Aggregate-Query Processing in Data Warehousing Environments*

Ashish Gupta **Venky Harinarayan** **Dallan Quass**

IBM Almaden Research Center Stanford University

Abstract

In this paper we introduce *generalized projections* (*GPs*), an extension of duplicate-eliminating projections, that capture aggregations, groupbys, duplicate-eliminating projections (**distinct**), and duplicate-preserving projections in a common unified framework. Using *GPs* we extend well known and simple algorithms for SQL queries that use **distinct** projections to derive algorithms for queries using aggregations like **sum**, **max**, **min**, **count**, and **avg**. We develop powerful query rewrite rules for aggregate queries that unify and extend rewrite rules previously known in the literature. We then illustrate the power of our approach by solving a very practical and important problem in data warehousing: how to answer an aggregate query on base tables using materialized aggregate views (summary tables).

1 Introduction

With the growing number of large data warehouses for decision support applications, efficiently executing *aggregate queries* (queries involving aggregation) is becoming increasingly important. Aggregate queries are frequent in decision support applications, where large history tables often are joined with other tables and aggregated. Because the tables are large, better optimization of aggregate queries has the potential to result in huge performance gains. Unfortunately, aggregation operators behave differently from standard relational operators like select, project, and join. Thus, existing rewrite rules for optimizing queries almost never involve aggregation operators.

To reduce the cost of executing aggregate queries in a data warehousing environment, frequently used aggregates are often precomputed and materialized. These materialized aggregate views are commonly referred to as *summary tables*. Summary tables can be used to help answer aggregate queries other than the view they represent, potentially resulting in huge performance gains. However, no algorithms exist for replacing base relations in aggregate queries with summary tables so the full potential of using summary tables to help answer aggregate queries has not been realized. This paper makes three contributions to efficiently answering aggregate queries.

- We propose a framework for treating aggregation operators as an extension of duplicate-eliminating projection operators.
- Using this framework we present more powerful query rewrite rules for aggregation operators than rules known previously.
- Utilizing our rewrite rules, we extend existing work on answering queries using materialized views by giving an algorithm for answering aggregate queries using materialized aggregate views.

1.1 Optimizing Aggregations

Viewing aggregation as an extension of duplicate-eliminating (**distinct**) projection provides very useful intuition for reasoning about aggregation operators in query trees. Rewrite rules for duplicate-eliminating projection often can be used as building blocks to derive rules for the more complex case of aggregation. In addition to the intuition obtained by viewing aggregation as extended duplicate-eliminating projection, modeling both with one operator makes sense from

*Work was supported by NSF grants IRI–91–16646 and IRI–92–23405, by ARO grant DAAL03–91–G–0177, and by Air Force Grant F33615–93–1–1339 Authors' address: Department of Computer Science, Stanford University, Stanford, CA 94305-2140. Email: {agupta,venky,quass}@cs.stanford.edu

Proceedings of the 21st VLDB Conference
Zurich, Swizerland, 1995

an implementation point of view. Typically, in existing query optimizers both aggregations and duplicate-eliminating projections are implemented in the same module [G93].

We present a set of query rewrite rules for moving aggregation operators in a query tree. Other authors have previously given rewrite rules for pulling aggregations up a query tree [Day87, CS95] and for pushing aggregations down a query tree [CS94, YL94]. Our work unifies their results in a single intuitive framework, and using this framework we derive more powerful rewrite rules. We present new rules for pushing aggregation operators past selection conditions (and vice-versa) and show how selection conditions with inequality comparisons can cause aggregate functions to be introduced into or removed from a query tree. We also present rules for coalescing multiple aggregation operators in a query tree into a single aggregation operator, and conversely, rules for splitting a single aggregation operator into two operators.

1.2 Materialized Views with Aggregates

The new rewrite rules we present have enabled us to develop an algorithm for determining whether materialized aggregate views can be used to answer aggregate queries. The algorithm uses the rewrite rules to transform a query tree containing aggregation operators into an equivalent tree with some or all of the base relations replaced by materialized views. Previous work on answering queries with materialized views has dealt only with simple Select-Project-Join (SPJ) type queries and views without aggregation [CKPS95, LMSS94]. Our algorithm is a novel and important result for efficiently executing aggregate queries using preexisting materialized aggregate views.

Currently, to use a materialized view in a query, the view must be specified explicitly in the `FROM` clause. However, requiring that materialized views be specified in the `FROM` clause puts the onus on the query writer to be aware of all available views and to know whether using the views is more efficient than querying the base relations. A better approach is to allow the query optimizer to choose which materialized views are used in answering a query. Using our algorithm a query optimizer can transform an aggregate query tree over base relations into a query that incorporates materialized aggregate views and choose the most efficient tree. Our algorithm can easily be integrated into a conventional query optimizer using the approach for SPJ-type queries and views developed in [CKPS95].

1.3 Outline of Paper

Section 2 motivates our work with an example showing how an aggregate query tree can be transformed into a more efficient query tree that takes advantage of a materialized aggregate view. The example illustrates how our framework for reasoning about aggregation and the rewrite rules we present can be brought together into an algorithm for solving an important and practical problem.

The body of this paper is divided into three sections describing our three contributions. Section 3 presents our framework for reasoning about aggregation. The query rewrite rules are given in Section 4. The algorithm for transforming an aggregate query into one that uses materialized views is given in Section 5. It is possible to read later sections first, referring back to previous sections as necessary.

Related work is discussed in Section 6. We give our conclusions in Section 7. [GHQ95] is the full version of this paper and is available online.

2 Motivating Example

We give an example showing how a materialized aggregate view can be used to help answer an aggregate query. We do not explain the query rewrite rules or the algorithm used for transforming the query tree in this section. We revisit this example and explain the transformations involved when we describe our algorithm in Section 5.

EXAMPLE 2.1 Consider a data warehouse with historical sales data for a large chain of department stores. The data warehouse has the following relations.

```
item(item_id, item_name, category,
        manufacturer, our_cost)
store(store_id, street_addr, city, state)
sales(sales_id, item_id, store_id,
          month, year, sale_amt)
```

The first attribute of each relation is a key for the relation. The `item` relation contains information about each item that is stocked. The `our_cost` attribute contains the wholesale cost of the item. The `store` relation contains the address of each store. The `sales` relation contains one tuple for every sale that is made. Due to periodic discount and clearance sales, the sale amount of items sold is not functionally determined by `item_id`. It is instead stored in the `sales` relation. The relations have the following characteristics.

- There are 1000 items in the `item` relation, 10 of which are in the toy category.
- There are 1000 stores in the `store` relation, 100 of which are in the state of California.
- There are 10 years worth of sales in the `sales` relation, from 1986 through 1995.
- On average each store sells each item 200 times a year, resulting in two billion entries in the `sales` relation.

Suppose one wants to know if toy sales made by stores in the state of California have been going up or down during the past five years. This type of query, aggregating large amounts of data, is typical of decision support applications. The following SQL query can be written to calculate total sales of all toys in all California stores by year. The expression tree corresponding to this query appears in Figure 1. In the figure, each arc of the query tree has been annotated with the number of tuples flowing up the arc. We assume uniform selectivity of the selection conditions. The sizes of intermediate results is often a good predictor of query execution time, so we annotate the arcs to compare the query trees before and after using the materialized view. In the annotations, we abbreviate billion as "B," million as "M," and thousand as "K."

```
SELECT year, sum(sale_amt)
FROM   sales, store, item
WHERE  sales.store_id = store.store_id
  AND  sales.item_id = item.item_id
  AND  sales.year >= 1991
  AND  item.category = "toy"
  AND  store.state = "CA"
GROUPBY  year
```

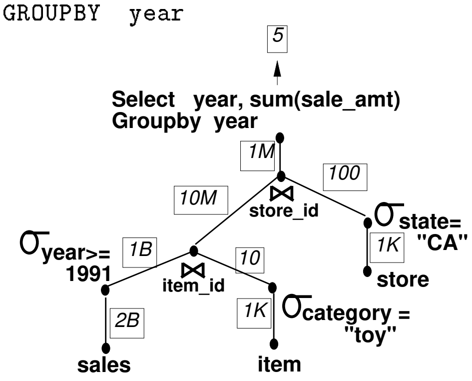

Figure 1: Query tree to compute total toy sales for California stores by year

Now suppose a yearly_sales view is materialized, listing the total yearly sales by item and store for stores in the state of California. The view definition appears below. The tree corresponding to the view definition appears in Figure 2.

```
CREATE VIEW yearly_sales AS
SELECT sales.store_id, sales.item_id,
       sales.year, SUM(sale_amt) AS total
FROM   sales, store
WHERE  sales.store_id = store.store_id
  AND  store.state = "CA"
GROUPBY sales.store_id, sales.item_id,
        sales.year
```

Notice that the materialized view involves the relations `sales` and `store`, while the query involves the relations `sales`, `store`, and `item`. Starting with the query tree of Figure 1, by reordering the joins and using our rewrite rules (see Section 4.3) to push the aggregation down past the topmost join, the tree in

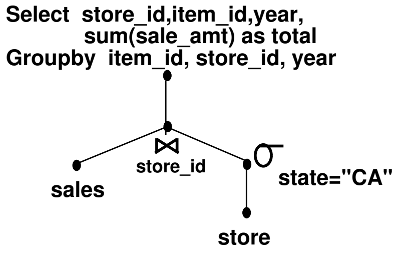

Figure 2: Query tree corresponding to yearly_sales view definition

Figure 3 is obtained. Using our algorithm for answering aggregate queries using materialized aggregate views (see Section 5), we can now transform this query tree into one that uses the `yearly_sales` materialized view, shown in Figure 4. Since the number of tuples in `yearly_sales` is several orders of magnitude less than the number of tuples in `sales`, the query tree using the materialized view is likely to be much more efficient than the query over the base relations. Using our rewrite rules and algorithm, a cost-based optimizer could generate both trees and select the best one. □

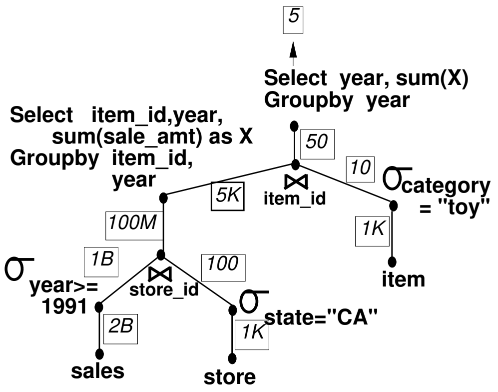

Figure 3: Query tree after reordering joins and pushing an aggregation down past the topmost join

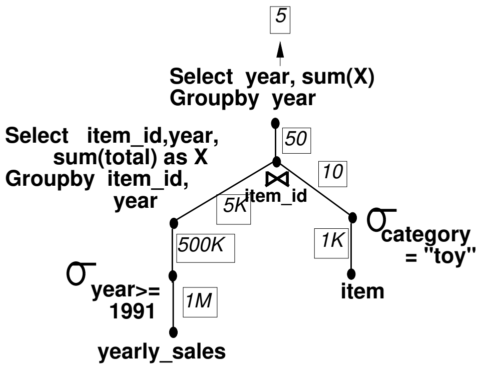

Figure 4: Query tree after integrating materialized view

3 GP Framework

This section defines *GP*s and then state some properties of *GP*s that are used in later sections.

3.1 GP Definition

A central theme of this paper is that algorithms for optimizing duplicate elimination can be extended to handle aggregation-groupby operators. Duplicate-eliminating projection, also referred to as **distinct** projection, is the simplest form of aggregation because it can be expressed as a simple **groupby** statement that does not compute any aggregates. Thus, we introduce a generalized projection operator (GP), denoted by the same operator π as we use for **distinct** projections. This extension of notation is appropriate, since a *GP* with no aggregate components behaves exactly like a **distinct** projection, *i.e.*:

$$\textbf{select distinct } D \textbf{ from } R \equiv GP : \pi_D \ (R(D,S)) \equiv \textbf{select } D \textbf{ from } R \textbf{ groupby } D$$

In general, a *GP* takes as its argument a relation R and produces a new relation according to the subscript of the *GP*. The subscript has two parts:

1. A set of groupby components. We refer to them as components and not attributes because they may be functions of attributes and not just attributes. For instance, the *GP* $\pi_D(R)$ is written as the following SQL query:

   select D **from** R **groupby** D.

2. A set of aggregate components. For example, we can write the *GP* $\pi_{D,\mathbf{max}(S)}\ (R)$ as the query:

   select D, $\mathbf{max}(S)$ **from** R **groupby** D.

   Here D is the only groupby component and $\mathbf{max}(S)$ is the only aggregate component.

For *GP*s that use only SQL aggregate components like **max** or **sum**, the equivalent SQL query is obtained by copying the entire subscript of the *GP* as the **select** clause of the SQL query and by copying the groupby components as the arguments of the **groupby** clause of the SQL query.

We use a different symbol π^{dup} to denote conventional projections that preserve duplicates. Section 4.3.2 discusses how to use *GP*s to represent these projections as well. Thus, *GP*s capture **distinct** projections, aggregate computations, and duplicate-preserving projections.

Based on the aggregate components of a *GP* we classify *GP*s into two categories:

- *duplicate insensitive*: The generation or removal of duplicate tuples in their input does not affect the result of such *GP*s. E.g. **distinct** projections and aggregations like **max** and **min**.

- *duplicate sensitive*: Duplicates must be preserved in their input. E.g. aggregations **sum** and **count**.

3.2 Properties

The following properties of *GP*s should be noted:

- If a *GP* uses only SQL aggregate components like **max** or **sum**, then the *GP* can be expressed using one SQL aggregate query. In the general case *GP*s may have non-SQL aggregate components requiring multiple SQL queries to express them (refer to Example 4.1).

- The **groupby** components of a *GP* are the key of the result. Thus, a *GP* outputs exactly one tuple for each value of the groupby components and *produces no duplicates in its output.*

- If the **groupby** components of a *GP* include a key of the input of the GP, then we can always rewrite the *GP* to have no aggregate components.

3.3 Scope Of Results

We develop rules for transforming aggregate queries and for answering aggregate queries using aggregate materialized views. We consider queries and views whose query trees have the following nodes: selection nodes, cross-product nodes, and *GP* nodes. For ease of exposition we represent join nodes as a cross-product followed by a selection. The query trees we consider in this paper represent SQL queries having the **select**, **from**, **where**, **groupby**, **having** keywords. We do not consider correlated subqueries. The relations in the **from** clause can be base relations or views. The views can themselves be single block SQL queries with the same structure. The query tree may thus have nested aggregates. We consider aggregates where the aggregate over a set of tuples S can be computed from aggregates over subsets of S. The SQL aggregates **max**, **min**, **sum**, **count** are examples (we handle **avg** by expressing it in terms of **sum** and **count**).

4 GP Transformations

4.1 Overview

This section considers three important query transformations

- pushing *GP*s down query trees.

- pulling *GP*s up query trees.

- coalescing two *GP*s into one, or equivalently, splitting up a *GP* into two.

These transformations are not independent of each other but are derived from the same underlying query equivalences.

We outline below our approach at deriving the transformation rules. We start by extending the rules for **distinct** projections to duplicate-insensitive *GP*s. Then we extend the rules for duplicate-insensitive *GP*s to derive those for duplicate-sensitive *GP*s. Our approach enables us to derive more powerful rules than those previously known.

Duplicate-insensitive aggregation has two components: (a) fragmenting the input relation into groups of tuples (b) for each group computing the required aggregate function. A distinct projection involves step (a) only. When transforming an aggregate query, the *GP*s interact with other operators like selections, other *GP*s, and cross-products. The transformation rules for distinct projections address the interaction of step (a) with these operators. We extend these transformation rules to consider how step (b) interacts with these operators.

Note, step (a) is done on a tuple-by-tuple basis and thus is a tuple-based operation. On the other hand, step (b) deals with sets of tuples at a time and thus is a relation-based operation. Relational operators like selections and distinct projections are tuple-based and hence relation-based computations cannot be "folded into" them. Therefore, to obtain transformation rules for duplicate-insensitive *GP*s we extend the rules for transforming queries with **distinct** projections to handle this mismatch in computation types. In particular, for the push-down and pull-up transformations we augment the corresponding **distinct** transformations for the case where a selection predicate involves aggregation attributes. In the coalescing transform, we add rules for the case when an attribute created by an aggregate computation is used in another *GP*, since we want to move this computation into the other *GP*.

Now consider duplicate-sensitive *GP*s. Such *GP*s behave differently from duplicate-insensitive *GP*s only when duplicates are generated or destroyed. Of the relational operators, only cross product nodes introduce duplicates because a cross product node causes tuples on either incoming branch to get repeated many times, above the cross product. Thus, the transformations for duplicate-insensitive *GP*s are extended only for the cross-product node case to get the corresponding transformations for duplicate-sensitive *GP*s. The interaction of a duplicate-sensitive *GP* with other relational operators is similar to that of a duplicate-insensitive *GP*.

In this paper we also show a close relationship between duplicate-insensitive *GP*s and selection predicates which involve the arithmetic comparisons $>, \geq, <, \leq$. We use such comparisons to generate or remove aggregations like $\mathbf{max}, \mathbf{min}$ in certain cases.

4.2 Arithmetic Comparisons: The $\top$ And $\perp$ Functions

This section discusses the interaction of *GP*s with selection conditions that use arithmetic comparisons by discussing what happens if *GP*s are pushed below such selections. The ideas developed herein also apply to *GP* pull-up and coalescing.

Consider any attribute that occurs in a query tree. If the attribute is needed as an output of the query tree, we cannot delete any distinct value of this attribute by pushing *GP*s down. Similarly, all distinct values of an attribute are required if the attribute participates in an equality predicate $(=, \neq)$. For instance, consider a generalized projection π_A being pushed down a query tree below a selection predicate $\sigma_{B=C}$. Since we require all distinct values of B, C to make the comparison, and we need all the distinct values of A in the answer, a new *GP* $\pi_{A,B,C}$ is introduced below the selection predicate. However, if the attribute only occurs in an arithmetic comparison $(<, >, \geq, \leq)$, we can do better. The following example suggests how.

EXAMPLE 4.1 Consider a *GP* π_A being pushed down a query tree below a selection predicate $\sigma_{B \geq C}$. The *GP* π_A says that above the selection predicate, only distinct values of attribute A are needed. Let the the selection predicate take as input the relation $R(A, B, C)$.

Eventually we want all those A values that have associated with them a B and C value that satisfies $\sigma_{B \geq C}$. Now consider two tuples of R: $t1 = (a, 40, 10)$ and $t2 = (a, 30, 20)$. Since $t1.B \geq t2.B$ and $t1.C \leq t2.C$, whenever $t2$ satisfies the selection predicate so does $t1$. Now, both tuples contribute the same value of A to the answer, and the answer does not retain duplicates. Thus, $t1$ can be discarded even before the selection node, without affecting the final answer. We can thus prune the relation R to remove irrelevant tuples such as $t1$. □

In the above example the attributes B, C are merely "filters" and their actual values are not important. We create the new aggregate functions $\top, \perp$ when we push down *GP*s below selection nodes with such filters. In Example 4.1, on pushing π_A below $\sigma_{B \geq C}$ we get the new *GP* $\pi_{A,\top(B),\perp(C)}$. The *GP* $\pi_{A,\top(B),\perp(C)}(R)$ says that we can discard any tuple s in R, if there exists another tuple t such that $s.A = t.A$, $s.B \leq t.B$, and $s.C \geq t.C$.

Consider now the *GP* $\pi_{A,\top(B)}$. It says we can discard any tuple s if there exists another tuple t such that $s.A = t.A$ and $s.B \leq t.B$. In particular, this means that we only need to keep the $\mathbf{max}(B)$ value for each value of A. In other words:

$$\pi_{G,\top(B)} = \pi_{G,\mathbf{max}(B)}$$

where G is a set of **groupby** attributes. A similar equality relates $\perp$ and **min**.

While the functions $\top$ and $\perp$ seem the same as **max** and **min** respectively, there are some important differences. Since $\pi_{A,\top(B),\perp(C)}$ merely prunes the relation $R(A, B, C)$, we have the following important property of $\top$,$\perp$ operators:

$$\pi_{G,\top(B),\perp(C)}(R(G, B, C)) \subseteq R(G, B, C)$$

where G is a set of **groupby** attributes. The functions **max** and **min** do not satisfy this property. To see this, consider $R(A,B,C) = \{(a,40,20),(a,30,10)\}$. $\pi_{A,\mathbf{max}(B),\mathbf{min}(C)}(R(A,B,C)) = \{(a,40,10\}$ which is not a subset of R. Thus in general, $\pi_{A,\top(B),\bot(C)} \neq \pi_{A,\mathbf{max}(B),\mathbf{min}(C)}$.

We can replace $\top$ ($\bot$) by **max**(**min**) in *GP*s that have no other aggregate components. In the presence of other aggregate components the rules for evaluating $\top$ ($\bot$) are more involved. The $\top, \bot$ operators and their algebra are explained in greater detail in [HG94].

4.3 GP Push-down

Now we discuss pushing *GP*s down query trees and examine the interaction of *GP*s with the different types of nodes in the query tree.

4.3.1 Duplicate-Insensitive *GP*s

Selection Nodes: Duplicate-insensitive *GP*s behave similarly to **distinct** projections if the selection predicate does not contain an attribute used to compute an aggregation. To see why, consider pushing a **distinct** projection π_A down below the selection node $\sigma_{C=D}$. On push-down, we get the new projection $\pi_{A,C,D}$ and we keep the original projection above the selection node, since the attributes C, D do not appear in the output. The new projection $\pi_{A,C,D}$ does a "partial" duplicate elimination: for each C, D value we eliminate all duplicate values of A. The original projection π_A then does the total duplicate elimination. The above technique of doing partial and total duplicate elimination is applicable to aggregations like **max**, **min** too. By a similar reasoning, on pushing $\pi_{A,\mathbf{max}(B)}$ below $\sigma_{C=D}$, we get the new *GP* $\pi_{A,C,D,\mathbf{max}(B)}$. The new *GP* computes the partial maxima and the original *GP*, the total maxima. Thus when the selection predicate does not involve aggregation attributes, we have the same rule as **distinct** projections for all duplicate-insensitive projections:

PDRule 1 *When a selection predicate does not involve any attribute used in aggregation computation, we can push a GP P below it. We add the attributes occurring in the selection predicate as* **groupby** *attributes, to get the new GP Q and keep the original GP P above the selection.* ⊙

As a general rule, if the new *GP*, after push-down, is no different from the original *GP* then the original *GP* can be discarded on push-down because the partial aggregation is the same as the total aggregation. Thus, as a special case of Rule 1 we obtain the following query equivalence:

$$\pi_{G,H}(\sigma_{f(G)}) = \sigma_{f(G)}(\pi_{G,H}) \qquad (1)$$

where G and H are the **groupby** and aggregate components respectively.

By using push-down rule 1 and by adding attributes to the original *GP* to get the new *GP*, we may cause an attribute to appear in multiple components of the *GP*. We rewrite such *GP*s where possible by dropping the redundant component.

EXAMPLE 4.2 Consider pushing the *GP* $\pi_{A,X=\mathbf{max}(B)}$ below the selection $\sigma_{B=C}$. By push-down rule 1, we get the new *GP* $\pi_{A,B,C,X=\mathbf{max}(B)}$. We write this *GP* as $\pi_{A,B,C}$ since computing $\mathbf{max}(B)$ is redundant. □

In the presence of certain arithmetic comparisons, we can create and push aggregation computations (Section 4.2). In particular, we saw that we can add the $\top$ and $\bot$ functions as follows:

PDRule 2 *When a selection predicate is of the form $B \geq C$ or $B > C$, on pushing a GP P below the selection, we get the new GP Q which is P with the additional components $\top(B)$ and $\bot(C)$.* ⊙

Push-down rule 2 extends previous work on pushing down aggregation computation and allows us to create aggregation computation in queries that had none to start with.

Example 4.2 also illustrates that in general it is not possible to push aggregate computations when the selection predicate involves an aggregation attribute. We cannot compute partial maxima, since we need all distinct values of B for the selection. However in certain cases, by using the $\top$ and $\bot$ functions, we can indeed push aggregation computations even if the selection predicate involves an aggregation attribute. The following example illustrates this:

EXAMPLE 4.3
Consider pushing the *GP* $\pi_{A,X=\mathbf{max}(B)}$ below the selection $\sigma_{B\geq 10}$. By push-down rule 2, the new *GP* created is $\pi_{A,X=\top(B)}$. Now since there is no other aggregate computation in the *GP*, we can replace the $\top$ by a **max** to get $\pi_{A,X=\mathbf{max}(B)}$. □

Example 4.3 leads to the following commutativity relationship:

$$\pi_{G,X=\mathbf{max}(B)}(\sigma_{B\geq C}) = \sigma_{X\geq C}(\pi_{G,X=\mathbf{max}(B)})$$

Note G is a set of **groupby** attributes and C is an attribute belonging to the set G. If the *GP* being pushed has other aggregate components, then the above relationship does not hold. A similar result holds for $\bot$ and $\leq, <$.

Cross-Product Nodes: For duplicate-insensitive *GP*s it does not matter if the aggregation computation is done above or below the cross product, since the presence of duplicate attribute values does not affect the result of such computations. The following example illustrates this fact.

EXAMPLE 4.4 Let $\pi_{A,B,X=\mathbf{max}(C),Y=\mathbf{max}(D)}$ be pushed through a cross product. Let attributes A, C belong to the left branch and attributes B, D to the right branch. On pushing $\pi_{A,B,X=\mathbf{max}(C),Y=\mathbf{max}(D)}$ we get $\pi_{A,X=\mathbf{max}(C)}$ in the left branch and $\pi_{B,Y=\mathbf{max}(D)}$ in the right branch and we drop the original GP. $\Box$

We have the same rule as for **distinct** projections:

PDRule 3 *On pushing a GP P through a cross-product node, we drop P and get the new GPs Q_{left} and Q_{right} on the left and right branches respectively. $Q_{left}(Q_{right})$ contains the components of P whose attributes are from the left (right) branch.* $\odot$

Push-down rule 3 also extends previous work on pushing down aggregations.

GP **nodes:** If while being pushed down, a *GP* encounters another GP, we attempt to coalesce the two *GP*s into one. The rules for coalescing are given in Section 4.5. The interaction of *GP*s with conventional projections that preserve duplicates during push down is syntactic (renaming of attributes) just as with pushing **distinct** projections below duplicate-preserving projections.

4.3.2 Duplicate-Sensitive *GP*s

The output of a generalized projection is always a set, *i.e.*, there are no duplicates. It is still possible to use a *GP* to write a conventional projection that outputs duplicates. We use π^{dup} to denote a conventional projection that preserves duplicates in its output. Consider now the *GP* $\pi_{A,X=\mathbf{count}(*)}$ and the conventional projection π^{dup}_{A}. We do not lose any information by dropping duplicates and incorporating a **count** column to indicate multiplicity because the two forms are equivalent. We introduce the "expand" operator e to express this equivalence:

$$\mathsf{e}_X(\pi_{A,X=\mathbf{count}(*)}) = \pi^{dup}_{A}$$

Consider a relation $R(A,B,X)$, for each tuple (a,b,x) in R, the expand operator e_X outputs x copies of the tuple (a,b). Thus we do not need to change the output semantics of *GP*s to accommodate duplicates. *GP*s only produce sets as outputs (no duplicates). Duplicate semantics are simulated using **count** and the expand operator e.

Properties of the Expand Operator e: The expand operator helps us understand aggregations that are sensitive to duplicates in their input. Usually e is pulled up a query tree; so we mention some relevant properties of the expand operator.

It can be seen that the expand operator commutes (can be pulled up) with selection and cross product nodes. Also the expand operator can be discarded when it encounters above it a duplicate-insensitive *GP*. The interactions of interest arise when an expand operator encounters a duplicate-sensitive *GP*. For the aggregate **sum**:

$$\pi_{A,Y=\mathbf{sum}(B)}(\mathsf{e}_X) = \pi_{A,Y=\mathbf{sum}(B*X)}$$

since X just indicates how many times each B value is repeated. The aggregate **count** can be thought of as being **sum**(1) which gives us

$$\pi_{A,Y=\mathbf{count}(*)}(\mathsf{e}_X) = \pi_{A,Y=\mathbf{sum}(X)}$$

We use these results in explaining the push down algorithm for aggregates like **sum** and **count**. The expand operator also lets us create **count** aggregations anywhere in a query tree and can be used to reduce the size of intermediate relations when there are many duplicate tuples [GHQ95]. Prior work on rewriting aggregate queries has not considered introducing aggregates in queries.

We now consider pushing a duplicate-sensitive *GP* down a query tree and examine its interactions with the different nodes present in the tree.

Selection Nodes: As mentioned in Section 4.1 duplicate-sensitive *GP*s follow the same basic algorithm as duplicate-insensitive *GP*s for selections. Push-down rule 1 thus applies to all *GP*s. However, as illustrated below, arithmetic comparisons cannot be used to create the $\top$ and $\bot$ functions with duplicate-sensitive *GP*s. So we cannot use push-down rule 2.

EXAMPLE 4.5 Consider pushing *GP* $\pi_{A,\mathbf{count}(*)}$ below a selection node $\sigma_{C\geq D}$. In this case for a given value of A we are not just interested in seeing if there exists some C,D that will cause the value of A to be selected. Instead we are interested in the number of times such a value will be selected. Adding $\top(C)$ and $\bot(D)$ to the projection pushed below $\sigma_{C\geq D}$ allows us to determine only if there exists some C,D such that $C\geq D$. In some sense we end up getting only a TRUE/FALSE answer for each A value where we want a number. So on pushing $\pi_{A,\mathbf{count}(*)}$ below $\sigma_{C\geq D}$ we get $\pi_{A,C,D,\mathbf{count}(*)}$ as the new *GP*. $\Box$

Cross-Product Nodes: When a duplicate-sensitive aggregation computation is pushed down one branch we have to account for the multiplicative effect of the other branch. Thus we cannot eliminate duplicates in the other branch if the GP is duplicate-sensitive.

EXAMPLE 4.6 Let $\pi_{A,A',X=\mathbf{sum}(B)}$ be pushed down through a cross product where attributes A,B go down the left branch and A' down the right. Since **sum** requires duplicates be preserved in its input we cannot eliminate duplicates in the right branch, and so must push the conventional duplicate-preserving projection $\pi^{dup}_{A'}$ rather than the *GP* $\pi_{A'}$. But we can replace $\pi^{dup}_{A'}$ by $\mathsf{e}_X(\pi_{A',X=\mathbf{count}(*)})$. We thus have the following expression after push down:

$$\pi_{A,A',X=\mathbf{sum}(L)}(\pi_{A,L=\mathbf{sum}(B)}\times\mathsf{e}_X(\pi_{A',X=\mathbf{count}(*)}))$$

Unlike with duplicate-insensitive *GP*s, we require the original *GP* to do the computation above the cross product. Now, pulling up the expand operator and merging into the *GP* above the cross product, we get

$$\pi_{A,A',X=\mathbf{sum}(L*X)}(\pi_{A,L=\mathbf{sum}(B)} \times \pi_{A',X=\mathbf{count}(*)})$$

Since A, A' are keys of the left and right branches (being **groupby** attributes), A, A' is a key of the relation above the cross product. Hence we can replace $\mathbf{sum}(L * X)$ by $L * X$ above the cross product. □

The detailed algorithm for *GP* push down is in [HG94].

4.4 Pulling Up *GP*s

We are often interested in pulling *GP*s up query trees for a number of reasons. One important reason is to express queries with aggregations in a normal form (Section 5.1). The pull-up rules are derivable from the push-down rules we saw earlier.

Selection Nodes: From query equivalence 1 we have the following rule:

PURule 1 *A GP can always be pulled up above a selection if all the attributes in the selection predicate occur as* **groupby** *attributes in the GP.* ⊙

From push-down rule 2 we have the following pull-up rule:

PURule 2 *If all the attributes in the selection predicate are* **groupby***, $\top$, or $\perp$ aggregate attributes of the GP and if the $\top(\perp)$ attributes occur to the left (right) of a $\geq$ or $>$ comparison in the selection predicate, we can pull up the GP.* ⊙

Cross-Product Nodes: From push-down rule 3 we have the corresponding pull-up rule:

PURule 3 *If we have two GPs that are duplicate-insensitive on either branch of a GP, we can pull them up as one GP, by combining all their attributes.* ⊙

We can derive the pull-up rule for duplicate-sensitive *GP*s from the push-down rule for duplicate-sensitive *GP*s. We state the rule without derivation.

PURule 4 *To pull up a GP above a cross product when the other branch of the cross product has no duplicates, we add all the attributes coming up the other branch as* **groupby** *attributes of the GP.* ⊙

The above rule is very general and applies to all *GP*s but is less powerful than pull-up rule 3 for duplicate-insensitive *GP*s.

When the other branch in the cross product has duplicate tuples we cannot pull up *GP*s as we did above. This statement applies even to the simple **distinct** projection.

EXAMPLE 4.7 Consider a cross product node with the **distinct** projection $\pi_{A,B}$ on the left branch and $R(C, D)$ on the right branch, where R is some relation that may have duplicates. Then,

$$\pi_{A,B}(S(A,B,F)) \times R(C,D) \neq. \\ \pi_{A,B,C,D}(S(A,B,F) \times R(C,D)).$$

□

We can use the *GP* pull-up rule 4 if R is made duplicate-free. There are many ways of making R duplicate-free. One method is to add key attributes (or unique tuple ids) to R. This method of adding keys or tuple ids to relations with duplicates is similar to the ADDKEYS rule in the Starburst Query Rewrite facility [PHH92] and the rule given in [Day87]. Another option to making R duplicate-free is to use the expand operator e. We can replace $R(C, D)$ with $\mathsf{e}_X(\pi_{C,D,X=\mathbf{count}(*)}(R(C,D)))$. We can now pull e_X up above the cross product as mentioned in section 4.3.2. After we pull up e we have $\pi_{C,D,X=\mathbf{count}(*)}(R(C,D))$ on the right branch, which has no duplicates. With these modifications, we can apply pull-up rule 4. Note, when we have duplicate-sensitive *GP*s on each branch of the cross product, we pull them up one at a time, using pull-up rule 4.

4.5 Coalescing *GP*s

This section gives rules for combining two *GP*s into one *GP* (coalescing) and thus the reverse, splitting a *GP* into two *GP*s (splitting). Coalescing and splitting are valuable tools in rewriting query trees with aggregations. Coalescing has not been considered by researchers before.

When we attempt to coalesce two *GP*s, we try to move all the computation from the lower *GP* into the upper *GP* and then to drop the lower *GP*. We assume that all the attributes occurring in the upper *GP* are output by the lower *GP*, because otherwise the original query is incorrect. In Section 4.1 we saw that the difference between **distinct** projections and other duplicate-insensitive *GP*s was in the inability to move relation-based computations into tuple-based operators. In coalescing, this inability translates to additional rules when the upper *GP* contains an attribute created by aggregate computation in in the lower GP. First the simple **distinct** projection rule:

CRule 1 *If the upper GP is duplicate-insensitive and all the attributes required by the upper GP occur as* **groupby** *attributes of the lower GP, we can discard the lower GP.* ⊙

What happens if an attribute A required by the upper *GP* is created by an aggregate computation at the lower *GP*? There are two cases to consider here: attribute A is a **groupby** attribute of the upper *GP* or attribute A is an aggregate attribute of the upper *GP*.

Consider the first case. As we saw in Section 4.1, **groupby** computations in a GP are tuple based, so we cannot move the aggregation computation of the lower *GP* into a **groupby** component of the upper *GP*.

CRule 2 *If a* **groupby** *attribute in the upper GP is created by an aggregate computation in the lower GP we cannot coalesce the two GPs.* ⊙

Consider now the second case: an aggregate attribute in the upper *GP* is created by an aggregation computation in the lower *GP*. We can coalesce *GP*s when the lower *GP* does a partial aggregation computation and the upper *GP*, the total aggregation. So for the aggregates like **sum**, **max**, **min** where partial aggregation and the total aggregation involve the same aggregate function, we have the following rule:

CRule 3 *If an aggregation attribute in the upper GP is created by an aggregation computation in the lower GP and if both aggregations are the same and are* **max**,**min**,**sum** *then we can do the aggregation computation in the upper GP.* ⊙

When we do a partial computation of **count**, the total computation of **count** requires the use of a **sum** aggregation. We can use the same principle as coalescing rule 3 above, to remove the partial computation.

If we can move all the aggregation computation into the upper *GP*, using coalescing rule 3, we can drop the lower *GP*. Otherwise we cannot coalesce the two *GP*s and must let them remain as they were originally. It is incorrect to move some aggregation computation up and not others.

5 Answering Queries Using Materialized Views

We now present an algorithm for transforming an aggregate query tree over base relations into one that uses a materialized aggregate view.

Given an aggregate query tree Q and the tree corresponding to a materialized aggregate view definition V, if the algorithm determines that Q can be answered using V it returns a modified query tree Q' such that $Q'(V) = Q$. The aggregate query tree Q may be a subtree of a larger query tree, so in general the algorithm can be applied to several subtrees of a large query tree, resulting in the incorporation of several materialized views.

The *GP* framework and transformation rules have proven very useful in the development of our algorithm. Rewrite rules for moving *GP*s up a query tree allow us to transform the query and view definition into a normal form (described later) making reasoning about aggregation easier. Rewrite rules for pushing *GP*s down a query tree make it possible to obtain a tree rooted at a *GP* operator having the same base relations as the materialized view under consideration. Rewrite rules for pushing selection conditions through *GP*s and for splitting one *GP* into two *GP*s are used by the algorithm to transform the aggregate query tree into one that uses a materialized aggregate view.

Section 5.1 describes the class of queries and views handled by the algorithm. The algorithm is outlined in Section 5.2. Section 5.3 illustrates how the query tree of our motivating example (see Section 2) is transformed by the algorithm. Due to space limitations we are unable to present the full details of the algorithm. The details can be found in [GHQ95].

5.1 Preconditions on View V and Query Q

Our algorithm requires that the view and query be put into a *normal form* that has all aggregations and selections above all joins. The normal form makes reasoning about aggregation much easier than if the query had nested aggregations. The normal form consists of a a selection over a generalized projection over a selection over a set of joins, *i.e.*,

$$\sigma_h \pi \sigma_l \mathcal{X}$$

In the normal form σ_l and σ_h are conjunctive selection conditions, π is the *GP*, and the $\mathcal{X}$ symbol represents a set of join operations.

A large class of aggregate queries can be reduced to this normal form using the *GP* push-down and pull-up rules (see Section 4). In particular, **select-from-where-groupby-having** queries can be reduced to this normal form if the attributes in the **groupby** and **having** clauses appear in the **select** clause, no aggregate function definition uses the **distinct** keyword (*e.g.*, `SUM(DISTINCT sale_amt)`), and the **where** clauses are conjunctive. In addition, queries that include in the **from** clause one or more nested aggregate views can be rewritten in this form if the aggregates can be pulled above the joins and coalesced into a single *GP*.

We require that view V and query Q use the same set of relations $R_1, \ldots, R_m$ joined using the same join conditions. We refer to the *GP* and selection conditions in view V as $GP(V)$, $\sigma_h(V)$, $\sigma_l(V)$. Similarly for query Q.

Due to the undecidability of the implication problem for the class of aggregations we consider [RSSS94] there are cases where a query can be answered using a view that are not detected by the algorithm. However, the algorithm handles a very large class of queries and views that includes many common cases.

5.2 Algorithm

The input to the algorithm is a query tree Q and a tree for view V. Both trees must have been reduced to the prescribed normal form of Section 5.1 using our transformation rules. Note, since Q can be a subtree of a larger query tree, the algorithm is applicable to a larger class of queries that cannot be completely reduced to this normal form. If Q is a subtree of a larger query tree, before applying the algorithm it is useful to push as many selection conditions as applicable from the larger query tree into Q, because further restricting Q makes it more likely that Q is computable using V.

The output of the algorithm is either `FAIL` if the algorithm cannot determine that Q can be answered using V, or a modified query tree Q' over V instead of the base relations such that $Q'(V) = Q$.

Intuitively, Q' is derived by transforming Q such that the bottom portion of the tree is equivalent to the query tree for V[1] and the upper portion becomes the query tree Q'. The steps of the algorithm are outlined below.

In Step 1 we push selection conditions from σ_h down through the *GP* to σ_l for both the query and the view. [GHQ95] gives a table, derived from the rules in Section 4, that enumerates the cases when selection conditions can be pushed down past GPs. Selection conditions are pushed down in preparation for Step 2.

In Step 2 we test whether the selection conditions in the resulting $\sigma_l(V)$ are more restrictive than the selection conditions in the resulting $\sigma_l(Q)$. If so, then tuples that could appear in the groups formed by $GP(Q)$ would be filtered out by $\sigma_l(V)$, and the algorithm determines it is not possible to derive Q'.

In Step 3(a) we transform query tree Q to include a *GP* operator similar to $GP(V)$. If the groupby components of $GP(Q)$ are a proper subset of the groupby components of $GP(V)$ and $\sigma_h(V)$ is empty, then the groups created by V partition the groups needed in Q. We can therefore combine the groups created by V into the groups needed by Q using a *GP* operator. We split $GP(Q)$ into two *GP*s, $GP_{bot}(Q)$ and $GP_{top}(Q)$. $GP_{top}(Q)$ does the same computation as the original $GP(Q)$. $GP_{bot}(Q)$ has the same groupby components as $GP(V)$. An enumeration of the GP-splitting rules is given in [GHQ95].

If the groupby components of $GP(Q)$ and $GP(V)$ are the same, then $GP(Q)$ is not split and $GP_{bot}(Q)$ and $GP_{top}(Q)$ in the remainder of the algorithm both refer to $GP(Q)$.

If there are additional groupby components in $GP(Q)$ then Q is grouping at a different (or finer) granularity than V, and the algorithm determines it is not possible to derive Q'.

In Step 3(b) we test whether each aggregate component of $GP_{bot}(Q)$ is computable from the aggregate components of $GP(V)$. If not, then the algorithm determines it is not possible to derive Q'.

In Step 4 we identify conditions in $\sigma_l(Q)$ that are not implied by $\sigma_l(V)$ and try to pull them up past $GP_{bot}(Q)$. If a selection condition in $\sigma_l(Q)$ is not implied by the selection conditions in $\sigma_l(V)$ and it cannot be pulled up past $GP_{bot}(Q)$, then tuples that could appear in the groups formed by $GP(V)$ would be filtered out by the conditions of $\sigma_l(Q)$, and the algorithm determines it is not possible to derive Q'.

In Step 5 we test whether the selection conditions in $\sigma_h(V)$ are more restrictive than the selection conditions in $\sigma_h(Q)$. If so, then tuples that could appear in the result of Q would be filtered out by the conditions of $\sigma_h(V)$, and the algorithm determines it is not possible to derive Q'.

After Step 5 the view tree V is equivalent to the subtree of the transformed query tree Q rooted at $GP_{bot}(Q)$.[2] The algorithm returns as Q' the transformed query tree Q with the subtree rooted at $GP_{bot}(Q)$ replaced by the materialized view.

We have omitted several enhancements that extend the algorithm to cover additional cases when Q' can be derived. The enhancements can be found in [GHQ95].

5.3 Example

We illustrate how the algorithm can be applied to the query and view from Example 2.1.

EXAMPLE 5.1 Consider again Example 2.1. A query is posed to compute total sales of all toys in all California stores for each year beginning with 1991. The initial query tree for the query appears in Figure 5 using our *GP* notation. Materialized view `yearly_sales` computes total yearly sales by item and store for stores in California (Figure 2). We want to determine if view `yearly_sales` can be used to answer the query.

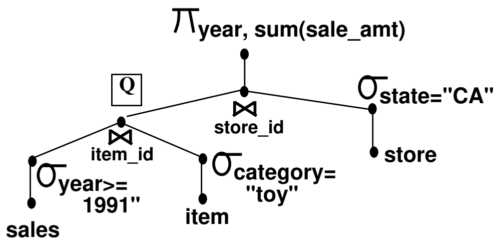

Figure 5: Initial query

Note that the query is over the relations `sales`, `store`, and `item`, while the `yearly_sales` view is only over relations `sales` and `store`. We reorder the joins in the query so that `sales` is joined first with `store`, then with `item`. To facilitate join reordering, all *GP*s are first pulled up above all joins in the query using the rules for *GP* pull-up. In our example the single *GP* is already above all joins. Next we use the *GP* pushdown rules from Section 4.3 to push the *GP* down past the topmost join to the subtree that contains only the `sales` and `store` relations. We refer to the subtree rooted at this new *GP* as Q.

Before applying our algorithm we put the query trees in the normal form described in Section 5.1. V is normalized by pulling up the selection condition on

[1] Actually, it is possible for the bottom portion of the query tree to return *a superset* of the tuples returned by V, so long as additional selection conditions in the upper portion of the query tree filter out the additional tuples.

[2] It is possible for the subtree rooted at $GP_{bot}(Q)$ to return additional tuples as mentioned in an earlier footnote.

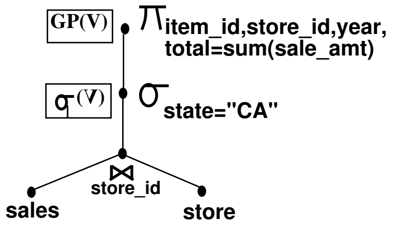

Figure 6: Normalized view `yearly_sales`

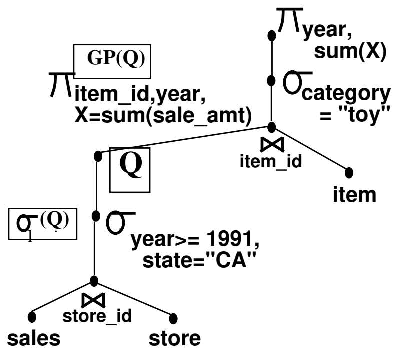

Figure 7: Query tree after normalizing subtree Q

`state` above the join (Figure 6). Q is normalized by pulling up the selection conditions on `year` and `state` above the join. Figure 7 shows the entire query tree after subtree Q has been normalized. Note that for our example $\sigma_l(Q)$ is `year >= 1991 AND state = "CA"`, $\sigma_l(V)$ is `state = "CA"`, and $\sigma_h(Q)$ and $\sigma_h(V)$ are both empty.

We now apply the algorithm. Step 1 can be skipped since $\sigma_h(Q)$ and $\sigma_h(V)$ are both empty.

All selection conditions in $\sigma_l(V)$ are also in $\sigma_l(Q)$ so the test in step 2 succeeds.

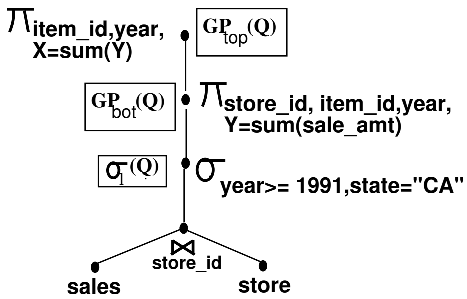

Figure 8: Subqtree Q after splitting $GP(Q)$

Step 3(a) identifies that the groupby components of $GP(Q)$ (`item_id` and `year`) are a proper subset of the groupby components of $GP(V)$ (`item_id`, `year`, and `store_id`). $GP(Q)$ is split into $GP_{top}(Q)$ and $GP_{bot}(Q)$ as shown in Figure 8.

The aggregate function $Y = \mathbf{sum}(sale\_amt)$ of $GP_{bot}(Q)$ can be obtained from $total = \mathbf{sum}(sale\_amt)$ of $GP(V)$ so the test in step 3(b) succeeds.

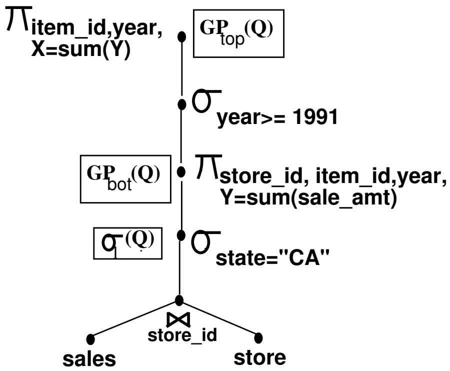

Figure 9: Pulling up year $\geq$ 1991 after splitting $GP(Q)$

Step 4 identifies that the condition `year >= 1991` in $\sigma_l(Q)$ is not implied by $\sigma_l(V)$. Since `year` is a groupby component of $GP_{bot}(Q)$ the condition can be pulled up above $GP_{bot}(Q)$, yielding the query tree in Figure 9.

Both $\sigma_h(Q)$ and $\sigma_h(V)$ are empty so the test in step 5 succeeds.

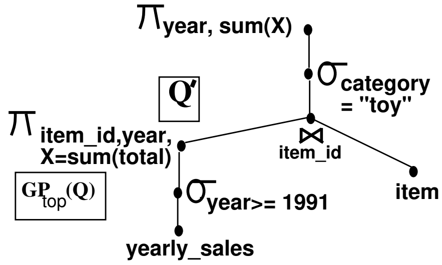

Figure 10: After replacing subtree Q with Q'

At this point the subtree rooted at $GP_{bot}(Q)$ is identical to V, so the algorithm derives Q' by replacing the subtree rooted at $GP_{bot}(Q)$ with view `yearly_sales`. Figure 10 shows the original query tree with subtree Q replaced by Q'.

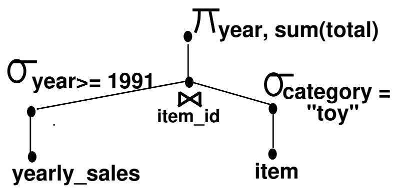

Figure 11: Optimizing after replacing Q with Q'

The resulting query tree can be further transformed. For instance, the GP on the `yearly_sales` materialized view can be pulled up and coalesced with the GP at the top of the query tree to yield the tree shown in Figure 11. □

6 Related Work

Answering queries using materialized views has been studied in [LY85, YL87, LMSS94, CKPS95]. However, existing work has not considered queries that use aggregation. Aggregates are an important extension to the previously considered queries because aggregations are at the heart of decision support and warehousing.

The query transformations we give unify and properly subsume the push-down transformations given in [CS94, YL94] and the pull-up transformations given in [Day87, CS95]. In particular, we give new transformations for the following cases:

- We use certain arithmetic comparisons to create aggregations in query trees that have none to start with and delete aggregations in those that do.
- By treating duplicate-insensitive *GP*s differently from duplicate-sensitive *GP*s, we can infer more powerful transformation rules for duplicate-insensitive *GP*s. For example, we can pull up two duplicate-insensitive *GP*s simultaneously past a cross product.
- We can introduce the **count** aggregation anywhere in a query tree using the expand operator.
- We can push aggregation down both branches of a cross product.
- We can coalesce and split aggregations.

[CS94, CS95] discuss how to integrate aggregation push-down and pull-up into a system-R style query optimizer.

7 Conclusions

In this paper we present a new framework for reasoning with **groupby** and aggregation in SQL queries. We generalize **distinct** projections to yield the notion of "generalized projections" that capture **groupby** and aggregation computations. *GP*s also capture arithmetic comparison operators that are not expressible as SQL aggregate computations. The *GP* framework allows us to obtain many new and promising results. In this paper, we discuss two sets of results that we have obtained using the *GP* framework.

For aggregate queries we derive transformation rules that unify and generalize previously proposed transformation rules. The new rules we derive include rules for coalescing multiple aggregate computations into single computations, introducing and eliminating aggregate computations using arithmetic inequality selection conditions, and pushing aggregate computations down both branches of a cross product (or join).

We give an algorithm for a hitherto unsolved problem, namely, how to use materialized aggregate views to help answer aggregate queries. This algorithm is very useful for decision support applications in data warehousing environments. The algorithm is developed using the new transformation rules that we obtain using the *GP* framework.

In [GHQ95] we discuss how the transformations given in section 4 can be used by query optimizers to reduce the cost of query evaluation. We pick the Starburst query optimizer [PHH92] and mention how and where our transformations can be used. We also briefly discuss how the expand operator can be used in query optimization when there are relations with many duplicates.

Acknowledgements

We thank Prof. Jeff Ullman and Prof. Jennifer Widom for the many discussions on the ideas presented above. We also thank Anand Rajaraman, Surajit Chaudhuri, and Waqar Hasan for feedback.

References

[CS94] S. Chaudhuri and K. Shim. Including Group-By in Query Optimization. In *VLDB*, 1994.

[CS95] S. Chaudhuri and K. Shim. Optimizing Complex Queries: A Unifying Approach. Technical Memo HPL-DTD-95-20.

[CKPS95] S. Chaudhuri et al.. Query Optimization in the presence of Materialized Views. In *ICDE*, 1995.

[Day87] U. Dayal. Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers. In *VLDB*, 1987, pages 197–28.

[G93] G. Graefe. Query Evaluation Techniques for Large Databases. In *ACM Computing Surveys*, Vol. 25, No. 2, June 1993.

[H95] R. Hackathorn. Data WAREHOUSING Energizes Your Enterprise. In *DATAMATION*, Feb 1, 1995.

[HG94] V. Harinarayan and A. Gupta. Generalized Projections: A Powerful Query-Optimization Technique. Stanford Technical Report No. STAN-CS-TN-94-14.

[GHQ95] A. Gupta, V. Harinarayan and D. Quass. Aggregate-Query Processing in Data Warehousing Environments. http://www-db.stanford.edu/pub/harinarayan/1995/GP2.ps

[LMSS94] A. Y. Levy et al.. Answering queries using views. In *PODS*, 1995.

[PHH92] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *SIGMOD*, 1992.

[LY85] P. A. Larson and H.Z. Yang. Computing queries from derived relations. In *VLDB*, 1985.

[RSSS94] K. A. Ross, D. Srivastava, P. J. Stuckey, and S. Sudarshan. Foundations of Aggregation Constraints. In *Principles and Practice of Constraint Programming*, Orcas Island, WA, 1994.

[YL87] H. Z. Yang and P. A. Larson. Query transformation for PSJ-queries. In *VLDB*, 1987.

[YL94] W. P. Yan and P. A. Larson. Performing Group-By Before Join. In *ICDE*, 1994.