for compute the group-by Y using X and Z star join method. Since the group-by $A'B'C''$ is different from $A''B'C'$, there are no common query tasks to share for the two queries. The ETPLG algorithm produces the following query plan for each query: $(A'B'C'' \Rightarrow A'B'C')_{Shared-HB-SJ}$ and $(A'B'C'' \Rightarrow A'B'C')_{Shared-HB-SJ}$. The two queries $A'B''C''$ and $A''B'C''$ use the same base table $A'B'C''$. Hence, they are able to share the scan of the table $A'B'C''$. The main difference between global plans 1 and 2 is that plan 2 explores the possibility of sharing of the base table.

Example 1 suggests that we should consider using locally sub-optimal plans to create opportunities for global sharing. This is the goal of the Extended Two Phase Local Greedy (ETPLG) algorithm. ETPLG creates a global plan by adding new queries (required group-bys) one at a time. When adding a new query into the global plan, it chooses between the best plan that allows the new query to share a base table with other selected queries, and the best local plan for the query. In phase two, just like in TPLO, we merge the query plans at the query tree node level.

During optimization of a set of queries ETPLG maintains two sets. The shared group-by set contains materialized group-bys shared by the selected queries. The unused materialized group-by set includes all the materialized group-bys that are not used by any chosen query. When ETPLG picks a new query N and adds it to the global plan, it chooses the base table from the unused materialized group-by set or the shared group-by set. ETPLG calculates the cost of the best query plan in each set and picks the most efficient plan from each set. Thus we have:

- The best plan B_D that uses an unshared materialized group-by D, and
- The best plan B_S that uses a shared group-by S.

If the cost of B_D is higher than that of B_S, ETPLG adds the query N to the set of queries using the shared group by S. In general, we define a class X to be the set of queries using the same base table X. If the cost of B_D is lower than that of B_S, then it creates a new class D and adds the query N to the class D. It also needs to update the two sets by adding the group-by D to the shared group-by set and deleting it from the unshared materialized group-by set.

It is useful to define for each group by the GroupbyLevel, which is the sum of the group-by levels along each dimension. That is, a group-by's GroupbyLevel is the sum of the hierarchy level of the group-by on each dimension.

In ETPLG, we sort the queries by their GroupbyLevel and pick the next query with the smallest GroupbyLevel. The heuristic used here is to create more logical sharing, that is, sharing base tables among queries. The smaller the value of a group-by's GroupbyLevel, the more likely we can share the group-by's base table with other queries because a base table's GroupbyLevel is never greater than the computed group-by's GroupbyLevel. The smaller a group-by's GroupbyLevel, the more group-bys we can use it to compute.

Now, we present the ETPLG algorithm in greater detail.

Extended Two Phase Local Greedy Algorithm

```
//Phase One
Set SharedSet empty;
Initialize MSet;
Set Group-bys G = {G1, G2, .., Gn};
Sort G by GroupbyLevel;
For each Gi in G
{
   Find the group-by D in MSet with
   Minimum (Gi.CostOfUsing(D));
   Find the Class S in SharedSet with
   Minimum(Gi.CostOfUsing(S.BaseTable());
   B = S.BaseTable();
   if (Gi == G1 && Gi.CostOfUsing(B) > Gi.CostOfUsing(D))
   {
       SharedSet  = SharedSet Union D;
       MSet = MSet - D;
       Construct a new Class D;
       Generate the best plan Q for Gi using D;
       Add Q to the Class D;
       Add Class D to the ClassList;
   }
   else
   {
       Generate the best plan Q for Gi using B;
       Add Q to Class S;
   }
}
//Phase Two
For each Class in the ClassList
    Merge the query plans;
```

The set $MSet$ is first initialized to contain all the precomputed group-bys and the lowest level base table LL. The CostOfUsing(X) function estimates the cost of computing a query from the materialized group-by X. $X.BaseTable()$ returns the shared base table of Class X. Each class in the ClassList contains a set of queries using the same base table.

Figure 7 shows the query $A'B''C''$ being processed by the ETPLG algorithm. It adds the query $A'B'C''$ to the Class $A'B'C'$ because the cost of using the group-by $A'B'C'$ is cheaper than using the group-by $A'B'C''$ to compute the group-by $A'B''C''$. The cost of scanning $A'B'C'$ is shared by two queries. In the next Figure 8, the ETPLG algorithm adds the query $A''B''C''$ to the global plan. The ETPLG algorithm decides to use the materialized group-by $A'B'C'$ to compute the query $A''B''C''$. In the next subsection, we explain the cost model used to compute the cost of queries sharing the base table.

5.1 Cost Formula

To compute $X.CostOfUsing(B)$ for a query X using the shared materialized table B for the Class Y, we have to consider two types of cost. One is the I/O cost shared by the queries in the class Y; the other is the CPU and I/O cost, not shared by the other queries. Adding the query X to the class Y may change the shared I/O cost for the class.

If we use a hash-based star join to compute the query X from the shared base table B, we use $C_{B\rightarrow X} = Cost_{CPU} + \triangle Cost_{IO_B}$ to compute the cost. $\triangle Cost_{IO_B}$

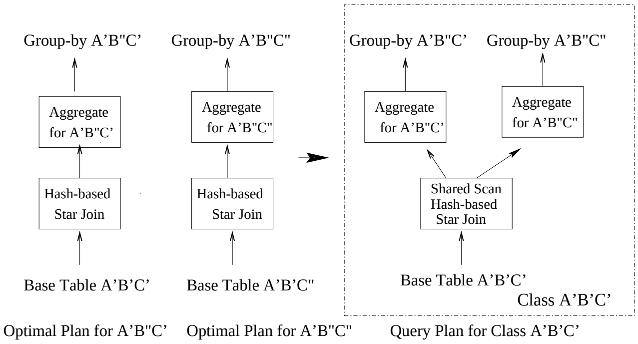

Figure 7: ETPLG: Adding the Query A'B"C" to the Global Plan

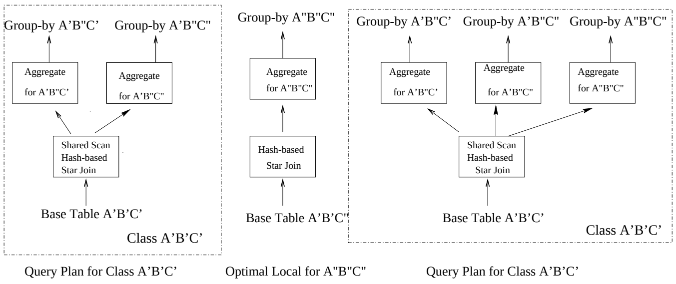

Figure 8: ETPLG: Adding the Query A"B"C" to the Global Plan

is the change of the shared IO cost for the class Y after adding X to the class. Using the hash-based star join to compute X from B may change the shared IO cost of a class. If all the other queries in the class Y use an index-based star join, using the hash-based star join to compute X changes the shared I/O cost of the class Y from a random read of some data pages to a sequential scan of the table B. In this case, the value of the $\triangle Cost_{IO_B}$ might be positive.

If we use an index-based star join to compute query X, we use $C_{B \rightarrow X} = Cost_{CPU} + Cost_{IO_{Index}} + \triangle Cost_{IO_B}$ to compute the query cost. $Cost_{IO_{Index}}$ contains the cost of the query index lookup. If any other group-by in class Y scans the group-by B, the shared I/O cost of the class $\triangle Cost_{IO_B}$ is equal to 0 because the I/O cost of probing the fact table B for query X is already counted in the scan cost. If all the group-bys in the class use an index-based star join, we have $\triangle Cost_{IO_B} \geq 0$ because we may retrieve more data pages for computing the query X.

Next, consider the situation in which we use an unused materialized table U instead of a shared group-by to compute the query X. Since the I/O cost for the table U is not shared, we use different formulas to compute *X.CostOfUsing(U)* according to the star join method.

- If the query plan for the group-by X involves a hash-based star join, we use $C_{U \rightarrow X} = Cost_{CPU} + Cost_{IO_U}$.
- If it uses the index-based star join, we choose $C_{U \rightarrow X} = Cost_{CPU} + Cost_{IO_{Index}} + Cost_{IO_U}$ to compute the query cost.

In the second case, $Cost_{CPU}$ includes the CPU cost for index operation such as ANDing or ORing bitmaps.

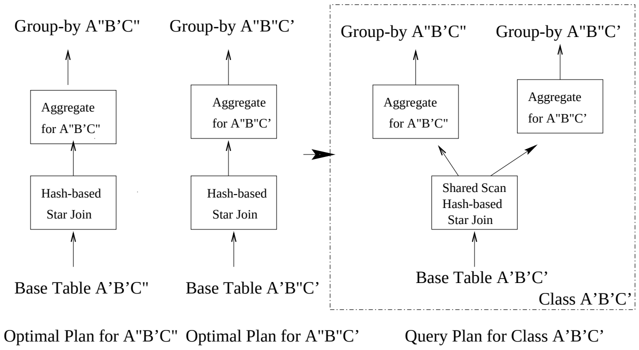

Figure 9: GG Query Plan for Example 2

Now we are ready to compute the cost of a global plan generated by ETPLG. The cost of computing all queries in the class i is given by:

$$C_{class_i} = \sum_{k=1}^{p_k} (Cost_{CPU_{plan_k}} + Cost_{IO_{plan_k}})$$

$$+ Cost(SharedIO_i) + Cost(SharedCPU_i)$$

where p_k is number of group-bys in the class i. The $Cost_{CPU_{plan_k}}$ for a plan k is the CPU cost not shared by the other plans in the class i. Similarly, The $Cost_{IO_{plan_k}}$ is the non-shared I/O cost for plan k. The cost of reading data and processing the tuples and the cost of building shared hash tables are included in $Cost(SharedIO_i) + Cost(SharedCPU_i)$. $Cost(SharedIO_i) + Cost(SharedCPU_i)$ is shared by all the group-bys of the class i. The cost of the global plan is the sum of the cost of each class. $C_{global} = \sum_{i=1}^{n} C_{class_i}$ where n is number of classes in the global plan.

6 Global Greedy Algorithm

The Extended Two Phase Local Greedy (ETPLG) algorithm employs heuristics to increase the sharing of base tables among multiple queries. But the global plan produced by the ETPLG algorithm may still not be a good global plan. The reason is that ETPLG never "changes its mind" about which base table to use for a query. We use Example 2 below to elaborate.

Example 2 : We want to compute the queries for group-bys $A''B''C'$ and $A''B'C''$ of an MDX expression. Let $A'B'C'$, $A'B'C''$, and $A'B''C'$ be precomputed group-bys. We also have $Size(A'B'C'') + Size(A'B''C') > Size(A'B'C')$, $Size(A'B'C'') < Size(A'B'C')$, and $Size(A'B'C'') < Size(A'B'C')$. The two queries are not very selective. So the ETPLG algorithm generates the global query plan for the MDX expression: $(A'B'C'' \Rightarrow A''B'C'')_{hash-basedSJ}$ and $(A'B''C' \Rightarrow A''B''C')_{hash-basedSJ}$. It uses the hash-based star join and the materialized table $A'B'C''$ to compute the query for $A''B'C''$. ETPLG would not choose the the materialized table $A'B'C'$ to compute either of the two queries since it costs more CPU and I/O time to process a large table than a small table. But, if we use the group-by $A'B'C'$ to compute both queries, we may get a better global plan. The I/O cost of scanning the table $A'B'C'$ is shared by the two queries. We reduce I/O cost for the two queries, but increase the

CPU cost since we have to process more tuples. For Example 2, let us assume the increase of CPU cost is smaller than the reduction of I/O cost. This means that we have a better global plan than the ETPLG's global plan.

To find a better global plan, we extended the ETPLG to the Global Greedy (GG) algorithm. Similar to the ETPLG algorithm, the Global Greedy algorithm grows the global plan by adding a new query (required group-by) to the plan one at a time. The difference between the two is that the GG algorithm allows a class to change its shared base table in order to include the new group-by, which is not allowed in the ETPLG algorithm.

Before adding a new query to a global plan, we have to decide whether adding the query N to any existing classes is a better plan than using the best unused materialized group-by U to compute N. For each existing class, we first have to determine a new materialized group-by S' for the class so that we can compute the query N and all queries in the class from S'. The aggregated cost of using S' for the query N and the class should be the smaller than the aggregated cost of using any other materialized group-bys. So, we pick the S'_i for each existing class i. Then, we choose the class with the minimum aggregated cost of computing all queries in the class and the query N as the candidate class. Now, we have to decide whether to use the candidate class's base table S' or use the best unused materialized group-by U to compute N. The details of this step are listed in the algorithm below. If we choose to use the candidate class to compute N, we have to use S' as the base table for the class. If the S' is different from the old base table S of the class, we replace S with S'. If $S' = S$, we use S for the class. We add the query N to the class. If we decide to use the table D to compute N, we leave the class unchanged and create a new class D and add the query N to the class.

The main difference the ETPLG and GG algorithms is that we allow the class to choose new materialized group-by as the new base table for the class when adding a new required the group-by to the class. As we explained in the Example 2, it allows to consider some suboptimal local materialized group-by as the class base table, which may produce a better global plan. We present the algorithm GG below:

Global Greedy Algorithm

```
Set ShareSet empty;
Initialize MSet;
Set Group-bys G = {G1, G2, .., Gn};
Sort G by GroupbyLevel;
For each Gi
{
    Find the group-by N in MSet
    with Min(Gi.CostOfUsing( N ));
    Choose Class Ci with Min(Ci.CostOfAdd(Gi));
    if (Gi == G1 && Ci.CostOfAdd(Gi) >
       Gi.CostOfUsing(N))
    {
        SharedSet  = SharedSet Union N;
        MSet = MSet - N;
        Construct new Class N;
        Generate the best plan Q for Gi using N;
        Add Q to Class N;
        Add Class D to the ClassList;
    }
    else
    {
        S' = Ci.NewBaseTable();
        S = Ci.OldBaseTable();
        If ( S' == S)
        {
           Generate the best plan Q for Gi using S;
           Add Q to the Class Ci;
        }
        else
        {
           Generate the new best plan using S'
           for each query in Class Ci;
           SharedSet = SharedSet - S + S';
        }
        MergeClass();
        if S' in other Class Cj ( j != i )
             remove the plan of S' from Class Cj;
    }
}
For each Class in the ClassList
   Merge the query plans;
```

The group-by S' and S may be the same. In this case, we just add the query N to the class without changing the class base table. If $S' = N$, we add the query N to the Class S'. We compute $CostOfAdd(N)$ for a class i using the formula: $CostOfAdd(N) = Cost(Class_i \cup N)_{S'} - Cost(Class_i)_S$. $Cost(X)_Y$ compute the aggregate cost of X using the base table Y.

The return value of $CostOfAdd(N)$ should not be the negative for any class i. If the value is negative, it is equivalent to say $Cost(Class_i)_{S'} < Cost(Class_i \cup N)_{S'} < Cost(Class_i)_S$. This is not possible because we must choose S' over S as the base table for the class i before we process the group-by N.

In the *MergeClass*() procedure, we handle the following cases. If a class has already chosen S' as the base table, then we should merge these two classes together and use group-by S' for the new class's base table. This step prevents the plan from doing repeated I/O on the same table.

The GG algorithm trades the more expensive I/O cost by sharing it with queries for the less expensive CPU cost by processing more tuples for each query. Choosing a non-optimal base table for queries increases the CPU cost for processing more tuples for hash-based star join queries or doing more bitmap operations for index-based star join queries. In the next section, we study how well Global Greedy perform when compared with the global optimal plan.

7 Performance Examples

Doing a performance analysis for optimization techniques is never easy; analyzing the performance of algorithms for optimizing multiple simultaneous dimensional queries is particularly difficult. The primary reason for this is that the problem space is immense. There is the combinatorial effect of considering multiple queries as a unit (instead of the more common optimization task of optimizing individual queries); added to this is yet another combinatorial effect that arises from the exponentially many alternatives for precomputed aggregates that exist in OLAP workloads. For this reason, rather than attempting to claim an exhaustive evaluation of these optimization techniques, in this section we present a number of examples that illustrate that under a fairly realistic setting (measured numbers from an actual implementation rather than a simulation) these techniques can offer substantial performance benefits as compared to naive approaches that do not attempt any simultaneous optimization or evaluation.

7.1 System Configuration

These tests were run on an 200MHZ Intel Pentium Pro processor with 64MB main memory and a 2.0 GB Quantum Fireball disk. The operating system used was Solaris 2.4. We used version 0.5 of Paradise [DKLPY94], configured with an 16MB buffer pool. The test databases were created using the Unix file system on a local disk. We flushed both the Unix file system buffer and Paradise buffer pool before running each test.

7.2 Data Sets

The test database is set up in the following tables. The base table tuple has four dimensional attributes and one measure attribute. Each tuple in the base table has length 20 bytes. The base table tuple has four dimensional attributes and one measure attribute. There are star join bitmap indexes created on attributes A', B' and C' of the group-by $A'B'C'D$. There are four dimensional tables. We have a three level hierarchy along dimension A, B, and C, D. Dimension A has a hierarchy $A \rightarrow A' \rightarrow A''$; Dimension B has the hierarchy $B \rightarrow B' \rightarrow B''$; Dimension C has the hierarchy $C \rightarrow C' \rightarrow C''$. Dimension D has the hierarchy $D \rightarrow D' \rightarrow D''$.

7.3 MDX Queries

We first list of set of basic queries. At the top level of the hierarchy along each dimension A, B, and C, we assume that each of them has three distinct values $A1$,$A2$, $A3$, $B1$, $B2$, $B3$, $C1$, $C2$, and $C3$.

Query 1

```
{A''.A1.CHILDREN} on COLUMNS
{B''.B1} on ROWS
{C''.C1} on PAGES
CONTEXT ABCD FILTER ( D.DD1 );
```

The query asks the sum of $ABCD$ for the group-by $A'B''C''D$ with selection predicates $A' = A''.A1.CHILDREN$ and $B'' = B1$ and $C'' = C1$ and $D = DD1$.

Query 2

```
{A''.A1, A''.A2, A''.A3} on COLUMNS
{B''.B2.CHILDREN} on ROWS
{C''.C2} on PAGES
CONTEXT ABCD  FILTER (D.DD1);
```

The query asks the sum of the table $ABCD$ for the group-by $A''B'C''D$ with selection predicates ($A'' = A1$ or $A'' = A2$ or $A'' = A3$) and $B' = B''.B2.CHILDREN$ and $C'' = C2$ and $D = DD1$. The key word CONTEXT in MDX is equal to FROM in the standard SQL.

Query 3

```
{A''.A2} on COLUMNS
{B''.B2} on ROWS
{C''.C1, C''.C3} on PAGES
CONTEXT ABCD  FILTER (D.DD1);
```

Query 4

```
{A''.A3, A''.A2} on COLUMNS
{B''.B3} on ROWS
{C''.C1, C''.C2, C''.C3 } on PAGES
CONTEXT ABCD FILTER  (D.DD1);
```

Query 5

```
{A''.A1.CHILDREN.AA2} on COLUMNS
{B''.B1} on ROWS
{C''.C3} on PAGES
CONTEXT ABC FILTER (D.DD1)
```

This query asks the sum of $ABCD$ for the group-by $A'B''C''D$ with selection predicates $A' = A''.A1.CHILDREN.AA2$ and $B'' = B1$ and $C'' = C3$ and $D = DD1$.

Query 6

```
{A''.A2.CHILDREN.AA5} on COLUMNS
{B''.B1.CHILDREN} on ROWS
{C''.C3.CHILDREN.CC2} on PAGES
CONTEXT ABCD FILTER (D.DD1)
```

Query 7

```
{A''.A3.CHILDREN.AA2} on COLUMNS
{B''.B2.CHILDREN.BB3} on ROWS
{C''.C1.CHILDREN.CC1} on PAGES
CONTEXT ABCD FILTER (D.DD1)
```

Query 8

```
{A''.A1.CHILDREN.AA2} on COLUMNS
{B''.B2.CHILDREN.BB1} on ROWS
{C''.C1} on PAGES
CONTEXT ABCD FILTER (D.DD1)
```

Query 9

```
{A''.A1.CHILDREN} on COLUMNS
{B''.B2,  B''.B3} on ROWS
{C''.C1.CHILDREN} on PAGES
CONTEXT ABCD FILTER (D.DD1)
```

Query 5 is selective on dimension A and is to compute the group-by $A'B''C''D$; Query 6 and Query 7 are selective on dimension A, B, and C and computes the group-by $A'B'C'D$. Query 8 is selective on dimension A and B and computes the group-by $A'B'C''D$.

7.4 Performance of the Three Shared Operators

We ran three tests to measure the performance of the three shared star join operators.

Materialized Group-by	number of tuples
ABCD	2000000
A'B'C'D	1000000
A'B"C'D	700000
A'B'C"D	750000
A"B'C'D	700000
A"B"C'D	150000

Table 1: Table Size of Materialized Group-bys

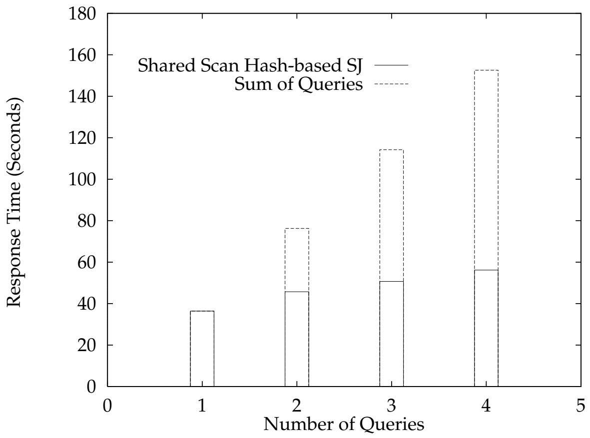

Figure 10: Performance Results for Test 1

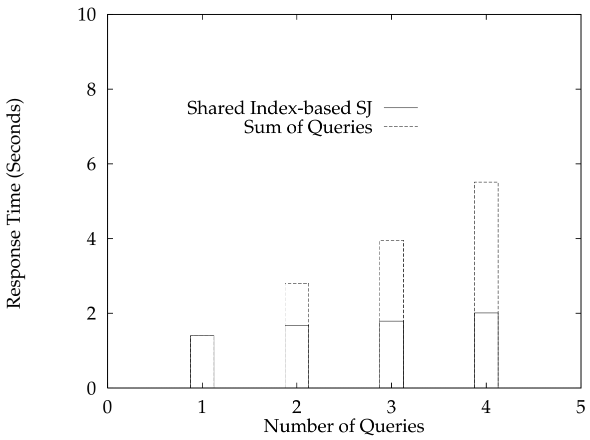

Figure 11: Performance Results for Test 2

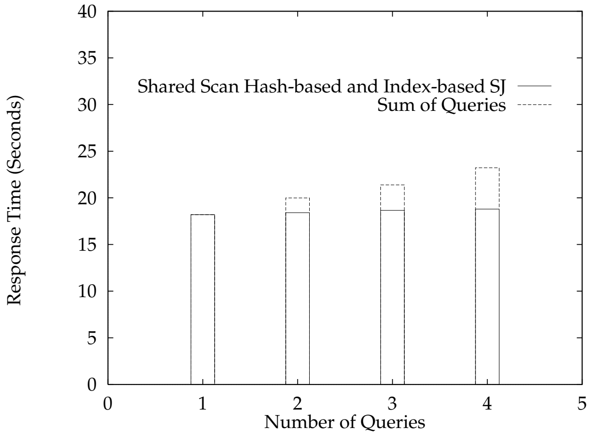

Figure 12: Performance Results for Test 3

Test 1: We ran Queries 1, 2, 3, and 4. We forced each query plan to use hash-based star join using the same base table $ABCD$. The result is shown in Figure 10. The dotted bars represent the total execution time of the queries running separately. The solid bars are the execution time of the queries using the shared scan hash-based star join operator. Each time we add a new query, we increase the CPU cost of the operator, but we save the I/O cost of scanning the base table for the new query. The CPU cost for hash-based star join is not small due to memory copying attributes for result tuples and probing of hash tables for both star join and aggregation.

Test 2: We ran Queries 5, 6, 7, and 8. All the queries use the bitmap index star join method and use the $A'B'C'D$ as the base table. Figure 11 shows the test results. The solid bars are the execution time of Queries 5, 6, 7, and 8. The dotted bars represent the execution time of using shared index-based star join to process the queries. We find that more than 80% of the shared index star join time is spent on probing the base table. When we increase the number of queries from 2 to 4, the probing time changes from 1.651 seconds to 1.969 seconds. Sharing the probing among index-based star join query reduces the aggregated cost dramatically.

Test 3: we use the hash-based star join method for Query 3 and the bitmap index star join method for Query 5, 6, and 7. We use $A'B'C'D$ as the base table for all the queries. The test results are shown in Figure 12. The dotted bars represent the time of execution of queries separately. The solid bars represent the execution time of queries using the shared scan hash-based and index-based star join operator. We see that adding a new index-based query to the operator only increases the total execution time by a small amount. The reason is that the I/O cost of the new index-based query is avoided by sharing the scan of the base table. The CPU cost for each index-based query is very small.

In general it is clear that the sharing operators can have a non-negligible positive effect on overall execution times.

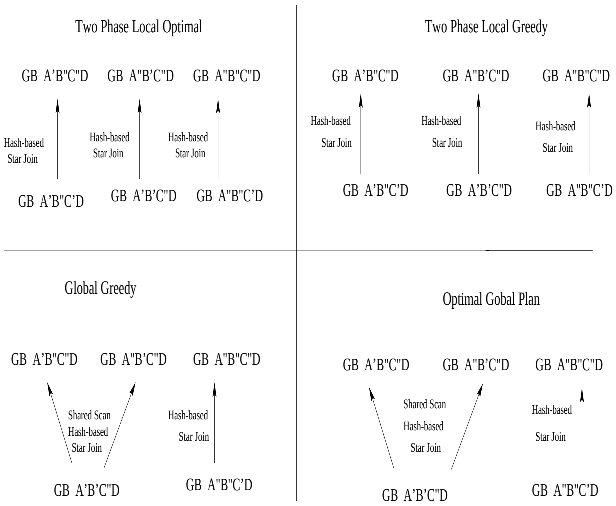

Figure 13: Query Plans for Test 4

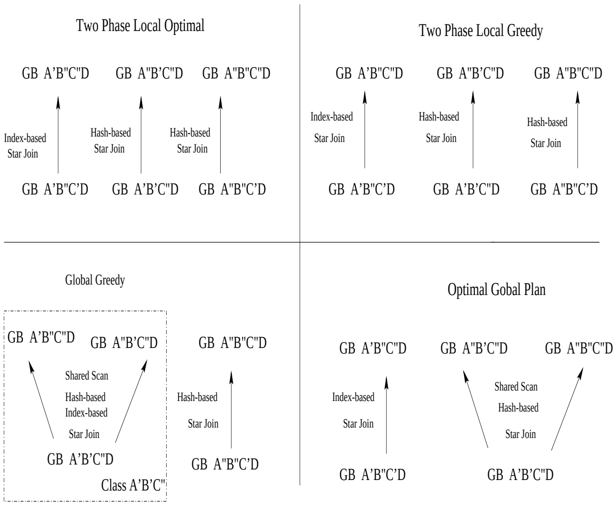

Figure 14: Query Plans for Test 5

Algorithm	Test 4	Test 5	Test 6	Test 7
TPLO	30.866	17.803	1.992	46.627
ETPLG	30.866	17.803	1.992	32.883
GG	19.234	16.561	1.992	32.883
Optimal	19.234	15.367	1.992	32.883

Table 2: Execution time (in seconds) for Test 4, 5, 6 and 7

7.5 Performance of the Three Optimization Algorithms

Test 4: We used all three algorithms: Two Phase Local Optimal (TPLO), Extended Two Phase Local Greedy (ETPLG), and Global Greedy (GG) to generate a global plan for an MDX expression including Queries 1, 2, and 3. The performance results of the global plan produced by the three algorithms and the optimal global plan are listed in Table 2. Both TPLO and ETPLG generate the following query plans $(A'B''C'D \Rightarrow A'B''C''D)_{Hash-basedSJ}$, $(A'B'C''D \Rightarrow A''B'C''D)_{Hash-basedSJ}$, and $(A''B''C'D \Rightarrow A''B''C''D)_{Hash-basedSJ}$. The execution time of the each above query plan is 13.897, 14.405, and 2.564 seconds. ETPLG does not use $A'B'C''D$ to compute $A''B''C'D$ because the CPU cost of computing $A''B''C''D$ from $A'B'C''D$ is higher than the cost of the plan $(A''B''C'D \Rightarrow A''B''C''D)_{Hash-basedSJ}$. The GG algorithm generates the plan $(A'B'C''D \Rightarrow A'B''C''D)_{Shared-HB-SJ}$, $(A'B'C''D \Rightarrow A''B'C''D)_{Shared-HB-SJ}$, and $(A''B''C'D \Rightarrow A''B''C''D)_{Hash-basedSJ}$ The GG algorithm adds each query to the global plan in the order $A'B''C''D$, $A''B'C''D$, and $A''B''C''D$. The total execution time for queries in the Class $A'B'C''D$ is 16.670 seconds and the running time of the queries in the Class $A''B''C'D$ is 2.564 seconds.

Test 5: We used the three algorithms to generate a global plan for Queries 2, 3, and 5. We show the performance results for the global plan produced by the three algorithms and for the optimal global plan in the table below.

Test 5 is different from Test 4. The local optimal plan for the query $A'B''C''D$ is $(A'B'C''D \Rightarrow A'B''C''D)_{Index-basedSJ}$. The TPLO and ETPLG algorithms generate the query plans for Test 5: $(A'B''C'D \Rightarrow A'B''C''D)_{Index-basedSJ}$, and $(A'B'C''D \Rightarrow A''B'C''D)_{Hash-based-SJ}$, $(A''B''C'D \Rightarrow A''B''C''D)_{Hash-basedSJ}$. The execution time of each plan is 1.342, 13.897, and 2.564 seconds. GG generates the plans: $(A'B''C'D \Rightarrow A'B''C''D)_{Shared-HB-IB-SJ}$, and $(A'B'C''D \Rightarrow A''B'C''D)_{Shared-HB-IB-SJ}$, $(A''B''C'D \Rightarrow A''B''C''D)_{Hash-basedSJ}$. The running time for the queries in the Class $A'B'C''D$ is 13.997 seconds. We also have the optimal global plan for Test 5: $(A'B''C'D \Rightarrow A''B'C''D)_{Hash-basedSJ}$, $(A'B'C''D \Rightarrow A''B''C''D)_{Shared-HB-IB-SJ}$, and $(A'B'C''D \Rightarrow A''B'C''D)_{Shared-HB-IB-SJ}$. The running time for the Class $A'B'C''D$ is 14.025 seconds. The query time for the last plan is 1.342 seconds. This optimal plan was found by exploring all possible query plans.

From Test 4 and Test 5, we see that GG produces a global plan which performs close to the optimal global plan. We also observe that GG's global plan performs better than ETPLG for both tests. The simple reason for this is that the GG creates more logical sharing, i.e., base table sharing than ETPLG does. So, the logical sharing of materialized group-bys is important for multiple OLAP query optimization.

Test 6: The MDX expression consists of Queries 6, 7, and 8. We list the performance results in Table 2. Since each of the Queries 6, 7, and 8 are very selective, the best sharing among the queries is at the query tree level i.e. using the shared index-base star join operator. There is not much logical sharing to be found by the ETPLG and GG algorithm. Therefore, the different global plans perform about the same for this situation.

Test 7: The MDX expression contains the Queries 1, 7, and 9. The performance results are shown in Table 2. The GG and ETPLG generate the same global plan as the optimal global plan. All three queries use a hash-based star join. TPLO produces the worst global plan. Since it chooses different fact table for each query, the three queries do not share any common tasks.

From Test 6 and Test 7, we see that GG and ETPLG algorithms generate a better global plan if all the queries are not very selective. On the other hand, if all queries are very selective, GG and ETPLG algorithm do not have much opportunity to create logical level sharing.

8 Conclusion and Future Work

In this paper, we design three new shared star join operators for different star join methods to create sharing among different query plans. Our performance examples show that the three query primitives can substantially improve multiple query performance. To pursue sharing behind the query plan, we also designed and implemented three algorithms for multiple OLAP query optimization. Both ETPLG and GG create logical level sharing by letting the queries share the same base table. We found that, in the most of our tests, the GG algorithm produces a global plan close to the global optimal plan in terms of query performance. The selectivity of a set of related queries also affects the degree of sharing available among query plans. If all queries of an MDX expression are not selective, the optimizer will choose hash-based star join for each query. In this case ETPLG and GG are able to explore opportunities for logical sharing, i.e. sharing base tables among queries. On the other hand, if all the queries of an MDX expression are very selective, which causes the optimizer choose index-based star join for those queries, TPLO produces a fairly good global plan since the local optimal plans are very efficient and logical sharing is not very useful. In this situation, ETPLG and GG would generate the same global plans as TPLO.

The true test of any optimization scheme is how well it works on "real" workloads. As OLE DB for OLAP or a similar standard becomes available to application developers we will have a chance to examine the kinds of simultaneous multiple queries that arise in practice. We regard the results obtained from our implementation as encouraging, suggesting that common subtask sharing and global optimization will yield substantial performance improvements when relational database systems are used as data sources for multidimensional analysis.

An important issue to be explored in the future is the time and space trade-off for the three algorithms TPLO, ETPLG, and GG. In terms of the number of global plans searched, GG dominates ETPLG and ETPLG dominates TPLO. However, this comes at a price - the run time of GG is bigger than that of ETPLG, and ETPLG is slower than TPLO. The study of this trade-off may lead to the discovery of new algorithms that have both better time and space performance than the above three algorithms.

Acknowledgements We would like to thank Rick Stellwagen who first lead us to attack this problem. We also like to thank the anonymous referees for helpful suggestions.

References

[CS94] S. Chaudhuri and K. Shim. "Including group-by in query optimization". In VLDB Conference, page 354-366, 1994.

[CR96] Damianos Chatziantoniou, Kenneth A. Ross. Querying Multiple Features of Groups in Relational Databases". Multidimensional Aggregates". In *Proceedings of the 22nd International Conference on Very Large Databases*, Mumbai (Bombay), pp295-306.

[DKLPY94] D.J. DeWitt, N. Kabra, J. Luo, J.M. Patel, J. Yu. "Client-Server Paradise." Proceedings of the 20th VLDB Conference, Santiago, Chile, 1994.

[HRU96] V. Harinarayan, A. Rajaraman, and J.D. Ullman. " Implementing Data Cubes Efficiently", Proc. ACM SIGMOD '96.

[GHQ95] A. Gupta, V Harinarayan, D. Quass. "Aggregate-Query Processing in Data Warehousing Environments", Proceedings of the 21st VLDB Conference Zurich, Swizerland, 1995.

[MS] Microsoft Corporated. "OLE DB for OLAP Design Specification – Beta 2". **http://www.microsoft.com/data/oledb/olap/prodinfo.html**

[OQ97] Patrick O'Neil and Dallan Quass. "Improved Query Performance with Variant Indexes." Proc. of the 1997 SIGMOD Conference, May, 1997.

[PS88] J. Park and A Segev "Using common subexpressions to optimize multiple queries". In Proc. 4th Intern. Conf. on Data Engineering, pages 311-319, February, 1988.

[S88] Timos K. Sellis. "Multiple-Query Optimization". ACM Transactions on Database Systems, Vol 13, No.1, March 1988, Pages 23-52.

[SS94] K. Shim and T.Sellis, "Improvements on a Heuristic Algorithm for Multiple- Query Optimization", Data and Knowledge Engineering, Vol. 12, No.2, March 1994.

[SM94] Sunita Sarawagi, Michael Stonebraker, "Efficient Organization of Large Multidimensional Arrays". In *Proceedings of the Eleventh International Conference on Data Engineering*, Houston, TX, February 1994.

[Su96] Prakash Sundaresan. "Data Warehousing Features in Informix OnLine XPS." Presentation at the Fourth International PDIS Conference, December 18-20, 1996, Miami Beach, Florida.

[YL95] W. P. Yan and P. Larson. "Eager aggregation and lazy aggregation". In VLDB Conference, page 345-357, 1995.

[ZTN96] Y.H. Zhao, K. Tufte, and J.F. Naughton. "On the Performance of an Array-Based ADT for OLAP Workloads". Technical Report CS-TR-96-1313, University of Wisconsin-Madison, CS Department, May 1996.