Efficient Computation of Multiple Group By Queries

Zhimin Chen   Vivek Narasayya
Microsoft Research
{zmchen, viveknar}@microsoft.com

ABSTRACT

Data analysts need to understand the quality of data in the warehouse. This is often done by issuing many Group By queries on the sets of columns of interest. Since the volume of data in these warehouses can be large, and tables in a data warehouse often contain many columns, this analysis typically requires executing a large number of Group By queries, which can be expensive. We show that the performance of today's database systems for such data analysis is inadequate. We also show that the problem is computationally hard, and develop efficient techniques for solving it. We demonstrate significant speedup over existing approaches on today's commercial database systems.

1. INTRODUCTION

Decision support analysis on data warehouses influences important business decisions and hence the accuracy of such analysis is crucial. Understanding the quality of data is an important requirement for a data analyst [18]. For instance, if the number of distinct values in the State column of a relation describing customers within USA is more than 50, this could indicate a potential problem with data quality. Other examples include the percentage of missing (NULL) values in a column, the maximum and minimum values etc. Such aggregates help a data analyst evaluate whether the data satisfies the expected norm. This kind of data analysis can be viewed as obtaining frequency distributions over a set of columns of a relation; i.e., a Group By query of the form: SELECT X, COUNT(*) FROM R GROUP BY X, where X is a *set of columns* on relation R. Note that X may sometimes contain derived columns, e.g., LEN(*c*) for computing the length distribution of a column *c*. For example, consider a relation *Customer*(Lastname, FirstName, M.I., Gender, Address, City, State, Zip, Country). A typical scenario is understanding the distributions of values of each column, which requires issuing several Group By queries, one per column. In addition, it is sometimes necessary to also understand the joint frequency distributions of sets of columns, e.g., the analyst may expect that (LastName, FirstName, M.I., Zip) is a key (or almost a key) for that relation. We note that some data mining applications [15] also have similar requirements of computing frequency distributions over many sets of columns of a relation.

This kind of data analysis can be time consuming and resource intensive for two reasons. First the number of rows in the relation can be large. Second, the number of *sets of columns* over which Group By queries are required can also be large. A naïve approach is to execute a different Group By query for each set of columns.

SIGMOD 2005 June 14-16, 2005, Baltimore, Maryland, USA.
Copyright 2005 ACM 1-59593-060-4/05/06 $5.00.

In some commercial database systems, an extension to the GROUP BY construct called GROUPING SETS is available, which allows the computation of multiple Group By queries using a single SQL statement. The multiple Group By queries are specified by providing *a set of column sets*. For example specifying GROUP BY GROUPING SETS ((A), (B), (C), (A,C)) would cause the Union All of all four Group By queries to be returned in the result set. The query optimizer has the ability to optimize the execution of the set of queries by potentially taking advantage of the commonality among them. However, as the following experiment on a commercial database system shows, GROUPING SETS functionality is not optimized for the kinds of data analysis scenarios discussed above.

Example 1. Consider a scenario where the data analyst wishes to obtain single-column value distribution for each character or categorical column in a relation. For example, there are 12 such columns in the *lineitem* table of the TPC-H database (1GB) [24]. We used a single GROUPING SETS query on a commercial database system to compute results of these 12 Group By queries. We compared the time taken to execute the GROUPING SETS query with the time taken using the approach presented in this paper. Using our approach was about 4.5 times faster.

The explanation for this is that GROUPING SETS is not optimized for scenarios where many column sets with little overlap among them are requested, which is a common data analysis scenario. In Example 1, where we request 12 single column sets only, the plan picked by the query optimizer is to first compute the Group By of all 12 columns, materialize that result, and then compute each of the 12 Group By queries from that materialized result. It is easy to see that such a strategy is almost the same as the naïve approach of computing each of the 12 Group By queries from the *lineitem* table itself. In contrast, our solution consisted of materializing the results of the following Group By queries into temporary tables: (1) (l_receipdate, l_commitdate) (2) (l_tax, l_discount, l_quantity, l_returnflag, l_linestatus); and computing the single column Group By queries of these columns from the respective temporary tables. The Group By queries for each of the remaining columns were computed directly from the *lineitem* table.

An important observation, illustrated by the above example, is that materializing results of queries, including *queries that are not required*, can speed up the overall execution of the required queries. It is easy to see that the *search space*, i.e., the space of queries that are not required, but whose results, if materialized, could speed up execution of the required queries, is very large. For example, for a relation with 30 columns, if we want to compute all single column Group By queries, the entire space of relevant Group By queries to consider for materialization is 2^{30}.

We note that previous solutions to the above problem, including GROUPING SETS, were designed for efficient processing of the (partial) datacube e.g., [2,14,16,21], as a generalization of CUBE and ROLLUP queries. Unlike the kind of data analysis scenarios presented above, these solutions were geared towards OLAP like scenarios with a small set of Group By queries typically having significant overlap in the sets of columns. Hence, most of these previous solutions to this problem assume that search space of queries can be *fully enumerated* as a first step in the optimization. Thus, these solutions do not scale well to the kinds of data analysis scenario described earlier.

Our key ideas and contributions can be summarized as follows. First, we show that even a simple case of this problem, where we need to compute *n* single-column Group By queries of a relation is computationally hard. Second, unlike previous solutions to this problem, our algorithm explores the search space in a bottom up manner using a hill-climbing approach. Consequently, in most cases, we do not require enumerating the entire search space for optimizing the required queries. This allows us to scale well to the kinds of data analysis scenarios mentioned above. Third, our algorithm is cost based and we show how this technique can be incorporated into today's commercial database systems for optimizing a GROUPING SETS query. As an alternative, we show how today's client data analysis tools/applications can take advantage of our techniques to achieve better performance until such time that GROUPING SETS queries are better optimized for such scenarios. Finally, we have conducted an experimental evaluation of our solution on real world datasets against commercial database systems. These experiments indicate that very significant speedups are possible using our techniques.

The rest of this paper is structured as follows. In Section 2, we discuss related work. Section 3 formally defines the problem and establishes the hardness of the problem. We present our algorithm for solving this problem and analyze key properties of the algorithm in Section 4. In Section 5 we discuss how our solution can be implemented both inside the server for optimizing a GROUPING SETS query as well as on the client side (i.e., using just SQL queries). Section 6 presents results of our experimental evaluation and Section 7 discuses extensions to the basic solution to handle a broader class of scenarios. We conclude in Section 8.

2. RELATED WORK

Our problem can be viewed as a multi-query optimization problem for Group By queries. The first body of related work proposes techniques for efficiently computing a (partial) data cube. The GROUPING SETS support in commercial database systems such as IBM DB2 and Oracle provide exactly the required functionality, i.e., the ability to compute the results of a set of Group By queries. However, as discussed in the introduction, the performance of GROUPING SETS does not scale well if a large number of non-overlapping Group By queries is provided as input, which is the kind of data analysis scenarios we described earlier. The work that is closest to ours is the work on partial CUBE computation [14,16]. This work proposes a technique for materializing new nodes (Group By queries) to help reduce the cost of computing other queries. A key difference is that their solution (which is an approximation algorithm to the Minimum Steiner Tree problem), assumes that the search lattice has been constructed. However, it is easy to see that if we need to compute all single column Group By queries of a table with many columns, the above solution does not scale, since the step of constructing the search lattice itself would not be feasible. In contrast, our approach constructs only a small part of the lattice in a bottom-up manner, interleaved with the search algorithm (Section 4). This allows us to scale to a much larger inputs. We note that once the set of queries to be materialized is determined, several *physical operators*, e.g., PipeHash, PipeSort [14,16], Partitioned-cube, Memory-Cube [16], etc. for efficiently executing a *given set of queries* (without introducing any new queries) can be used. This problem is orthogonal to ours, and these techniques can therefore be leveraged by our solution as well. The basic ideas is to take advantage of commonality across Group By queries using techniques such as shared scans, shared sorts, and shared partitions [2,8,15,16,21]. We note that work on efficient computation of a CUBE (respectively ROLLUP) query is a special case of the partial cube computation discussed above where the *entire* lattice (resp. a hierarchy) of queries needs to be computed. In contrast our work can be thought of as introducing logically equivalent rewritings of a GROUPING SETS query in a cost based manner.

The second category of work is creation of materialized views for answering queries. The paper [17] studies the problem of how to speed up update/creation of a materialized view V by creating additional materialized views. They achieve this by creating a DAG over logical operators by considering various equivalent expressions for V (e.g., those generated by the query optimizer); and then determining an optimal set of additional views (sub-trees of the DAG) to materialize. Similarly, the work in [10,11] explores the problem of which views to materialize in a data warehouse to optimize the (weighted) sum of the cost of answering a set of queries as well as the cost of materializing the views themselves. However, the solution assumes that the graph G (a DAG) containing all possible candidate views to materialize is provided as input. In both the above studies, the proposed solution requires the DAG to be constructed *prior* to optimization. Once again, this approach does not scale for our scenario since the size of the DAG is exponential in the number of input queries.

The idea of introducing a new query/view to answer a given set of queries (used in this paper) is not novel, and has been proposed in several contexts before [2,10,17]; including the ideas of: (a) merging views in [3,25] as well as building a common subsumer expression in [9]. However, in [3,14,25], the goal is to pick a set of materialized views to optimize the cost of executing a given set of queries, and the cost of materializing the chosen views is not part of the objective function. Thus a naïve adaptation to our problem is not possible. In [9], the exact space of common subsumer views that would be considered for a given set of Group By queries is not defined.

In [15], a modification to GROUPING SETS syntax is proposed (called the Combi operator) to allow easy specification of a large number of Group By queries as input (e.g., all subsets of columns up to size *k*). Such a syntactic extension would be useful for the kinds of data analysis scenarios presented in this paper where e.g., all single-column and two-column Group By queries over a relation are required. Finally, there has been work in multi-query optimization for join queries e.g., [19]. In these papers, the focus is on reusing results of join expressions, unlike our case, where we focus on sharing work done by different Group By queries referencing the same single table/view.

3. PROBLEM STATEMENT and HARDNESS

3.1 Definitions

We assume a relation R(c_1, … c_m) with *m* columns. Let C be the set of columns in R, i.e., C = {c_1, …c_m}. Let S = {s_1, s_2, … s_n}, where each s_i is a subset of columns of R, represent a *set of Group By queries* over R. Thus each s_i is a Group By query:

SELECT s_i COUNT(*) FROM R GROUP BY s_i

In the rest of this paper, we assume that all queries require only the COUNT(*) aggregate. In Section 7 we discuss extensions when different aggregates are needed by different queries.

Search DAG

Suppose we are given a relation R, and a set S = {s_1, s_2, … s_n} of Group By queries over R. Let G = (V, E) be a *directed acyclic graph* (DAG) that is defined as follows. A node in the graph corresponds to a Group By query. The set of nodes V contains all elements of the power set of $s_1 \cup s_2 \cup \ldots s_n$,. Note that s_1, s_2, … s_n themselves will be nodes in the graph. We refer to the nodes in S as *required* nodes, since we are required to produce the results for these nodes. The edge set E contains a directed edge from node *u* to *v* iff, $u \supset v$. We refer to *u* as the ancestor of *v*, and *v* as the descendant of *u*. In addition, there is one distinguished node called the *root* node, which represents the relation R itself. The root node has an outgoing edge to every other node in V (since it is an ancestor of every other node). We call G the *Search DAG*.

Example 2. Suppose R(A,B,C,D), and S = {(A), (B), (C), (A,C)}. The search DAG for this example is shown in Figure 1.

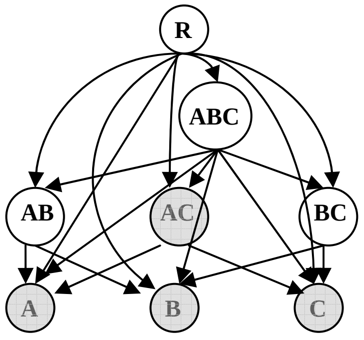


Figure 1. A search DAG for the input {(A), (B), (C), (AC)}

Logical plan for computing a set of Group By queries

Let **P** be a *logical plan* for computing S, i.e., for computing all queries s_1, … s_n. **P** is a directed *tree* over the Search DAG, rooted at R, and including all required nodes. This tree can also be viewed as a partial order of SQL queries. An edge from node *u* -> *v* in the tree means that *v* is computed as a Group By query over the table *u*. Note that if $u \neq R$, (i.e., *u* is an intermediate node in the tree) then *u* will need to be materialized as a temporary table before *v* can be computed from it. Our notion of a logical plan is different from the notion of physical execution plan used in a query optimizer since the "operators" in our plan are SQL Group By queries rather than physical operators.

Figure 2 shows two different logical plans for the input S = {(A), (B), (C), (AC)}. The required nodes are shaded. In logical plan $\mathbf{P_1}$, all the required nodes are computed from the root node, i.e., base relation R. In plan $\mathbf{P_2}$, (A,B) is computed from R, its results are materialized, and both (A) and (B) are computed from it. Likewise, (A,C) is computed directly from R , its results materialized, and (C) is computed from the results of (A,C).

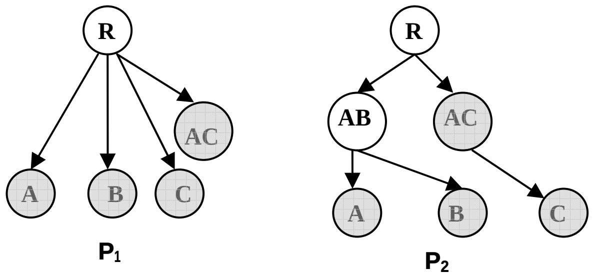


Figure 2. Two logical plans $\mathbf{P_1}$ and $\mathbf{P_2}$ for the input S = {(A),(B),(C),(AC)}

A *sub-plan* is a sub-tree of a logical plan *whose root node is directly pointed to by R*. For example, in Figure 2 in $\mathbf{P_2}$, the sub-trees rooted at (AB) and (AC) are both sub-plans. But the sub-tree rooted at (A) is not a sub-plan since its parent is not R.

Finally, given a logical plan (a tree), there is the issue of how much additional storage (disk space) is consumed for materializing the intermediate nodes of the tree. Note that depending on the *sequence* (i.e., depth first or breadth first) in which the queries in the logical plan are executed, the maximum storage consumed by that plan may be different. In Section 4.4 we show how to sequence the execution of queries in the tree so that at any point during the execution of the logical plan, the storage taken up by intermediate nodes is as small as possible.

3.2 Cost Model

Our goal is to efficiently compute all Group By queries in S, i.e., find an efficient logical plan for S. Therefore we now discuss the issue of what metric to use to compare two different logical plans. We note that our techniques do not depend on the specific cost model used, although they do rely on the ability to compute a cost for a given logical plan (or sub-plan) **P**. In this paper we consider two cost models. We denote the cost of a plan **P** as Cost(**P**).

3.2.1 Cardinality Cost Model

This cost model assumes that the cost of an edge from *u* -> *v* in the Search DAG is the number of rows of the table *u*, denoted by |*u*|. Intuitively, this simple cost model captures the cost of scanning the relation *u*, which is often a reasonable indicator of the cost, particularly when there are no (relevant) indexes on the table *u*. Of course, in general, this cost model can be inaccurate -- e.g., it does not capture how "wide" the relation *u* is; it does not account for presence of indices on *u* etc. However, due to its simplicity, it is more amenable for use in analyzing the problem and its solutions, and has been used in previous work related to this problem e.g., [10,14].

We note that to use the Cardinality cost model, we still need to be able to estimate the cardinality of a Group By query, which is a hard problem. For this, we assume that known techniques for estimating number of distinct values such as [13] may be used.

3.2.2 Query Optimizer Cost Model

This cost model takes advantage of the fact that the logical plan is, in fact, a set of SQL queries. Thus, we can use the *query* *optimizer* of the DBMS itself (which is capable of estimating the cost of an individual query) as the basis of the cost model. More precisely, we model Cost (**P**) as the sum of the *optimizer estimated cost* of each SQL query in **P.** This cost model has a couple of advantages over the Cardinality cost model (Section 3.2.1). First it captures the effects of the current physical design in the database. For example, if a query can take advantage of an existing index in the database, then this effect will automatically be reflected in the optimizer estimated cost. Second, cost models used by the optimizer in today's database systems are already quite sophisticated, and are able to take advantage of database statistics (e.g., histograms, distinct value estimates) for producing accurate estimates for many cases.

We note that in order to be able to use this cost model, we need the ability to cost a query such as *u* -> *v* when *u* is not the base relation R, i.e., *u* does not actually exist as a table in the database. For this purpose, we take advantage of the capabilities of "what-if" analysis APIs in today's commercial query optimizers. These APIs allow us to pretend (as far as the query optimizer is concerned) that a table exists, and has a given cardinality and database statistics. Details of these APIs vary with each commercial system and can be found e.g., in [5,25].

The accuracy of this cost model is a function of the available database statistics. Collecting statistics can be expensive; however, the optimizer can create multiple statistics from one sample, and the cost of creating a statistics can be amortized because the same statistics can be used to optimize many other queries. Even we ignored the effect of amortization, the experiments in Section 6.7 suggests that for large data sets, the time to create statistics is only a small fraction compared to the savings in running time resulting from a better logical plan.

Finally, we note that the cost of materializing a temporary table can also be handled in this model in a straightforward way. For a query *u* -> *v*, where *v* needs to be materialized, we construct the query as a SELECT … INTO *v* … (or equivalently INSERT INTO *v* SELECT …), which can also be submitted to the query optimizer for cost estimation.

3.3 Problem Statement

Given a relation R, and a set of Group By queries on R denoted by S = {s_1,… s_n} find the logical plan for S having the lowest cost, i.e., find a logical plan **P** that minimizes Cost (**P**).

We refer to the problem in the rest of this paper as the Group By Multi-Query Optimization problem or GB-MQO for short. In general, any cost function Cost (**P**) may be used to measure the cost of an execution plan.

3.4 Hardness Results

It has previously been noted [2] that the GB-MQO problem is NP-Complete when the cost function can take on *arbitrary values*. The reduction is from the Minimum Steiner Tree problem over DAGs. In this paper, we show that even a very special case of the problem, namely the problem of computing even the *single column* Group By queries over a relation is also NP-Complete. This result is pertinent since as discussed in the introduction, this special case is a common occurrence in data analysis for understanding data quality or in certain data mining applications.

Claim: The GB-MQO problem is NP-Complete even if we restrict the input set S to include only single column Group By queries and use the Cardinality cost model (Section 3.2.1).

Proof: The reduction is from the problem of finding the optimal bushy execution plan for a cross product query over N relations [22]. See **Appendix A** for details.

4. SOLUTION

In Section 3 we defined the Group By Multi-Query Optimization (GB-MQO) problem, where the goal is to find a logical plan (a partial order to execute SQL queries) such that the cost of the plan (as determined by a cost model) is minimized. Given the hardness results presented in Section 3.4, it is unlikely that an efficient algorithm exists that can compute an exact solution to this problem. Moreover, it is important to note that even if efficient *approximation* algorithms exist for our problem (e.g., by adapting solutions to the Minimum Steiner tree problems), the input to such a solution would be the Search DAG (see Section 3.1). This would require constructing the Search DAG, whose size is exponential relative to the size of the input set of Group By queries S. In this section, we therefore present a heuristic solution to the above problem that scales well with the size of the input set S, while still producing a good logical plan as output.

Our algorithm starts with the "naïve" logical plan where each $s_i \in$ S is computed directly from R, and uses the idea of combining two *sub-plans* (see Section 3.1) **P**$_1$ and **P**$_2$ as the basic operator for traversing the space of logical plans. In Section 4.1, we present the operator for combining two logical sub-plans. In Section 4.2, we present and analyze our overall algorithm. In Section 4.3, we present two pruning techniques that significantly speed up our algorithm in practice. In Section 4.4, we present that given a logical plan, how to execute it to minimize the storage taken up by the intermediate nodes.

4.1 Merging Two Sub-Plans

We use the idea of merging two logical sub-plans **P**$_1$ and **P**$_2$ as the basic operation for generating new logical sub-plans as described below. We refer to this operator as the *SubPlanMerge* operator. The *SubPlanMerge* operator (described below) has the desirable property that the root node of each new sub-plan output by the operator is the node with minimal cardinality from which the sub-plans **P**$_1$ and **P**$_2$ can be computed.

Consider a pair of sub-plans **P**$_1$ and **P**$_2$ as shown in Figure 3. The set of new sub-plans introduced by merging **P**$_1$ and **P**$_2$ is shown in Figures 4 (a)-(d). We refer to the merging operator as SubPlanMerge(**P**$_1$, **P**$_2$) and it returns a set of sub-plans as output. Note that in each case, the root node of the new sub-plan is $v_1 \cup v_2$, which is the smallest relation from which *both* v_1 and v_2 can be computed. For example, if v_1 is (A,B) and v_2 is (A,C), then $v_1 \cup v_2$ is (A,B,C).

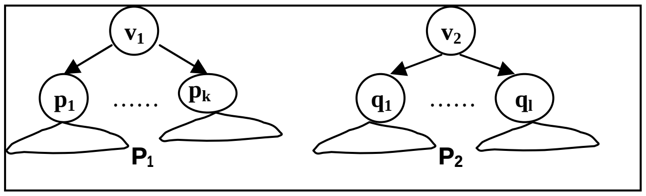


Figure 3. Two sub-plans P$_1$ and P$_2$.

Figure 4(a) creates a sub-plan where the children of v_1 and v_2 are computed directly from the parent, thereby avoiding the cost of computing and materializing *both* v_1 and v_2. Of course, this sub-plan is only generated when neither v_1 nor v_2 is a required node. On the other hand Figure 4 (b) creates a plan where both v_1 and v_2 are computed and materialized. This plan can be considered whether or not v_1 and v_2 are required nodes. Intuitively, the sub-plan (a) can be good, when the size of $v_1 \cup v_2$ is not much larger than the size of v_1 or v_2, whereas (b) can be good when the size of $v_1 \cup v_2$ is much larger than the size of v_1 and v_2. The former is more likely when the values of v_1 and v_2 are highly correlated, whereas the latter is more likely when v_1 and v_2 are independent.

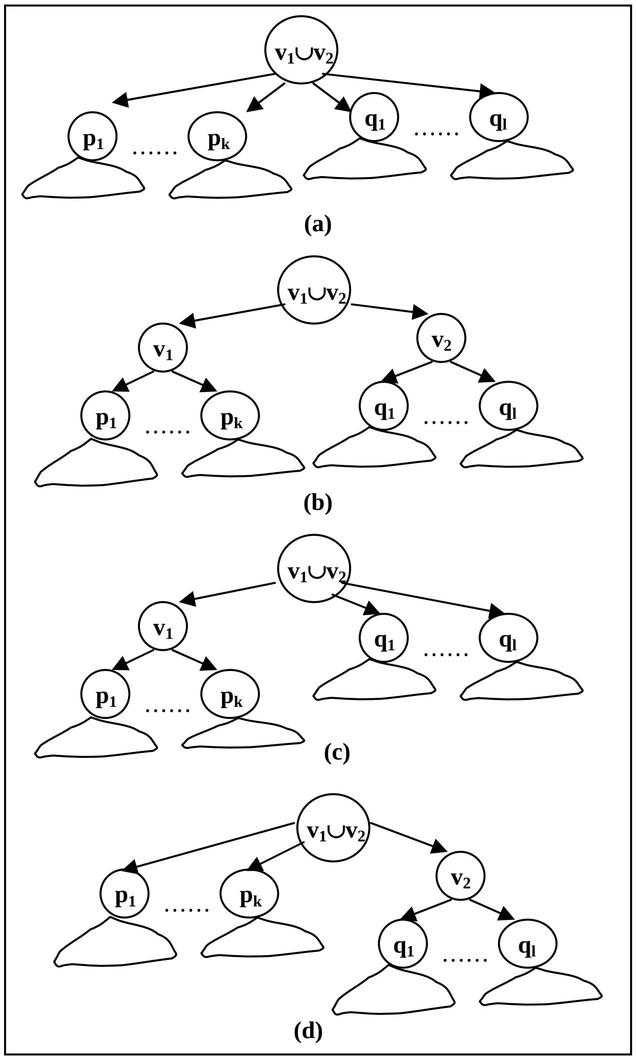


Figure 4. Space of new sub-plans introduced by *SubPlanMerge* operator

Sub-plans (c) and (d) can be beneficial when either one (but not both) of v_1 and v_2 are much smaller than $v_1 \cup v_2$. Thus for example if $v_1 \cup v_2$ has only a slightly higher size than v_2 but significantly higher size than v_1, then in sub-plan (c) could be the best plan. This is because although sub-plans $q_1 \ldots q_l$ will incur a higher cost since they are now computed from $v_1 \cup v_2$ instead of from v_2, the increased cost may be more than offset by the reduced cost of not computing and materializing v_2. On the other hand, for sub-plans $p_1 \ldots p_k$ it may be more beneficial to compute them from v_1 even after paying the cost of computing and materializing it.

If there is a subsumption relationship between v_1 and v_2 (which is common in practice), (b) (c) and (d) in Figure 4 degenerate into one case in which we compute v_2 from v_1, and then compute $q_1 \ldots q_l$ from v_2 (assuming $v_2 \subseteq v_1$).

4.2 Algorithm for the GB-MQO problem

Our algorithm for computing the logical plan for a given input set $S = \{s_1, \ldots s_n\}$ on a relation R is shown in Figure 5. The algorithm starts with the "naïve" plan where each s_i is computed directly from R and improves upon the solution until it reaches a local minimum. Observe that unlike previous algorithms for this problem [2,10,14,21], our algorithm does not require the Search DAG as input. Instead it constructs logical plans in a bottom-up manner. This allows it to scale for large input sizes, e.g., for the common case of computing all single column Group By queries over a relation with many columns.

```
1.  Let P be the naïve plan, i.e., where each $s_i \in S$ is a
    sub-plan, i.e., computed directly from relation R.
2.  Let C = Cost(S, P).
3.  Do
4.    Let MP = Set of all plans obtained by invoking
      SubPlanMerge on each pair of sub-plans in P.
5.    Let P' be the lowest cost plan in MP, with cost C'.
6.    BetterPlanFound = False
7.    If C' < C Then
8.      P = P'; C = C'; BetterPlanFound = True
9.    End If
10. While (BetterPlanFound)
11. Return P
```

Figure 5. Algorithm for finding a logical plan for a given set S of Group By queries.

Analysis of Running Time: Let $|S| = n$. A naïve implementation of the above algorithm would call *SubPlanMerge* $O(n^3)$ times, since each iteration of the loop does $O(n^2)$ merges and there are $O(n)$ iterations in the worst case. However, it is easy to see that if we are willing to store the logical sub-plans that have been computed in previous iterations (which incurs a memory cost of storing $O(n^2)$ sub-plans), then the algorithm needs to call *SubPlanMerge* only $O(n^2)$ times. This makes the algorithm scalable both in terms of running time as well as memory requirements for relatively large input sizes (e.g., up to 100's of Group By queries).

Restriction of search space to binary trees only: Given the above algorithm and the four different kinds of sub-plans returned by *SubPlanMerge*, it is easy to see that in general, the shape of the logical plan can be an arbitrary tree. As discussed earlier, an important special case of data analysis scenarios is computing all single column Group By queries. This special case has the property that all inputs are non-overlapping, i.e., no pair of inputs have any columns in common, e.g., S = {(A),(B),(C),(D)}. In this case, we have observed empirically (see Section 6.5) that restricting the search space to only binary trees can save a considerable amount of optimization time and still give very good logical plans in practice. This restriction is analogous to the restricted search space of left deep trees considered by many query optimizers today. Note that since our algorithm only considers merging pairs of sub-plans in any step, for the above case, restriction of the space of logical plans to binary trees can be done by limiting the *SubPlanMerge* operator to produce only plans of type (b).

4.3 Pruning Techniques

In this section, we present two techniques for pruning the space of execution plans considered by our algorithm for the GB-MQO problem (Section 4.2). These two pruning techniques are developed under the cardinality cost model and with the restriction of applying *SubPlanMerge* type (b) in Figure 4 only, and hence are heuristics for the general case; however, they appear to perform well in practice across several real data sets that we have tried in our experiments (Section 6.6). These pruning techniques significantly reduce the running time of the algorithm with relatively small impact on the cost of the logical plan produced.

4.3.1 Subsumption Based Pruning

In the algorithm presented in Section 4.2, we consider merging any two pairs of sub-plans in Step 4. The pruning technique is based on the intuition that it should be less expensive to compute Group Bys from a closer ancestor: in step 4 of the algorithm, given two sub-plans rooted at two nodes v_i and v_j, if there are any two sub-plans rooted at v_x and v_y such that $(v_i \cup v_j) \supset (v_x \cup v_y)$, then do not consider merging v_i and v_j, i.e., do not consider the sub-plan rooted at $(v_i \cup v_j)$. For instance, if there are 3 sub-plans rooted at (A,B), (B,C) and (C,D) respectively, then do not consider merging sub-plans rooted at (A,B) and (C,D) because $(A,B)\cup(C,D) \supset (A,B)\cup(B,C)$.

Claim: Suppose v_i and v_j are non-overlapping and so are v_x and v_y. The above pruning technique is sound (i.e., the answer found by our algorithm is not affected) when using the Cardinality cost model and restricting *SubPlanMerge* to type (b) in Figure 4.

Proof: Let us denote the cost of sub-plan T_u rooted at node *u* as Cost(*u*). Let Child(*u*) denote the set of children of *u*. Let R denote the base relation. So Cost(*u*)+Cost(*v*)-cost(*u* ∪ *v*) is the benefit of merging two sub-plans rooted at *u* and *v*. For the pruning technique to be sound, we need to show that: $\text{Cost}(v_x)+\text{Cost}(v_y)-\text{Cost}(v_x \cup v_y) \geq \text{Cost}(v_i)+\text{Cost}(v_j)-\text{Cost}(v_i \cup v_j)$, since this would ensure that the algorithm would pick merging of v_x and v_y over merging of v_i and v_j. Using the cardinality cost model we know that:

$$\text{Cost }(v_i)+\text{Cost }(v_j)= 2|R|+ \textstyle\sum_{c \in \text{Child}(vi)} \text{Cost}(c)+ \sum_{c \in \text{child}(vj)} \text{Cost}(c)$$

$$\text{Cost }(v_i \cup v_j)=|R|+2|v_i \cup v_j| + \textstyle\sum_{c\in \text{Child}(vi)} \text{Cost}(c)+ \sum_{c\in \text{child}(vj)} \text{Cost}(c)$$

Thus, $\text{Cost }(v_i)+\text{Cost }(v_j)-\text{Cost }(v_i \cup v_j)=|R|-2|v_i \cup v_j|$ ……….(1)

Likewise, $\text{Cost }(v_x)+\text{Cost }(v_y)-\text{Cost }(v_x \cup v_y)=|R|-2|v_x \cup v_y|$ …(2)

Since $(v_i \cup v_j) \supset (v_x \cup v_y)$ we know that $|v_i \cup v_j| \geq |v_x \cup v_y|$. Thus (2) ≥(1), which proves the claim.

4.3.2 Monotonicity Based Pruning

This pruning technique is based on the same intuition as the pruning technique used in the Apriori algorithm for frequent itemset generation [1]. Given two sub-plans rooted at v_i and v_j if *SubPlanMerge*(v_i,v_j) does not result in a lower cost sub-plan, then do not consider merging any two sub-plans v_x and v_y such that $v_i \cup v_j \subseteq v_x \cup v_y$. For instance, if *SubPlanMerge*((A), (B)) does not result in a lower cost sub-plan, then later on do not consider merging sub-plans like (A,C) and (B,D). Since our algorithm enumerates the plans in a bottom-up manner, we can take advantage of the above technique to prune out a potentially large space of plans from consideration, with the cost of storing the pair of sub-plans that cannot be merged in an n×n array.

Claim: Suppose v_i and v_j are non-overlapping and so are v_x and v_y. The above pruning technique is sound when using the Cardinality cost model and restricting *SubPlanMerge* to type (b) in Figure. 4.

Proof: We need to show that given two sub-plans rooted at v_i and v_j, if $\text{Cost}(v_i) + \text{Cost}(v_j) \leq \text{Cost }(v_i \cup v_j)$, then for any sub-plans rooted at v_x and v_y such that $(v_i \cup v_j) \subseteq (v_x \cup v_y)$, then $\text{Cost}(v_x) + \text{Cost}(v_y) \leq \text{Cost}(v_x \cup v_y)$.

From the cardinality cost model, it follows that:

$$\text{Cost }(v_i)+\text{Cost }(v_j)= 2|R|+ \textstyle\sum_{c \in \text{Child}(vi)} \text{Cost}(c)+ \sum_{c \in \text{child}(vj)} \text{Cost}(c)$$

$$\text{Cost}(v_i \cup v_j)=|R|+2|v_i \cup v_j| +\textstyle\sum_{c \in \text{Child}(vi)}\text{Cost}(c)+\sum_{c \in \text{child}(vj)} \text{Cost}(c)$$

Since $\text{Cost }(v_i) + \text{Cost }(v_j) \leq \text{Cost }(v_i \cup v_j)$, it follows that $|R| \leq 2|v_i \cup v_j|$. Furthermore, since $v_i \cup v_j \subseteq v_x \cup v_y$, we know that

$|v_i \cup v_j| \leq |v_x \cup v_y|$. Thus, $|R| \leq 2|v_x \cup v_y|$ ………………… (1)

Likewise, we have

$$\text{Cost}(v_x)+\text{Cost}(v_y)=2|R|+\textstyle\sum_{c\in \text{Child}(vx)}\text{Cost}(c)+\sum_{c\in \text{child}(vy)}\text{Cost}(c) \quad ..(2)$$

$$\text{Cost}(v_x\cup v_y)=|R|+2|v_x\cup v_y|+\textstyle\sum_{c\in \text{Child}(vx)}\text{Cost}(c)+\sum_{c\in \text{child}(vy)}\text{Cost}(c) \quad (3)$$

Comparing the expanded terms (2) and (3), and using the result from (1), we get: $\text{Cost}(v_x) + \text{Cost}(v_y) \leq \text{Cost}(v_x \cup v_y)$. QED.

4.4 Intermediate Storage Considerations

Each node in the logical plan corresponds to a SQL Group By query, and for an intermediate node, the results of the query need to be materialized into a temporary table. We focus here on two constraints that might be interesting to the users: minimizing the storage consumed, at any point during execution, by the intermediate nodes when executing *a given logical plan*, and constraining plan space with user-specified maximum storage consumed by intermediate temporary tables. Such constraints can be easily incorporated into today's commercial optimizer's query hints framework.

4.4.1 Minimizing Intermediate Storage

In general, the SQL statements corresponding to a given execution plan tree **P** can be generated using either a breadth first or depth first traversal of the tree. When all children of a node *u* have been computed from it, then the intermediate table corresponding to *u* can be dropped, thereby reducing the required storage. However, the manner in which the execution plan tree is traversed for generating the SQL can affect the required storage for intermediate nodes.

The example in Figure 6 illustrates this issue. Consider node (ABCD). If we use a depth-first traversal strategy, we need to execute the entire sub-tree rooted at (ABC) followed by the entire sub-tree rooted at (BCD) before we can delete the temporary table (ABCD). Thus the maximum storage consumed using this strategy is 20 (10+6+4), which corresponds to the storage for simultaneously materializing (ABCD), (ABC) and (AB). If a breadth-first strategy is used, the maximum storage is 18 (10+6+2), which corresponds to the storage for (ABCD), (ABC) and (BCD). Thus, in this example, a breadth-first traversal results in lower maximum required storage. It is easy to see that in other cases, a depth-first traversal may be preferable. Thus, for each node, one of these strategies will be better, depending only on the storage requirements on nodes in the subtree.

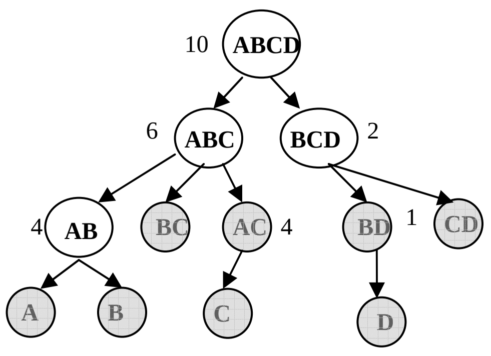


Figure 6. A sub-tree of an execution plan. The numbers are the storage required to materialize a node.

Let *u* be any node, and let d(*u*) denote the storage required for materializing node *u*. Let *Storage*(*u*) denote the minimum storage required for the intermediate nodes (among all possible ways in which the tree can be executed) for the sub-tree rooted at *u*. Let $v_1, \ldots, v_k$ be the children of node *u*. Then the minimum storage for the sub-tree rooted at *u* can be written using the following recursive formula:

$$Storage(u) = \min \begin{Bmatrix} d(u) + \sum_{i=1}^{k} d(v_i) \\ d(u) + \max_{i=1..k} Storage(v_i) \end{Bmatrix}$$

The first term represents the required intermediate storage if a breadth-first traversal is applied at node *u*, whereas the second term represents the required intermediate storage if a depth first traversal is applied at node *u*. Given the above formula, we can compute *Storage* (*u*) for any node in the tree in a bottom-up manner, and thereby determine at *each node*, whether a breadth-first (BF) or depth-first (DF) traversal is preferable. In particular, we mark a node as 'BF' or 'DF' depending on whether the first term or the second term in the formula is smaller in value. Once each node in the tree is marked in this manner, we traverse the tree obeying this marking and compute the nodes in that order. This traversal ensures that the storage consumed during execution of the plan is minimized.

It should be noted that such an optimization may affect the time to return the first result tuple though the time to complete the computation of the whole result set is the same, and therefore it may not be applicable to the case in which the user desires the first tuple is returned as early as possible. It should also be noted that the term d(u) is an estimate from the query optimizer; its accuracy is also subject to the accuracy of optimizer's statistics.

4.4.2 Constraint on Intermediate Storage

In section 4.4.1, we described how to determine the order of execution of queries in a given logical plan so as to minimize the storage consumed by intermediate tables *for that plan*. It is also possible to specify as part of the GB-MQO problem (Section 3.3) a *constraint* on the maximum storage consumed by intermediate tables for a plan. A simple method for solving the constrained problem is to combine the algorithm in Figure 5 with the algorithm to determine the minimum storage in Section 4.4.1: for each plan in MP in Step 4 it also computes its minimum intermediate storage, and only retains the plans that meet the specified constraint.

5. IMPLEMENTATION

In Section 5.1, we outline how the optimization techniques presented in the paper can be integrated with the query optimizer of a DBMS for optimizing a GROUPING SETS query. An alternative is to use these techniques in the client, i.e., in the application itself. This allows today's applications to potentially leverage the benefits of these optimizations on today's existing database systems (Section 5.2).

5.1 Integration with Query Optimizer

Our techniques presented in Section 4.2 can be implemented inside the query optimizer for optimizing a GROUPING SETS query. Today's query optimizers use algebraic transformations to change a logical query tree to an *equivalent* logical query tree [6]. In a Cascades style optimizer [12], these transformations are applied in a cost based manner. The algorithm presented in Section 4.2 can be viewed as a method for obtaining equivalent rewritings of the original GROUPING SETS query. For example, Figure 7 shows two equivalent logical expressions for computing GROUPING SETS ((A), (B)). The sub-tree labeled *Expr* in the figure is the logical expression for the rest of the GROUPING SETS query (e.g., base relation, joins, selections etc.). Each iteration of the GB-MQO algorithm in Figure 5 considers different logically equivalent expressions, each of which is equivalent to the input GROUPING SETS query. Furthermore, these expressions can be compared in a cost based manner. Note that costing of plans can be easily implemented in a query optimizer since these optimizers are already cost based, and we only require the ability to estimate the cardinality and average row size of the result of any Group By query. This capability is already present in today's query optimizers. We note that the capabilities of estimating statistics over query expressions [4] containing Group By queries can also help improve accuracy of cost estimation in this context.

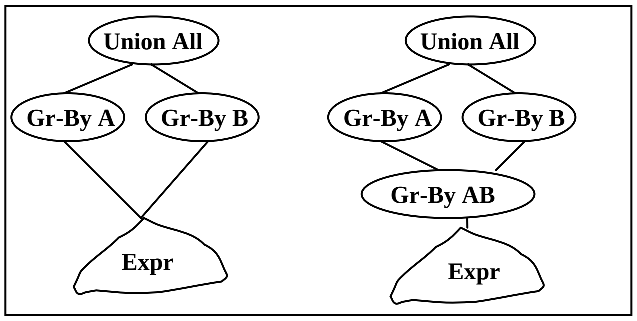


Figure 7. Two logically equivalent expressions of the GROUPING SETS query ((A),(B))

No new physical operators are required by our techniques. In principle, besides the standard Sort and Hash operators, any techniques described for efficient (partial) cube computation (e.g., PipeSort, PipeHash [2,21]) can be used. These operators use the basic ideas of shared scans and sharing sorts [2,8,15,16,21]. The ability to materialize intermediate results is provided by the Spool operator -- also available in today's DBMSs.

5.1.1 GROUPING SETS query with Selections and Joins

In general, a GROUPING SETS query may be defined over an arbitrary SQL expression, rather than a single base relation. Transformations to commute selection and join [7] with Group By can be extended to the GROUPING SETS construct. It is easy to see that selection can be pushed below the GROUPING SETS (e.g. as part of Expr in Figure 7).

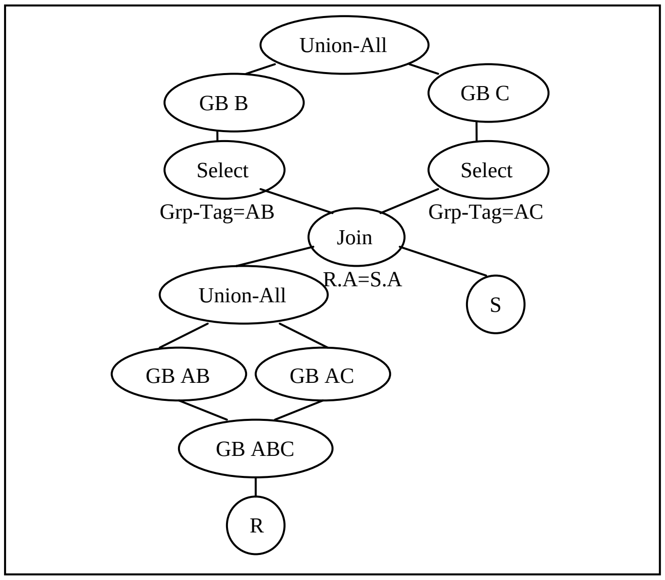


Figure 8. Logical expression for GROUPING SETS query ((B), (C)) over the view Join(R, S).

The transformation to commute join and group by in [7] is also applicable to GROUPING SETS with little change. For example, consider a GROUPING SETS query over the inner join of two relations R and S (joining column is A). Suppose, we are interested in computing the GROUPING SETS ((B), (C)). For simplicity assume both B and C are columns in R. Then the transformation shown in Figure 8 is possible, wherein the Grouping Set computation is "pushed down" below the join of R and S. Note that the pushed down Group By queries over R will need to include the join attribute in the grouping (to allow subsequent joining with S) as the coalescing grouping transformation in [7]. Observe that our optimization techniques can once again be leveraged as shown in the figure by introducing the Group By (A,B,C) on R. An important point to note is that the Union-All below the Join returns a single result set of all Group Bys below it. Thus we need to ensure that the Group Bys *above* the Join get only the respective relevant rows. This can be done by introducing the notion of a Grp-Tag (i.e., a new column) with each tuple that denotes which Group By query it is a result of. This tag can be used to filter out the irrelevant rows. This transformation can be thought of as a generalization of the idea in [7] to multiple Group By queries.

5.2 Client Side Implementation

As mentioned earlier, a client side implementation can be useful until such time that database servers begin to provide efficient GROUPING SETS implementation for certain kinds of data analysis scenarios. In this context, our algorithm in Figure 5 can be viewed as a method for determining *which* additional Group By queries need to be executed (and materialized) to efficiently execute the *input* set of Group By queries.

Given a logical plan (i.e., the output of our algorithm), the application can execute the plan as follows. Consider any edge u->v in the logical plan. Assume that the name of the table corresponding to a node x is T_x (if x is the root of the logical plan, then the table is R). If the node v is an intermediate node (and therefore needs to be materialized), generate a query: SELECT v, COUNT(*) AS cnt INTO T_v FROM T_u GROUP BY *v*. If *v* is a leaf node, then generate the query: SELECT v, COUNT(*) AS cnt FROM T_u GROUP BY *v*. Note that if T_u is an intermediate node (and not R), then we need to replace COUNT(*) with SUM(cnt).

6. EXPERIMENTS

As mentioned earlier, we have done a client side implementation of the algorithm presented in Section 4.2 on top of a commercial DBMS, using the query optimizer cost model (Section 3.2.2). In this section we present the results of experiments that show: (1) that our algorithm performs significantly better than GROUPING SETS for certain data analysis scenarios over real and synthetic data sets (2) the quality of GB-MQO plan compared to the optimal plan (3) that our algorithm scales well with the number of columns (and hence number of Group By queries) in the table (4) the impact of restricting the plan space to binary trees (Section 4.2) (5) the impact of the pruning techniques presented in Section 4.3 (6) the overhead of creating statistics (Section 3.2.2) (7) the impact of skew in the data (8) the impact of physical design of the database.

Table 1. Datasets used in experiments

Dataset	#rows	size	# columns used
1g TPC-H (*lineitem*)	6M	1G	12
10g TPC-H (*lineitem*)	60M	10G	12
SALES	24M	2.5G	15
NREF (*neighboring_seq*)	78M	5G	10

The experiments are run on a Windows XP professional box with a 3GHz Pentium4 CPU and 1 GB RAM. All experiments are run on top of Microsoft SQL Server except those depicted in section 6.1. The synthetic datasets used are the *lineitem* relation in TPC-H 10GB and 1GB datasets[24]. The real world datasets is a proprietary sales data warehouse dataset and the PIR-NREF [20] dataset in which we use the biggest relation neighboring_seq. Table 1 summarizes the data sets used and their characteristics. For the TPC-H datasets in Section 6.1, 6.2, 6.3, 6.5, 6.6 and 6.7 we use a standard physical design in the benchmark test: a clustered index on l_shipdate, a non-clustered index on l_orderkey, and a non-clustered index on the combination of l_suppkey and l_partkey.

6.1 Comparison with GROUPING SETS

We test the performance of our algorithm relative to GROUPING SETS supported by the commercial database system on the TPC-H database. We test GROUPING SETS for two kinds of inputs. In the first scenario we need to compute many single column Group By queries (we call this SC). Thus, in this case, there are no overlapping columns in the inputs. The second input is one in which there are many containment relationships among queries in the input, which is the kind of scenario for which GROUPING SETS is designed (we call this CONT for short). For ease of implementation, the GB-MQO plan (which is a series of Group By queries) reported in Table 2 is obtained by running our algorithm on top of Microsoft SQL Server against the same dataset with the same physical design. We note that ideally the reported GB-MQO plans should be obtained using the what-if API native to the DBMS which executes the plan. However, we expect that the reported plans should be less favorable in performance than the ones obtained via "ideal" methodology because the native what-if API is in tune with the optimizer and the query execution engine, which should result in better GB-MQO plans.

In the SC case, the input was all single column Group By queries except on the floating point columns (l_extendedprice, l_tax, l_discount). Thus the input was 12 single column Group By queries. From Table 2 it can be seen that for SC our approach significantly outperformed GROUPING SETS. By looking at the plan produced for the GROUPING SETS query for SC, we see that the plan cannot arrange any sharing in the processing of the 12 Group Bys except for grouping *all* the 12 columns and computing all 12 Group Bys from this result. Because the intermediate result of grouping 12 columns is almost as big as the base table, such an optimization is inadequate. On the other hand, our algorithm *introduced* 2 subsuming Group By queries and performs 7 out of 12 Group Bys on these two much smaller intermediate result sets. As a result, our algorithm was 4.5 times faster than using GROUPING SETS.

For CONT, the input was {(l_shipdate), (l_commitdate), (l_receiptdate), (l_shipdate, l_commitdate), (l_shipdate, l_receiptdate), (l_commitdate, l_receiptdate)}. In this case GROUPING SETS and our algorithm result in comparable performance. The GROUPING SETS plan takes advantage of the idea of shared sorts to speed up processing relative to the naïve approach of using one Group By per input query, since many containment relationships hold in the inputs; i.e., it arranges the sorting order so that if a grouping set subsumes another, the subsumed grouping is almost free of cost. Our algorithm did not introduce any new Group By, but arranged the singleton grouping sets to use index or the smallest result set of the two-column grouping-sets. We note that our implementation is on client side; so we are limited to issuing SQL queries. When implemented inside the server our approach can also potentially benefit from shared sorts as above. Thus, in a server side implementation, we would expect even greater speedup over GROUPING SETS.

Table 2. Speedup over GROUPING SETS

Query	GrpSet Time (s)	GB-MQO Time (s)	Speedup
CONT	213	167	1.2
SC	1230	270	4.5

6.2 Evaluation on different datasets

We studied the performance of our algorithm compared to the naïve approach on synthetic and real world datasets. We computed single column (denoted SC) as well as two-column Group By queries (denoted TC) on all columns in each data set. Table 3 summarizes the running times. We see that our approach consistently outperforms the naïve approach (where each input query is run against the original table) with a speedup factor from 1.9 to 4.5. Although we omit the numbers, we observed that once again GROUPING SETS performance was almost the same as the naïve approach in most of these cases.

Table 3. Speedup over naïve plan on different datasets

Datasets	Sales (SC)	NREF (SC)	10g (SC)	1g (SC)	Sales (TC)	NREF (TC)	10g (TC)	1g (TC)
#GrBys	15	11	12	12	105	44	66	66
Speedup	1.9	1.4	2.3	2.3	4.5	1.7	2.3	2.0

6.3 Comparison with Optimal Plan

To empirically study how the quality of the GB-MQO plans differs from that of the optimal plans, we implemented an exhaustive search algorithm to find the optimal plan under the Microsoft SQL Server optimizer's cost model. Due to the exponential nature of the exhaustive search, we restricted the number of columns to 7. We randomly generated 10 queries, labeled from Q0 to Q9, by randomly choosing 7 columns out of the 12 non-floating-point columns of the TPC-H 1G data set and then computing single column Group Bys on these 7 columns. Figure 9 summarized the running time reduction ratio of the GB-MQO plans and of the optimal plans against the naïve plans. From the figure we can see most of the times the quality of the GB-MQO plans are close to that of the optimal one.

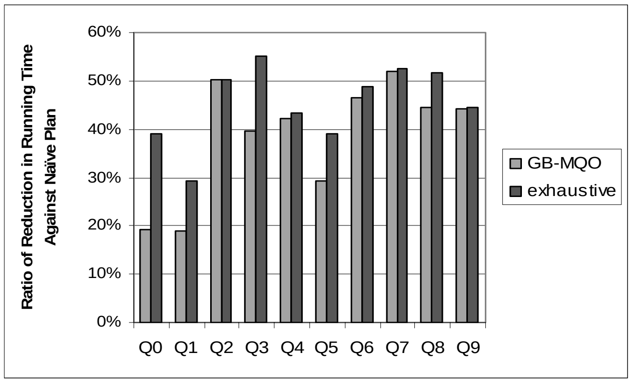
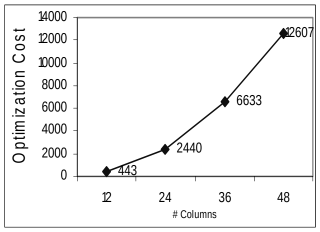


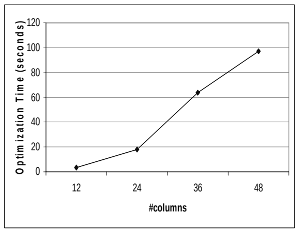
Figure 9. Ratio of run time reduction of GB-MQO plans and optimal plans.

6.4 Scaling With Number of Columns

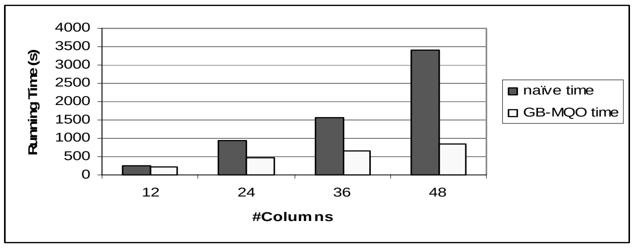
In this experiment, we investigate the scalability of our algorithm for the case where we want all single column Group By queries, and the number of columns in the table is increased. We start with the projection of the 1GB TPC-H *lineitem* relation on its 12 non-floating-point columns, and widen it by repeating all 12 columns. We put aside the time of creating statistics because the cost can be amortized by using them to optimize many queries other than this grouping-sets query. Since now the optimization overhead is dominated by calling the query optimizer to cost the queries, we use the number of optimizer calls as the metrics for the cost of optimization. We also report the running time of our algorithm compared to the naïve plan. Because of its quadratic nature, our approach scales reasonably well on wide table as seen in Figure 10. Thus optimizing 48 single-column Group By queries can be accomplished within 100 seconds. Once again, we note that if implemented inside the server, this optimization can be done even faster since we do not incur overhead of creating dummy intermediate tables as well as invoking the query optimizer repeatedly during optimization.



(a)



(b)



(c)

Figure 10. (a) Number of calls to the query optimizer (b) Optimization time (c) Running time relative to naïve plan.

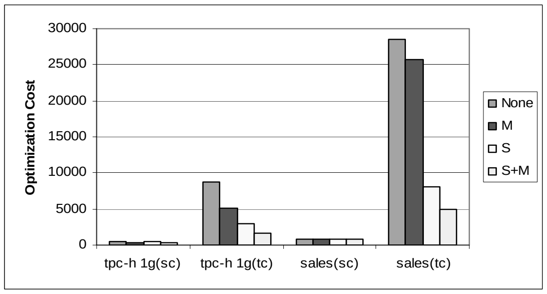
6.5 Impact of Restricting To Binary Tree Plan

We studied the impact of restricting the plan space searched by our algorithm to binary trees when computing all single column Group By queries over TPC-H and Sales databases. Specifically, we compared the optimization cost and the running time of the plan found by applying only *SubPlanMerge* (b) in Figure 4 vs. that found with all four ways of merging. As in Section 6.4, we use the number of queries sent to optimizer to cost as the optimization cost metrics. We found that for both these datasets the number of optimizer calls reduced by 30%, while the difference in the execution times was less than 10%. The reason is that our algorithm obtains much of its benefit by merging sub-plans with small intermediate results. Thus even though the binary tree introduces additional queries to be materialized (compared to a *k*-way (*k*>2) tree), the costs of materialization are relatively small.

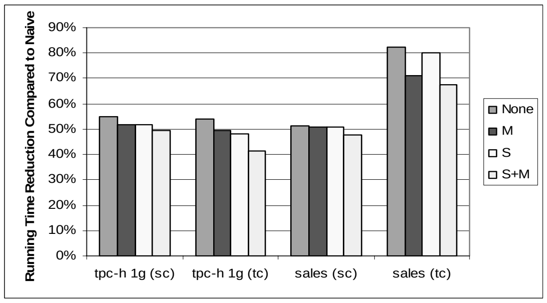
6.6 Impact of Pruning Techniques

In this experiment we used the TPC-H 1G dataset and the Sales dataset to evaluate the effect of the pruning heuristics in Section 4.3 on the quality of plan and the efficiency of the algorithm. We computed all the single-column and two-column Group By queries and compared the cost of optimization (measured by number of calls to the query optimizer) and the running time of the plan with one or both pruning techniques enabled.

From Figure 11 we can see that both the Subsumption based pruning technique (denoted by S in the figure) as well as the Monotonicity Based pruning technique (M) can dramatically reduce the search space in the two-column cases, and combined together, can cut the number of calls to the optimizer by as much as 80%, while the optimized plan can still reduce the running time of naïve approach by more than 65%.



(a)



(b)

Figure 11. (a) Number of calls to the query optimizer (b) Execution Time

6.7 Overhead of Statistics Creation

In this experiment we studied the overhead of statistics creation, which is defined as the time to create statistics as a percentage of the savings in running time. We computed all single column and two-column Group By queries over TPC-H 1G and 10G datasets. We assumed there was no existing statistics on the base table at the very beginning and the algorithm in Figure 5 (with subsumption based pruning enabled) created a statistics on the grouping columns of a Group By query if it encountered that Group By for the first time. From Figure 12 we can see that creating statistics is just a small fraction compared to the running time savings of the GB-MQO plans over the naïve plans. In general, the statistics creation overhead appears to become smaller as the dataset becomes larger.

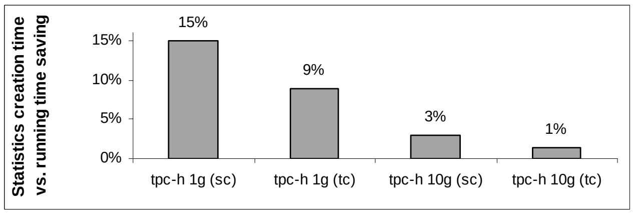


Figure 12. Statistics creation time versus savings in running time.

6.8 Varying Data Skew

In this experiment we examine how the effect of data skew affects our algorithm. We generated TPC-H 1GB datasets with varying Zipfian distributions of skew factor 0, 0.5, 1, 1.5, 2, 2.5, 3, and compared the execution time of our algorithm with the naïve plan.

Figure 13 is the plot of the speedup factor vs. data skew. The increase in speedup comes from the fact that as a column become more skewed, it becomes more sparse (fewer number of distinct values). This typically tends to make merging of sub-plans more attractive.

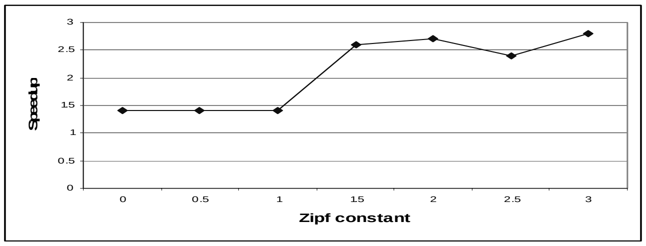


Figure 13. Speedup vs. Varying Data skew (Zipfian)

6.9 Impact of Physical Database Design

One attractive property of our algorithm (Section 4.2) is that it assumes no knowledge of the physical design. In this experiment we examine how it performs on different physical designs (on TPC-H 1GB database). Starting with a clustered index on the combined primary key l_orderkey and l_linenumber, we built non-clustered indices on l_receiptdate, l_shipdate, l_commitdate, l_partkey, l_suppkey, l_returnflag, l_linestatus, l_shipinstruct, l_shipmode, l_comment, one per step in the above order, and compare the running time after each step.

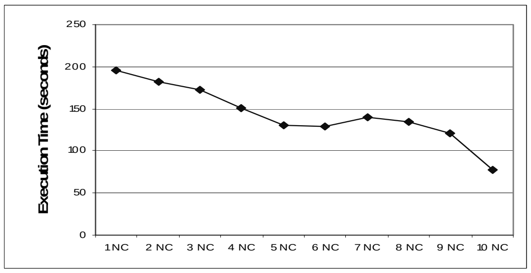


Figure 14. TPC-H 1GB Variation with physical design.

We can see from Figure 14 that our approach automatically benefits from the addition of indices, especially for the highly dense column which cannot be merged with other Group Bys (e.g., l_comment). Further examining the plans we see that the plans generated can adapt to the physical design. For instance, before an index on l_receiptdate is built l_shipdate and l_receiptdate are combined together, while after the index is built, l_receiptdate remains a singleton and l_shipdate gets merged with other columns. Such ability to adapt to physical design is not surprising since the optimizer cost model automatically captures the influence of different access plans.

7. EXTENSIONS

7.1 Using CUBE and ROLLUP

The definition of a node in a logical plan is a Group By query. However, in some cases it may be beneficial to also consider CUBE and ROLLUP queries in the plan. For example, suppose the nodes (AB), (A) and (B) are required nodes. Then instead of having a sub-tree of Group By queries, it may be less expensive to have a plan consisting of a single CUBE query on (AB), which would also give the same results. However, consider a case where only (A) and (B) are required nodes. In this case, the naïve plan of directly computing (A) and (B) from the relation R using Group By queries may be more efficient than using the CUBE operator (since the work done for computing (A,B) may be more than any benefit it provides). A similar argument applies for the ROLLUP query. For example, a ROLLUP A, B query will compute (A,B) as well as (A), but not (B). Thus, it can be more beneficial than using CUBE or simply Group By queries for certain cases. Incorporating the use of CUBE and ROLLUP queries into the algorithm (Figure 5) can be done as follows. When merging two sub-plans, in addition to the alternatives considered in Figure 4, we also consider replacing ($v_1 \cup v_2$) with CUBE ($v_1 \cup v_2$) or ROLLUP ($v_1 \cup v_2$). Once again, these alternatives plans can be considered in a cost based manner since the query optimizer is capable of costing CUBE and ROLLUP queries similar to regular Group By queries.

7.2 Handling Different Aggregates

Thus far we have assumed that all queries contain the same aggregate COUNT(*). Our solution can be extended to other aggregates such as MIN(X), MAX(X) and SUM(X). One way to handle multiple aggregates in the *SubPlanMerge* module (see Section 4.1) is by taking the union of all aggregates of nodes v_1 and v_2. The downside of such an approach is that it can potentially lead to a blowup in the size (specifically number of columns) of the node ($v_1 \cup v_2$), thereby making it less attractive to materialize. Thus, in principle, it may be beneficial to consider materializing multiple copies of ($v_1 \cup v_2$) each with only a subset of the aggregates required. The downside of the latter approach is that the cost of computing these copies and materializing them can be high. Once again, the decision of which method to use can be done in a cost based manner. We omit details due to lack of space.

8. CONCLUSION

We present an optimization technique for GROUPING SETS queries for common data analysis scenarios such as computing all single column Group By queries of a relation. Unlike previous approaches, we use a bottom up approach that does not require the entire search DAG as input. Our cost based approach is important since it enables easier integration with today's query optimizers as well as efficient implementation from a client application. We see significant performance improvements compared to the naïve approach or even using available GROUPING SETS functionality in today's commercial DBMSs. Developing transformations for optimizing a GROUPING SETS query with other relational operators is an interesting area of future work.

9. REFERENCES

[1] Agrawal, R., Ramakrishnan, S. Fast Algorithms for Mining Association Rules in Large Databases. In *Proc. of VLDB* 1994, 487-499.

[2] Agrawal et al. On the Computation of Multidimensional Aggregates. In *Proc. of VLDB* 1996, 506-521.

[3] Agrawal, S., Chaudhuri, S., and Narasayya, V. Automated Selection of Materialized Views and Indexes for SQL Databases. In *Proc. of VLDB* 2000, 496-505.

[4] Bruno, N. and Chaudhuri, S. Exploiting Statistics on Query Expressions for Query Optimization. Proc. of SIGMOD 2002, 263-274.

[5] Chaudhuri, S., and Narasayya, V. AutoAdmin 'What-If' Index Analysis Utility. In *Proc. of SIGMOD* 1998, 367-378.

[6] Chaudhuri, S. An Overview of Query Optimization in Relational Systems. In *Proc. of PODS* 1998, 34-43.

[7] Chaudhuri S., and Shim K. Including Group-By in Query Optimization. In *Proc. of VLDB* 1994, 354-366.

[8] Dalvi N., Sanghai S, Roy P., and Sudarshan S. Pipelining in Multi-Query Optimization. In *Proc. of PODS* 2001.

[9] Wolfgang Lehner, et al. fAST Refresh Using Mass Query Optimization. In *Proc. of ICDE* 2001, 391-398.

[10] Gupta, H. Selection of views to materialize in a data warehouse. In *Proc. of ICDT* 1997, 98-112.

[11] Gupta H., and Mumick, I.S. Selection of Views to Materialize Under a Maintenance-Time Constraint. In *Proc. of ICDT* 1999, 453-470.

[12] Graefe G. The Cascades Framework for Query Optimization. In Data Engineering Bulletin (Sept 1995), 19-29.

[13] Haas P., Naughton J., Seshadri S., and Stokes L Sampling-based estimation of the number of distinct values of an attribute. In *Proc. of VLDB* 1995, 311-322.

[14] Harinarayan V., Rajarama A., and Ullman J. Implementing Data Cubes Efficiently. In *Proc. of SIGMOD* 1996, 205-216.

[15] Hinneburg A., Habich D., and Lehner W. COMBI-Operator – Database Support for Data Mining Applications. In *Proc. of VLDB* 2003, 429-439.

[16] Ross K., and Srivastava D. Fast Computation of Sparse Datacubes. In *Proc. of VLDB* 1997, 116-125.

[17] Ross K., Srivastava D, and Sudarshan S. Materialized View Maintenance and Integrity Constraint Checking: Trading Space for Time. In *Proc. of SIGMOD* 1996, 447-458.

[18] Jack Olsen. Data Quality: The Accuracy Dimension. Morgan Kaufmann Publishers, 2002.

[19] Roy P., et al. Efficient and extensible algorithms for multi query optimization. In *Proc. of SIGMOD* 2000 249-260

[20] Protein Information Resource (PIR) web site. http://pir.georgetown.edu/

[21] Sarawagi S., Agrawal R., and Gupta A. On Compressing the Data Cube. IBM Technical Report.

[22] Scheufele, W., and Moerkotte, G.: On the Complexity of Generating Optimal Plans with Cross Products. In *Proc. of ACM PODS* 1997, 238-248.

[23] Sellis T., "Multiple Query Optimization", ACM TODS, 13(1) (March 1988), 23-52.

[24] TPC Benchmark H. Decision Support. http://www.tpc.org

[25] Zilio D. et al. Recommending Materialized Views and Indexes with IBM's DB2 Design Advisor. In *Proc. of ICAC* 2004, 180-188.

APPENDIX A: Hardness Result

Claim: The GB-MQO problem is NP-Complete even if we restrict the input set S to include only single column Group By queries and use the Cardinality cost model (Section 3.2.1).

Proof: We show a reduction from the problem of determining the optimal bushy plan for the cross product query of N relations (referred to as the XR problem), which is known to be NP-Complete[22] to the GB-MQO problem.

In XR, the cost of a cross product of two relations R_i and R_j is assumed to be $|R_i|\cdot|R_j|$, called the *table cardinality cost model*. Given N relations $R_1, ..., R_N$, without loss of generality, we can assume that all of them have just one column and all tuples are distinct. (We can make this true by constructing a new R_i' from R_i by concatenating a unique row id with all the columns belonging to the same tuple in R_i). Without loss of generality we can also assume that $|R_i|>1$ because under the table cardinality cost model, such single-tuple relations do not change the cost of the cross product.

Let $R = R_1\times...\times R_N$. Observe that Q_i = select c_i from R Group By c_i is indeed R_i and Q = select $c_1, \ldots, c_N$ from R Group By $c_1, \ldots, c_N$ is R. Let *f* be a mapping from a bushy cross product plan to a logical plan to compute all single column Group By queries from relation R defined as follows.

An internal node *n* in the cross product plan representing $R_{i_1} \times ... \times R_{i_k}$ is mapped to Q_n = select $c^{(1)}, ..., c^{(k)}$ from Q_P Group By $c^{(1)}, ..., c^{(k)}$, where Q_P is R if *n* is the root , or the mapping of *n*'s parent P otherwise, and a leaf node representing R_j is mapped to Q = select c_j from Q_P Group By c_j. It is easy to see that *f* is reversible.

Let T be the space of all cross product plans in XR. We need to show that (1) the optimal logical plan P_{opt} for GB-MQO is in the space of *f*(T) and (2) $f^{-1}(P_{opt})$ is the optimal join plan for XR.

Proof of sub-claim (1) by contradiction: it is equivalent to show that the optimal logical plan for GB-MQO consists of 2 sub-plans and both sub-plans are binary tree plans. First if there is only one sub-plan, the root of the sub-plan must be $\{c_1, \ldots, c_N\}$, which implies P_{opt} is sub-optimal because the edge pointing from the root to R is redundant. If there are more than 2 sub-plans, we apply *SubPlanMerge* (b) to merge the first two sub-plans, and get a new plan P. Let n_1 and n_2 be the root of the first two sub-plans and c_x be a column not contained in either n_1 or n_2. We have

$$\begin{aligned} C(P) - C(P_{opt}) &= |R| + 2|n_1 \cup n_2| + \sum_{e\in T_1} w(e) + \sum_{e\in T_2} w(e) \\ &\quad - |R| - \sum_{e\in T_1} w(e) - |R| - \sum_{e\in T_2} w(e) \\ &= 2|n_1 \cup n_2| - |R| = 2 \prod_{i\in n_1\cup n_2} |R^{(i)}| - |R^{(1)}| \mathrm{L} |R^{(N)}| \\ &\le (\prod_{i\in n_1\cup n_2} |R^{(i)}|)(2 - |R^{(x)}|) \le 0 \end{aligned}$$

where *w(e)* equals to |*u*| for an edge *e* pointing from *u* to *v*. The last inequality comes from the fact that $|R^{(x)}| \ge 2$. This contradicts P_{opt} being the optimal plan. So P_{opt} must have two and only two sub-plans. Likewise, if P_{opt} has a node *n* which has m children where m>2. Let n_1 and n_2 denote the first two children. We can create a new plan P with lower cost by introducing $n_1 \cup n_2$ as a child of *n* and making n_1 and n_2 be the children of $n_1 \cup n_2$.

$$C(P) - C(P_{opt}) = 2|n_1 \cup n_2| - |n| = (\prod_{i\in n_1\cup n_2} |R^{(i)}|)(2 - \prod_{j\in(n-n_1\cup n_2)} |R^{(j)}|) \le 0$$

The last inequality holds because $n - n_1 \cup n_2$ is not empty and any $|R^{(i)}|\ge 2$. So P_{opt} must be a binary tree.

Proof of sub-claim (2): Given a join plan tree T,

$C(T) = \sum_{n\in I} |n| + \sum_{n\in L} |n| = \sum_{n\in I} |n| + |R_1| + \mathrm{L} + |R_N|$ where I and L are the set of internal nodes and leaf nodes respectively. Note that the second term is a constant so to minimize C(T) is to minimize the first term $C'(T) = \sum_{n\in I} |n|$. Let P be *f*(T). Note that $C(P) = \sum_{n\in I} 2|n| = 2\cdot C'(T)$. So if we can find the optimal Group By plan P_{opt} with minimal $C(P_{opt})$, let $T_{opt}=f^{-1}(P_{opt})$, then $C(T_{opt})=0.5\times C(P_{opt})+|R_1|+\ldots+|R_N|$ is minimal. QED.