An Overview of Cost-based Optimization of Queries with Aggregates

Surajit Chaudhuri
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304
chaudhuri@hpl.hp.com

Kyuseok Shim
IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120
kshim@almaden.ibm.com

1 Introduction

The optimization problem of Select-Project-Join queries has been studied extensively. However, a problem that has until recently received relatively less attention is that of optimizing queries with aggregates. For example, for single block SQL, traditional query processing systems directly implement SQL semantics and defer execution of grouping until all joins in the FROM and WHERE clauses have been executed. Furthermore, for queries that reference views with aggregates, traditional optimizers do not consider flattening such views. In this paper, we will show that there is a rich set of execution alternatives that can significantly enhance the quality of the plans produced. We also discuss how one can choose among the alternatives.

First, let us consider single-block SQL queries. For these queries, the reason to consider alternatives where a group-by operation precedes a join is that an early evaluation of the group-by can significantly reduce the size of the input to the join. Such a transformation may also result in the grouping operation to be pushed down to a base table. In such an event, one can use indexes on the base tables to combine the operation of the join and the group-by.

Next, in case of queries containing views with aggregates, the presence of Group By hinders the ability to reorder relations within the view with relations outside. Yet, reordering joins across view boundaries has the potential of generating cheaper execution alternatives.

In order to be able to optimize queries with aggregates, we need:

- Transformations that enable us to optimize queries containing Group By and join.
- Optimization algorithms that can incorporate the above transformations without increasing the search space dramatically.

We begin by considering optimization of single-block SQL queries. We refer the readers to [CS94, CS95, CS96] for the technical details.

2 Transformations for Single Block SQL

There are several transformations that make it possible for a group-by to precede a join. The conditions that make such transformations possible fall in the following broad categories:

(1) Conditions that depend solely on the schema and the nature of the join predicates and independent of the nature of the aggregating functions.

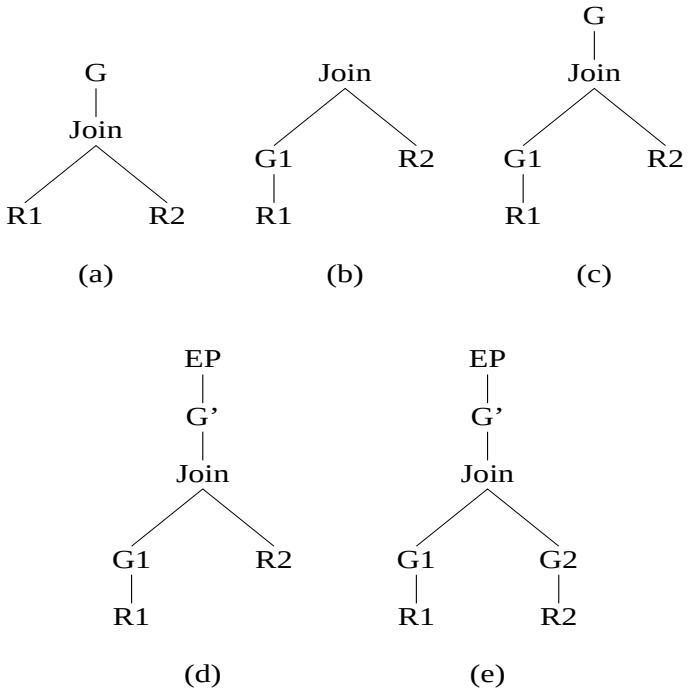

Figure 1: Transformations

(2) Conditions that depend on specific properties of aggregating functions. This can lead to *introduction* of additional group-by nodes.

(a) Conditions that do *not* require introducing derived columns.

(b) Conditions that require use of *derived columns*

Transformations that belong to group (1) are applicable for any aggregating function, including any side-effect free *user-defined function*. An application of a transformation to the operator tree in Figure 1(a) leads to the transformed operator tree illustrated by Figure 1(b). Observe that the transformation in the figure does *not* increase the number of operators. For example, consider the case with two tables `Sales(saleid, amount, prodid)`,`Product(prodid, division, state)` and the query is to calculate total sales for each product developed in Western states. In such a case, the traditional execution (Figure 1(a)) takes a join between the two tables and then computes a `Group By` on `prodid`. In the alternative scheme of Figure 1(b), we first group by `Sales` table and compute the total sales for each product. Subsequently, we eliminate results that do not correspond to products developed in the western states. If the selectivity of the join is large, then the execution of Figure (b) may be more efficient.

By exploiting the properties of the aggregating functions make it possible to do "stagewise" group-by leading to *introduction* of the group-by nodes (e.g., in Figure 1(c)). These aggregation functions follow the property:

$$Agg(S \cup S') \quad = \quad Agg(Agg(S) \cup Agg(S')) \tag{1}$$

where S and S' are bags of values and $\cup$ denotes SQL's union-all operation on the bags. For example, the aggregate function SUM follows the above property since $SUM(S \cup S') = SUM(SUM(S) \cup SUM(S'))$. Let us consider a simple modification of the query given earlier. Assume that we want to compute the total sales for each division where a division may have many products. By applying the transformation in Figure 1(c), we can compute the total sales for each product, then join with `Product` table, and finally add up the sum of sales for products that belong to the same division.

Finally, there are cases where keeping track of the *count* of the coalesced groups makes it possible to introduce group-by or to do an early group-by. In such cases, the properties of the aggregating functions that may be admitted can be more general than Equation 1:

$$Agg(S \cup S') = f(Agg_1(S), Agg_1(S'), Count(S), Count(S')) \quad (2)$$

where S and S' are bags of values and $\cup$ denotes SQL's union-all operation. However, computing stagewise grouping now requires use of derived column values as well. For example, given the count and the average for two groups, the average of the coalesced group can be computed. This is illustrated by Figure 1(d) where the node EP (*extended projection*) represents the computation of the aggregated value using the relationship in Equation 2. Observe that the execution of EP and the join node can be combined in one single join implementation.

Transformations (b)-(d) represent early evaluation of grouping on only one operand of the join node. They are generalized in Figure (e) to both sides of the operands and represent aggressive use of early grouping.

3 Optimization Algorithm for Single Block Query

The decision whether or not to apply a transformation needs to be made by the optimizer based on anticipated costs. However, such choice also *interacts with join ordering* since group-by has the effect of reducing the number of tuples at the cost of requiring sorting or hashing (unless the data stream is pre-sorted). Thus, the problem of optimization in the presence of group-by and aggregates has the following properties:

1. *Increase in the number of operators:* A group-by operator may be added, i.e., the number of operators may increase, e.g., transformation in Figure 1(c),(d)&(e).

2. *Effect on the Physical Properties:* Application of a grouping operation G on a relation R results in a relation not just with fewer tuples than in R, but also with differences in the physical properties:

   (a) Application of a group-by operator may *change* the width of the intermediate relation.

   (b) Application of a group-by operator interacts with *interesting orders*[SAC$^+$79].

The most difficult problem in handling the optimization problem in the presence of group-by arises from the fact that the transformations can introduce *additional* group-by operators. Thus, the potential execution space is dramatically increased since there may be an opportunity to place a group-by preceding every join!

Another challenge in optimizing queries is that it is not possible to locally compare segments of plans such as in Figure 1(a) and Figure 1(b) and to choose the winner, for two reasons. First, in grouping early, a new interesting order may be introduced if the grouping is implemented by sorting. Thus, although Figure 1(b) may not win locally, its order generated by sorting may reduce the cost of future join operations. This effect is similar to that of considering a sub-optimal sort-merge join in traditional Select-Project-Join queries. Moreover, one needs to address the problem of determining the major to minor order for sorting (if the grouping is implemented by sorting) and its relationship to the join predicates. Similar considerations arise if grouping is implemented by hashing as well. Next, observe that if there are more than one aggregation functions on the same column, then the width of the relation after application of the grouping will increase. The effect of increased width leads to an effect converse to that of interesting order. Thus, even if Figure 1(b) was locally better than Figure 1(a), it may perform suboptimally because increased width may lead to higher cost globally.

3.1 Greedy Conservative Heuristic

We proposed *greedy conservative* heuristic as a technique to optimize single block SQL with group-by [CS94] by exploiting the transformations in Figure 1. The design principles of greedy conservative heuristic were to ensure

that: (1) the quality of the plan produced is no worse than the execution plan produced by a traditional optimizer and (2) there is no significant overhead in optimization.

In order to ensure that the cost of increase in optimization is modest, we decided that the choice between applying an early grouping and late grouping must be made *locally*. The greedy conservative heuristic places a group-by preceding a join *if and only if* the following conditions hold:

(1) It is semantically correct.

(2) The record width in the output of the group-by operator is no larger than the one in its input.

(3) It results in a cheaper plan for *that* join.

Thus, while considering the join between two relations R and S, if the above conditions (1) and (2) hold, the cost of the optimal plan for $R \bowtie S$ is compared against the cost for $G(R) \bowtie S$, where $G(R)$ represents application of a group-by. Note that either R or S (or, both) can be intermediate relations. Furthermore, the greedy conservative heuristic considers only a *single* ordering of the group-by columns for each join method considered for implementing $R \bowtie S$. Thus, when sort-merge is used for the join method, we choose a major to minor ordering that is the same as that for the join node. Thus, by virtue of the above local decisions, the greedy conservative heuristic adds little overhead in optimization while considering a significantly extended execution space.

As pointed out in the previous section, a local decision such as those made by greedy conservative can be suboptimal. However, the greedy conservative heuristic ensures that the local decision does not result in a plan that is worse than the traditional execution. To enforce such a property, a plan such as $G(R) \bowtie S$ is considered only when the width of the intermediate obtained by $G(R)$ does not increase (i.e. Condition (2)). Our experiments with the implementation (an extension of [SAC+79]) indicate a significant increase (sometimes an order of magnitude) in the quality of the plans for a very modest increase in the optimization overhead [CS94]. Our results are particularly encouraging in view of the fact that we took a conservative cost model that discourages early evaluation of group-by operators. Although our implementation was with a System-R style optimizer, the greedy conservative heuristic can be incorporated in other optimizer architectures as well.

4 Beyond Single Block Queries

In general, a query that uses a view definition with aggregate, cannot be turned into a single-block Select-Project-Join (SPJ) query. In this section, we will discuss how group-by operators can be moved in the traditional query graph across query blocks. Relative placement of group-by across query blocks has significant impact on the cost of the plan of a query (with views containing aggregates) due to its effect on data reduction and reordering of relations across query blocks. In this section, we discuss a new approach to optimizing such queries.

Pull-up Transformation

We can use a transformation to pull up the group-by operator above join. Such a transformation is interesting since by pulling up the group by, it enables alternative join orders. Indeed, a sequence of pull-up transformations may result in collapsing a multi-block query into a single-block query. In this section, we explore how *given an aggregate view*, we can obtain a query with group-by on the top.

Let us consider the execution plans `P1` and `P2` in Figure 2. In the figure, `R1` and `R2`, `J1` and `J2`, and `G1` and `G2` are base tables (or query expressions), join nodes and group by operators, respectively. In this plan, we defer the group-by to obtain an equivalent plan `P2` in the following manner (See Figure 2): The grouping columns of `G2` will be the union of the grouping columns of `G1` and the key (columns) of `R2`. The intuition is that by "tagging" each tuple of `R1` with the key of `R2`, we can recover the original groups by including the key attribute of `R2`

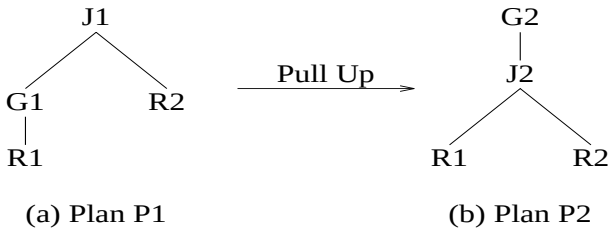

Figure 2: Pulling Up a Group-By operator

among the grouping columns of `G2`. Subsequent to deferred grouping (`G2`), we can throw away the key column of (`R2`) unless it is part of the projection columns. Note that since in `P2`, `G2` has key column(s) of `R2` among its grouping columns, the projection column of the join node `J2` includes these columns as well. Furthermore, the projection list of the join node `J2` includes aggregating columns. The transformation is applicable only if the join predicate in plan `P1` do not involve any aggregation column of `G1`. The formal statement of the transformation appears in [CS96].

The paper by Dayal [Day87] briefly touches on the idea of using keys for pull-up. Ganski-Wong [GW87] implicitly used a restricted form of pull-up for the specific class of Join-Aggregate queries that they optimized. These are special instances of the pull-up transformations defined above.

Optimization Algorithm

The class of queries $\mathcal{Q}$ for which we consider the optimization algorithm has the canonical form shown in Figure 3 (See [CS96] for details):

- Every aggregate view V_i consists of a select-project-join expression with a Group By clause and possibly a `Having` clause. Thus V_i contains a join and a group-by operator G_i.
- The query is a single block query, consisting of a join among aggregate views and a set of base tables i.e., join among $V_1, ..V_m, B_1, ..B_n$ where V_i are aggregate views and B_i are base relations. The query may contain a Group By and a `Having` clause. The set of relations $\{B_1, B_2, ..\}$ and $\{V_1, V_2, ..\}$ are denoted by $\mathcal{B}$ and $\mathcal{V}$ respectively.

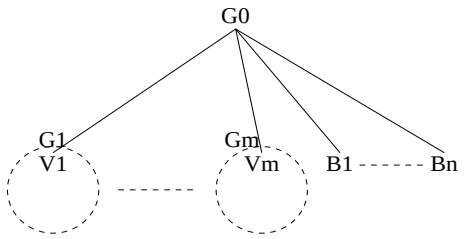

Figure 3: Canonical Form of Query

Our approach to extending the traditional optimization algorithm for multi-block queries by using pull-up as well as push-down transformations is guided by the following two criteria First, our approach should be adaptable to existing optimizers and should produce a plan that never does worse than a traditional optimizer. Next, we want to be able to limit the extent of search in a systematic fashion. Thus, our key idea is to apply pull-up transformation between a view V_i and a relation in $\mathcal{B}$, but not among relations in V_i and V_j where $i \neq j$. Additional reordering among relations in V_i and V_j is enabled by a preprocessing step. The details of such an algorithm is in [CS96].

Relationship to Correlated Nested Queries

We note that the *Join-Aggregate* class of nested queries is closely related to that of optimizing queries containing views with aggregates. This follows from the past work in *flattening* nested queries, pioneered by Kim [Kim82]. For example, the result of Kim's transformation is a query that is a join [1] of base tables and one or more aggregate views, i.e., views containing aggregates. Therefore, the result of optimizing queries containing views with aggregates can be used for optimizing queries with correlated nested subqueries. We can also show that alternative flattening schemes for nested subqueries such as that due to Ganski [GW87] are special cases of as execution alternatives due to optimization of aggregate views. The differences between the transformations due to Kim and Ganski are highlighted by Figure 4 and can be seen as arising due to two extreme placements of the group-by operators. Thus, the problem of optimizing queries containing aggregate views is not only a key problem in its own right, it also directly bears upon the problem of optimizing queries with nested subqueries.

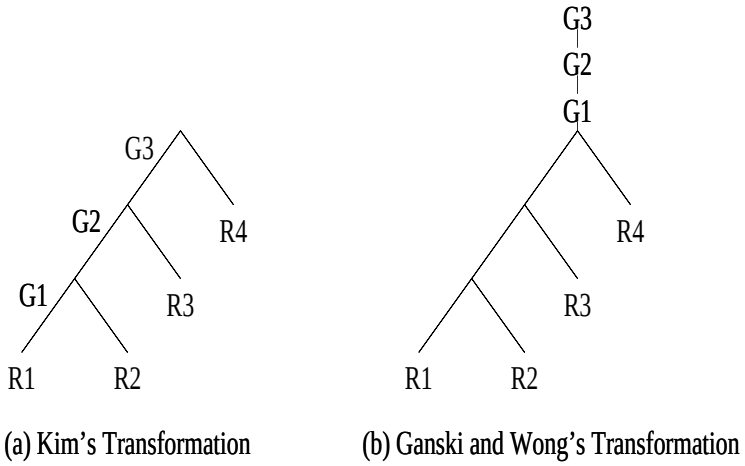

Figure 4: Unnesting Transformations

5 Conclusion

With the increasing emphasis on decision-support systems, the problem of optimizing queries with aggregates have assumed increasing importance. This article has pointed to some of the recent developments in this area. However, the evolving querying models for decision-support systems (e.g., OLAP interfaces) will spur further development in this area.

To Probe Further

Invariant grouping for a single block query was studied independently by us in [CS94] and in [YL94]. Our paper also introduced simple coalescing and generalized coalescing. Later, additional transformations (including pull-up) on group-by were reported in [YL95, GHQ95, CS96]. The problem of optimization using the transformations was addressed by us in [CS94] for single block SQL queries and in [CS96] for multi block SQL queries. A comprehensive treatment of the transformations as well as the optimization algorithms discussed in this article is forthcoming in [CS95].

[1] Note that for correctness, such transformations require outerjoin in the general case [GW87, Mur92].

References

[CS94] S. Chaudhuri and K. Shim. Including group-by in query optimization. In *Proceedings of the 20th International VLDB Conference*, Santiago, Chile, Sept 1994.

[CS95] S. Chaudhuri and K. Shim. Complex queries: A unified approach. Technical report, Hewlett-Packard Laboratories, Palo Alto, In preparation to submit for publication 1995.

[CS96] S. Chaudhuri and K. Shim. Query optimization with aggregate views. In *Proceedings of the 5th International Conference on Extending Database Technology*, Avignon, France, March 1996.

[Day87] U. Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *Proceedings of the 13th International VLDB Conference*, Brighton, August 1987.

[GHQ95] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. In *Proceedings of the 21st International VLDB Conference*, Zurich, Sept 1995.

[GW87] Richard A. Ganski and Harry K. T. Wong. Optimization of nested SQL queries revisited. In *Proc. of the 1987 ACM-SIGMOD Conference on the Management of Data*, San Francisco, CA, May 1987.

[Kim82] W. Kim. On optimizing an SQL-Like nested query. *ACM TODS*, Sept 1982.

[Mur92] M. Muralikrishna. Improved unnesting algorithms for join aggregate SQL queries. In *Proceedings of the 18th International VLDB Conference*, Vancouver, Canada, August 1992.

[SAC+79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD International Symposium on Management of Data*, Boston, MA, June 1979.

[YL94] W. P. Yan and P. A. Larson. Performing group-by before join. In *Proceedings of International Conference on Data Engineering*, Houston, Feb 1994.

[YL95] W. P. Yan and P. A. Larson. Eager aggregation and lazy aggregation. In *Proceedings of the 21st International VLDB Conference*, Zurich, Sept 1995.