Generalized Projections: A Powerful Approach to Aggregation *

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

Abstract

In this paper we introduce *generalized projections* (GPs), an extension of duplicate-eliminating projections, that capture aggregations, groupbys, conventional projection with duplicate elimination (**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 about base tables using materialized aggregate views (summary tables).

Keywords: aggregation, data warehousing, materialized views, query optimization

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 exhibit a different kind of behavior from standard relational operators like select, project, and join. For this reason, rewrite rules for optimizing query trees almost never involve aggregation operators.

To reduce the cost of aggregate queries in a data warehousing environment, aggregate views are often materialized into *summary tables* [H95]. The use of summary tables to help answer aggregate queries also has the potential to result in huge performance gains. However, algorithms for replacing base relations in aggregate queries with summary tables do not exist, so the full potential of using summary tables to help answer aggregate queries has not been realized. This paper describes three contributions to the problem of efficiently executing aggregate queries.

- We propose an intuitive framework for aggregation operators as an extension of duplicate-eliminating projection operators.
- Using this framework we present query rewrite rules for aggregation operators that are more powerful than those given previously.

*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

†Current address: IBM Almaden Research Center, San Jose

- 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

Thinking about aggregation as an extension of duplicate-eliminating (**distinct**) projection provides very useful intuition into the problem of reasoning about aggregation operators in query trees. Rewrite rules for duplicate-eliminating projection can often be used as building blocks to derive rules covering the more complex case of aggregation. In addition to the intuition obtained by thinking of aggregation as extending duplicate-eliminating projection, modeling both with one operator makes sense from an implementation point of view. Both aggregation and duplicate-eliminating project are typically implemented in the same module in existing query optimizers [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] 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 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, RSU95]. Our algorithm is a novel and important result for efficiently executing aggregate queries using preexisting materialized aggregate views.

Materialized aggregate views have become increasingly important lately for a variety of reasons. Data warehouses for decision support applications often create a set of carefully chosen materialized aggregate views, called summary tables, so that most queries can be posed on the summary tables instead of the base relations [H95]. The base relations are often very large, so querying summary tables instead of base relations can be a huge performance gain. In addition, in some cases the base relations are so large that it is infeasible to store them entirely on disk, so the summary tables contain the only data easily available. Materialized aggregate views are also used frequently in data visualization and data mining applications to obtain fast access to aggregate data.

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 that using the views may be 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. By 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 simple SPJ-type queries and views developed in [CKPS95].

We have used our framework to extend results in another important area—determining when the result of one query is contained within the result of another query (the *query containment* problem of [Ull89]). We have obtained necessary and sufficient conditions for when the result of an aggregate query is contained within the result of another aggregate query for a limited class of queries and sufficient conditions for a larger class of queries. Due to space limitations we do not discuss our containment results in this paper.

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.

Another application of our framework and rewrite rules, optimizing aggregate queries, is considered in Section 6, where we discuss how the rewrite rules could be integrated into a conventional cost-based query optimizer. Related work is discussed in Section 7. We give our conclusions in Section 8.

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 category of computer hardware.
- 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 computer hardware 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 list total sales of all computer hardware items 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 allow comparison between the query trees before and after using the materialized view. To conserve space 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 = "Computer Hardware" AND
       store.state = "CA"
GROUP BY year
```

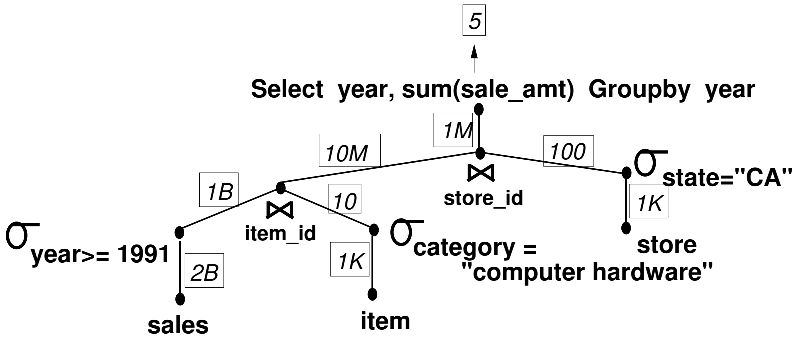

Figure 1: Query tree to compute total hardware sales for California stores by year

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"
GROUP BY sales.store_id, sales.item_id, sales.year
```

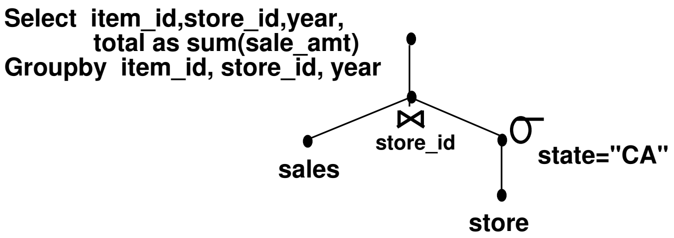

Figure 2: Query tree corresponding to yearly_sales view definition

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 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. □

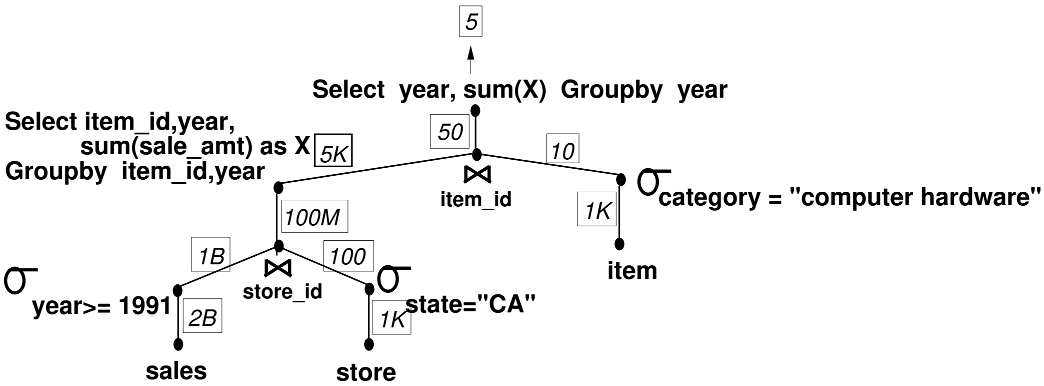

Figure 3: Query tree after reordering joins and pushing an aggregation down past the topmost join

3 GP Framework

In this section we define GPs and then state some properties of GPs 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 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.*:

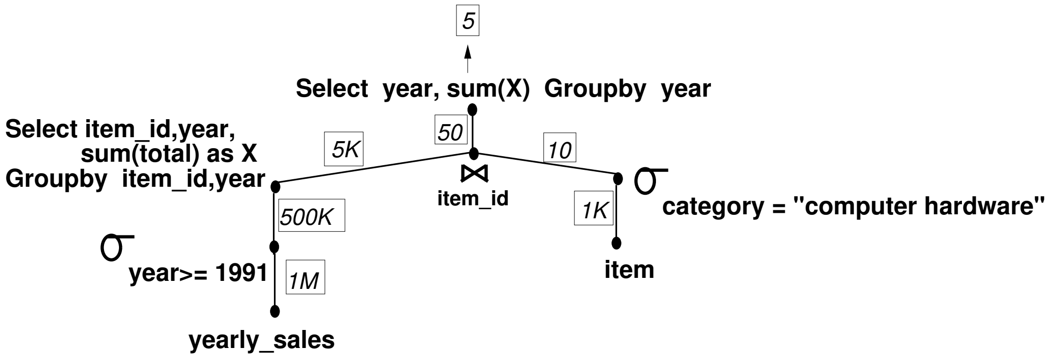

Figure 4: Query tree after integrating materialized view

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

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 GPs 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 GPs to represent these projections as well. Thus, GPs capture **distinct** projections, aggregate computations, and duplicate-preserving projections.

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

- *duplicate insensitive*: The generation or removal of duplicate tuples in their input does not affect the result of such GPs. The conventional **distinct** projection and SQL aggregations like **max** and **min** are of this type.

- *duplicate sensitive*: Duplicates must be preserved in their input. SQL aggregations **sum** and **count** are of this type.

3.2 Properties

The following properties of GPs 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 GPs may have non-SQL aggregate components requiring multiple SQL queries to express them (refer to Example 4.2).

- 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 drop the 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. We do not consider duplicate-preserving projection nodes in the query tree since they can be discarded when they appear as interior nodes and can be expressed using GPs when they occur as the topmost node (section 4.3.2). 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 do not consider the SQL **avg** aggregation in this paper, since it can be expressed in terms of the **sum** and **count** aggregations.

4 GP Transformations

4.1 Overview

In this section we consider three important query transformations

- pushing GPs down query trees.

- pulling GPs up query trees.

- coalescing two GPs 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. In the interest of clarity, since we use each of these transformations differently, we give the rules for the transformations rather than the underlying query equivalences.

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

First we build on the well-known transformation rules for the **distinct** projection to get those for all duplicate-insensitive GPs. To do so, it's important to see how duplicate-insensitive aggregation computation extends the notion of duplicate elimination. While duplicate elimination is a form of aggregation, no computation is done in the aggregation step. The aggregation step and computations done in the step are relation-based, since sets of tuples need to be considered together to derive the result, *e.g.*, the computation of **max**. We need to extend the **distinct** transformation rules since it is not possible to move such relation-based computation into tuple-based operators like selections. The following example illustrates this inability:

EXAMPLE 4.1 Consider the relational expression $\sigma_{X\geq 10}(\pi_{A,X=C*D})$. The expression can be rewritten as $\pi_{A,X=C*D}(\sigma_{C*D\geq 10})$ by moving the computation of X into the selection and doing the selection before the GP. If however we had $\sigma_{X \geq 10}(\pi_{A,X=\mathbf{min}(C)})$, then we could not move the computation of X into the selection, since selection is a tuple-based operator and **min** is a relation-based computation. That is, $\sigma_{\mathbf{min}(C) \geq 10}$ is not legal. □

The above example illustrates that computations in the selection predicate and the **groupby** components are tuple-based whereas aggregate computations are relation-based. To obtain transformation rules for duplicate-insensitive GPs we only need to extend the rules for transforming queries with **distinct** projections to handle this potential mismatch in computation types. In particular, for the push-down and pull-up transformations we extend the corresponding **distinct** transformations for the case where a selection predicate involves aggregation attributes (because only aggregate computations are relation-based). 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 GPs. Such GPs behave differently from duplicate-insensitive GPs only when duplicates are involved. Selection nodes do not affect duplicates in any way. Cross product nodes have a multiplicative effect, which results in tuples on either branch being repeated many times, above the cross product. Thus, the transformations for duplicate-insensitive GPs need be extended only for the cross-product node case to handle duplicate-sensitive GPs.

In this paper, we also show a close relationship between duplicate-insensitive GPs and selection predicates which involve the arithmetic comparisons $>, \geq, <, \leq$. We use such comparisons to generate or remove aggregations like **max**, **min** in certain cases. Our transformation rules for duplicate-insensitive GPs contains these rules as well.

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

In this section we discuss the interaction of GPs with selection conditions that use arithmetic comparisons, by discussing what happens if GPs 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 GPs down. This is true since in general every distinct value of the attribute may contribute to the result. 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, the 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 4.2 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 $\top, \bot$ when we push down GPs below selection nodes with such filters. In Example 4.2, on pushing π_A below $\sigma_{B \geq C}$ we get the new GP $\pi_{A,\top(B),\bot(C)}$. The GP $\pi_{A,\top(B),\bot(C)}(R)$ 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 $\bot$ and **min**.

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

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

where G is a set of **groupby** attributes. In general 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 GPs that have no other aggregate components. In the presence of other aggregate components the rules for evaluating $\top$ ($\bot$) are more involved. To replace a $\top$($\bot$) by a **max**(**min**) we cascade the aggregation computations such that $\top$($\bot$) is the only aggregate computed in the new step. For example, we evaluate $\pi_{A,X=\mathbf{max}(B),Y=\bot(C)}$ as $\pi_{A,X,Y=\bot(C)}(\pi_{A,C,X=\mathbf{max}(B)})$. Now we can replace the $\bot$ by **min** in the outer GP since there are no other aggregate computation, to get $\pi_{A,X,Y=\mathbf{min}(C)}(\pi_{A,C,X=\mathbf{max}(B)})$. By cascading $\top$ ($\bot$) computations we retain correctness but we do not remove as many tuples as we potentially could. The $\top, \bot$ operators and their algebra are explained in greater detail in [HG94].

4.3 GP Push-down

4.3.1 Duplicate-Insensitive GPs

In the ensuing discussion, we consider pushing GPs down query trees and examine the interaction of GPs with the different types of nodes in the query tree.

Selection Nodes: As mentioned in Section 4.1, duplicate-insensitive GPs 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 aggregation (duplicate elimination) computations 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:

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

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 GPs where possible by dropping the redundant component. Thus if an attribute occurs both as a **groupby** and **max** or **min** component, we can drop the latter. Similarly, we drop a **max**(**min**) component, if there is a $\top(\bot)$ component involving the same attribute.

EXAMPLE 4.3 Consider now 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 retaining all distinct values of B makes the $\mathbf{max}(B)$ computation possible. □

In this paper, in the interest of clarity, we do not consider aggregation computations on multiple attributes, for example $\mathbf{sum}(B*C)$. Redundant components are dropped similarly in such cases too and the details are in [HG94].

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 even allows us to create aggregation computation in queries that had none to start with.

Example 4.3 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 4.4 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 4.4 leads to the following commutativity relationship:

$$\pi_{G,X=\mathbf{max}(B)}(\sigma_{B\geq C}) = \sigma_{X\geq C}(\pi_{G,X=\mathbf{max}(B)}) \tag{2}$$

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 $\leq, <$. Table 3 in the Appendix is derived using this equivalence.

Cross-Product Nodes: For duplicate-insensitive GPs 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 4.5 Consider pushing $\pi_{A,B,X=\mathbf{max}(C),Y=\mathbf{max}(D)}$ 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. □

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.* ⊙

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

An interesting property of cross products that can be shown, is that if a relation R is the result of a cross product, $\pi_{A,X=\top(B),Y=\top(C)}(R)$ is identical to $\pi_{A,X=\mathbf{max}(B),Y=\mathbf{max}(C)}(R)$ when B, C come from different branches of the cross product. Recall, this equality is not true for a general relation R.

GP nodes: When a GP we are pushing down encounters another GP, we attempt to coalesce the two GPs into one. The rules for coalescing are given in Section 4.5. If we can indeed coalesce the two GPs into one, we continue pushing the new GP created down the query tree. If we cannot coalesce the two GPs, we cannot push the original GP down any further. However we can begin pushing the GP encountered down the query tree.

The interaction of GPs with conventional projections that preserve duplicates during push down is syntactic (renaming of attributes) just as with pushing **distinct** projections below duplicate-preserving projections.

The rules given above are summarized in Table 2 in the Appendix.

4.3.2 Duplicate-Sensitive GPs

The output of a generalized projection is always a set, *i.e.*, there are no duplicates. It is still possible to write a conventional projection that outputs duplicates using a GP. 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}. There is only a syntactic difference in the output of these two projections, even though the output of the latter is a set and the output of the former may have duplicates. 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_X 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 GPs to accommodate duplicates. GPs only produce sets as outputs (no duplicates). Duplicate semantics are simulated using **count** and the expand operator e.

Properties Of The Expand Operator The expand operator helps us understand aggregations that are sensitive to duplicates in their input. We are usually interested in pulling e up a query tree and 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 (Section 6). 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 GPs follow the same basic algorithm as duplicate-insensitive GPs for selections. Push-down rule 1 thus applies to all GPs. However, as illustrated below, arithmetic comparisons cannot be used to create the $\top$ and $\perp$ functions with duplicate-sensitive GPs. So we cannot use push-down rule 2.

EXAMPLE 4.6 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 $\perp(D)$ 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. □

Cross-Product Nodes: When we push a duplicate-sensitive aggregation computation 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 4.7 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 GPs, 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. □

EXAMPLE 4.8 Consider now pushing GP $\pi_{A,A',X=\mathbf{sum}(B),Y=\mathbf{sum}(C)}$. Let attributes A, B go down the left branch and A', C down the right branch. The computation of $\pi_{A,A',X=\mathbf{sum}(B)}$ requires us to compute **count** on the right branch and that of $\pi_{A,A',X=\mathbf{sum}(C)}$ requires us to compute **count** on the left branch. We fold together the computation of the **count** and the partial sum on each branch to get: $\pi_{A,L_S=\mathbf{sum}(B),L_C=\mathbf{count}(*)}$ on the left branch and $\pi_{A',R_S=\mathbf{sum}(C),R_C=\mathbf{count}(*)}$ on the right. The GP above the cross product is now $\pi_{A,A',X=L_S*R_C,Y=R_S*L_C}$. □

Table 1 gives the details of pushing a duplicate-sensitive GP below a cross product node. The table assumes that the attribute Y belongs to the left branch.

Component in GP	Left Branch	Right Branch	Change to GP
$X = \mathbf{sum}(Y)$	$L_s = \mathbf{sum}(Y)$	$R_c = \mathbf{count}$	$X = L_s * R_c$
$X = \mathbf{count}$	$L_c = \mathbf{count}$	$R_c = \mathbf{count}$	$X = L_c * R$

Table 1: Pushing duplicate-sensitive GPs through cross products

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

4.4 Pulling Up GPs

We are often interested in pulling GPs 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 selection attributes 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.* ⊙

Table 3, in the Appendix, is derived using this rule. Note that pulling up a GP above a selection is the same as pushing the selection down below 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.* ⊙

Pull-up rules 2 and 3 extend significantly the result in [Day87].

We can derive the pull-up rule for duplicate-sensitive GPs from the push-down rule for duplicate-sensitive GPs. We state the rule without the 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 GPs but is less powerful than pull-up rule 3 for duplicate-insensitive GPs.

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

EXAMPLE 4.9 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 $\mathtt{e}$. We can replace $R(C,D)$ with $\mathtt{e}_X(\pi_{C,D,X=\mathbf{count}(*)}(R(C,D)))$. We can now pull $\mathtt{e}_X$ up above the cross product as mentioned in section 4.3.2. After we pull up $\mathtt{e}$ 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 GPs on each branch of the cross product, we pull them up one at a time, suing pull-up rule 4.

4.5 Coalescing GPs

In this section we give rules for combining two GPs into one GP (coalescing) and so the reverse, namely rules for splitting a GP into two GPs (splitting). Coalescing and splitting are valuable tools in rewriting query trees with aggregations. Table 4 in the Appendix is derived using these rules. Coalescing has not been considered by authors before.

When we attempt to coalesce two GPs, 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 GPs 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.* ⊙

Now what happens if an attribute required by the upper GP is created by an aggregate computation at the lower GP? There are two cases to consider here: the attribute is a **groupby** attribute of the upper GP or the attribute 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 GPs 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.* $\odot$

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. The corresponding rule is given in Table 4 in the Appendix.

If we can move all the aggregation computation into the upper GP, using applying coalescing rule 3, we can drop the lower GP. Otherwise we cannot coalesce the two GPs 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

As motivated earlier, using materialized aggregate views to help answer aggregate queries is an important problem in decision support applications because using view can reduce query processing time potentially by several orders of magnitude (illustrated in Section 2).

We 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, the algorithm determines whether Q can be answered using V and if so, returns the 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 algorithm could be incorporated into a conventional query optimizer using the approach given in [CKPS95].

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 much 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 described in Section 5.2. Section 5.3 explains how the query tree of our motivating example (see Section 2) is transformed by the algorithm. Enhancements to the algorithm are described in Section 5.4.

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 sequence 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 given in Section 4. In particular, all **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* using the transformation rules described earlier.

We also require that view V and query Q use the same set of relations $R_1, \ldots, R_m$ and join them 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.

5.2 Algorithm

The algorithm for transforming an aggregate query tree over base relations into one that uses a materialized aggregate view appears in Figure 5.2. In this section we give the intuition behind the algorithm and explain each step.

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 that since Q can be a subtree of a larger query tree, the algorithm can be applied 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 transformations are made according to the rewrite rules presented in Section 4. First, selection conditions are pushed down past the *GP* operators. Then, the *GP* operator in the query is split so that the bottom *GP* is similar to the *GP* operator in the view. Finally, extra conditions below the *GP* in the query that can apply on top of the view are pushed above the *GP*. The algorithm checks the following conditions in order to derive Q'.

1. The selection conditions in V are no more restrictive than the selection conditions in Q.
2. The groupby components of $GP(Q)$ are a subset of the groupby components of $GP(V)$.[2]
3. The aggregate components of $GP(Q)$ are derivable from the aggregate components of $GP(V)$.

[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] We show later that this condition can be relaxed.

Algorithm 5.1
Input
Query tree Q for query, query tree V for materialized view. Both query trees are expected to be in the normal form described in Section 5.1: $\sigma_h \pi \sigma_l \mathcal{X}$
Output
Modified query tree Q' such that $Q'(V) = Q$, OR
FAIL, *i.e.*, the algorithm does not produce a Q'.
Method
% Step 1. Push down selection conditions.
Push selection conditions $\sigma_h(V)$ and $\sigma_h(Q)$ according to Table 3 past $GP(V)$ and $GP(Q)$ respectively to add new conditions to $\sigma_l(V)$ and $\sigma_l(Q)$.

% Step 2. Check that V is not more restrictive than Q before the aggregation.
If $\sigma_l(Q) \not\Rightarrow \sigma_l(V)$ then FAIL and exit algorithm *% $\sigma_l(V)$ is more restrictive than $\sigma_l(Q)$.*

% Step 3(A). Check whether $GP(Q)$ and $GP(V)$ have the same groupby components.
If the groupby components of $GP(Q) \not\subseteq$ the groupby components of $GP(V)$
FAIL and exit algorithm
Let $\bar{A}$ be those groupby components of $GP(V)$ that are not groupby components of $GP(Q)$.
If $\bar{A} \neq \emptyset$ *% $GP(V)$ groups by attributes that are not grouping attributes of $GP(Q)$.*
If $\sigma_h(V) = \emptyset$ then split $GP(Q)$ according to Table 4 into $GP_{bot}(Q)$ and $GP_{top}(Q)$.
Else FAIL and exit algorithm *% $\sigma_h(V)$ is not empty*
% If $GP(Q)$ is not split, i.e. $\bar{A}$ is empty, then $GP_{bot}(Q)$ and $GP_{top}(Q)$ are the same as $GP(Q)$.

% Step 3(B). Check that every aggregate component of $GP_{bot}(Q)$ is computable from $GP(V)$.
For every aggregate component F of $GP_{bot}(Q)$
If F is not computable from $GP(V)$ then FAIL and exit algorithm

% Step 4. Identify the extra selection conditions in Q and ensure they are applicable on V
For each conjunct $C \in \sigma_l(Q)$ such that $\sigma_l(V) \not\Rightarrow C$
if C can be pulled up using the converse of Table 3 then Pull C above $GP_{bot}(Q)$.
Else if C is implied by some selection conditions above $GP_{bot}(Q)$ in Q
remove C from $GP_{bot}(Q)$.
Else FAIL and exit algorithm

% Step 5. Check that V is not more restrictive than Q after the aggregation.
If $\sigma_h(Q) \not\Rightarrow \sigma_h(V)$ then FAIL and exit algorithm *% $\sigma_h(V)$ is more restrictive than $\sigma_h(Q)$.*

% SUCCESS
Replace the subtree rooted at $GP_{bot}(Q)$ in Q with view V.
Return the resulting query tree as Q'. $\diamond$

Figure 5: Algorithm to compute query Q' that answers query Q using materialized view V

4. If the selection conditions in Q are more restrictive than those in V, then these extra conditions must be applicable to the groupby or aggregate components of $GP(V)$.

5.2.1 Algorithm Steps

In Step 1 of the algorithm selection conditions in $\sigma_h(Q)$ that can be pushed past $GP(Q)$ according to the rules of Table 3 are pushed down to $\sigma_l(Q)$. In general, selection conditions that involve only constants and the groupby components of a *GP* can be pushed past the *GP*, but Table 3 also contains rules for pushing down certain conditions on the result of aggregate components. After pushing down selection conditions, the only conditions remaining in $\sigma_h(Q)$ involve the result of aggregate components. Similarly, selection conditions are pushed down the view tree V. Selection conditions are pushed down in preparation for the implication test in Step 2 below.

In Step 2 we check that the selection conditions in the resulting $\sigma_l(V)$ are no more restrictive than the selection conditions in the resulting $\sigma_l(Q)$. If $\sigma_l(V)$ is more restrictive than $\sigma_l(Q)$, then tuples that could appear in the groups formed by $GP(Q)$ would be filtered out by the conditions of $\sigma_l(V)$; hence the algorithm determines it is not possible to derive Q'.

In Step 3(a) we check whether the groupby components of $GP(Q)$ are the same as those in $GP(V)$. 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'.

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. In this case it is possible to derive the aggregate components in $GP(Q)$ from the aggregate components in $GP(V)$. 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 $GP(V)$, with its aggregate components derived from the aggregate components of the original $GP(Q)$ using the GP-splitting rules in Table 4.

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 $GP(Q)$.

In Step 3(b) we check whether the aggregate components of $GP_{bot}(Q)$ are derivable from the aggregate components of $GP(V)$. An aggregate component C of $GP_{bot}(Q)$ is derivable from $GP(V)$ if C is identical to some component in $GP(V)$ or if C is computable from one of more components of $GP(V)$. For instance, `AVG(a)` can be derived from `SUM(a)` and `COUNT(*)`. If an aggregate component of $GP_{bot}(Q)$ is not derivable from $GP(V)$ then the algorithm determines it is not possible to derive Q'.

Step 4 tries to make the conditions below $GP_{bot}(Q)$ less restrictive than V so that $GP_{bot}(Q)$ and below may be replaced by V without losing any information. Thus, selection conditions in $\sigma_l(Q)$ that are not implied by selection conditions in $\sigma_l(V)$ are pulled up past $GP_{bot}(Q)$ according to the converse of the rules given in Table 3. 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 $\sigma_l(Q)$; hence the algorithm determines it is not possible to derive Q'.

In Step 5 we check that the selection conditions in $\sigma_h(V)$ are no more restrictive than the selection conditions in $\sigma_h(Q)$. If $\sigma_h(V)$ is more restrictive than $\sigma_h(Q)$, 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 algorithm determines that it is possible to derive Q'. At this point the view tree V is equivalent to the subtree of the transformed query tree Q rooted at $GP_{bot}(Q)$.[3] The algorithm returns as Q' the transformed query tree Q with the subtree rooted at $GP_{bot}(Q)$ replaced by the materialized view.

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

5.3 Example

We explain 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 computer hardware items in all California stores for each year beginning with 1991. A `yearly_sales` view is materialized computing total yearly sales by item and store for stores in the state of California. The initial query tree for the view appears on the left-hand side of Figure 6 using our *GP* notation. We want to determine if the view `yearly_sales` can be used to answer the query.

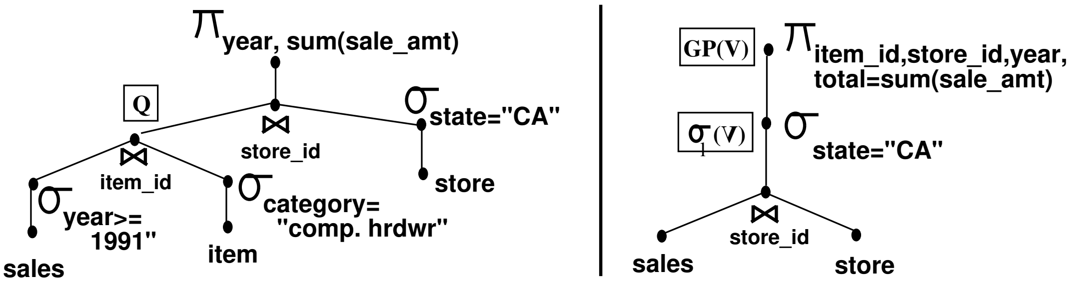

Figure 6: Initial query and normalized view `yearly_sales`

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 first joined with `store` followed by `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 described in Section 4.4. In our example the single *GP* of the example query is already above all joins. Next we use the *GP* push-down 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. Our algorithm considers whether Q can be answered using the tree V corresponding to the `yearly_sales` view definition.

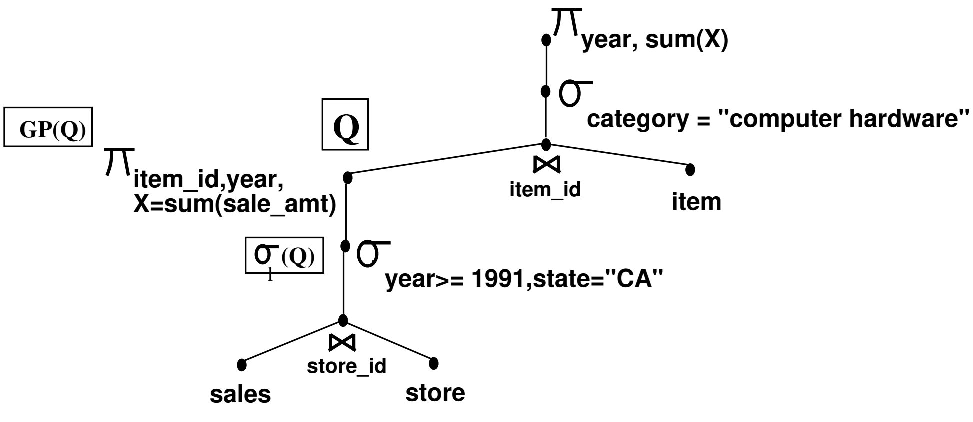

Figure 7: Query tree after subtree Q has been normalized

Before applying our algorithm to Q and V we put them in the normal form described in Section 5.1. V is normalized by pulling up the selection condition on `state` above the join (shown on the left-hand side of 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.

Step 1 of the algorithm pushes selection conditions from σ_h to σ_l for both the query and the view. Since $\sigma_h(Q)$ and $\sigma_h(V)$ are both empty we skip this step.

Step 2 checks that $\sigma_l(Q) \Rightarrow \sigma_l(V)$. Since all selection conditions in $\sigma_l(V)$ are also in $\sigma_l(Q)$ this test succeeds.

Step 3(a) splits $GP(Q)$ if the groupby components of $GP(Q)$ are a proper subset of the groupby components of $GP(V)$. The groupby components of $GP(V)$ include `store_id` in addition to `item_id` and `year`. Thus, $GP(Q)$ is split into $GP_{top}(Q)$ and $GP_{bot}(Q)$, as shown on the left-hand side of Figure 8.

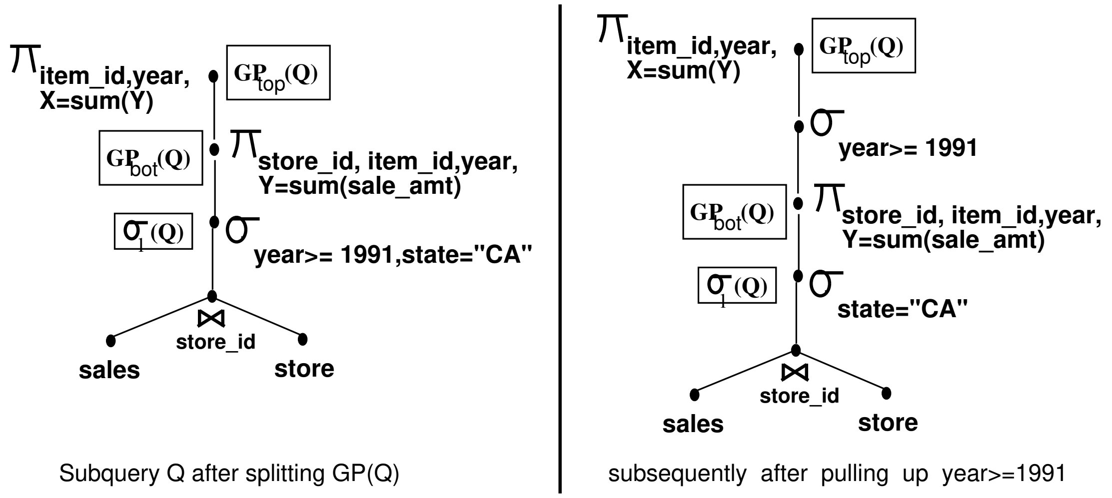

Figure 8: Subqtree Q after splitting $GP(Q)$ and then pulling up year $\geq$ 1991

Step 3(b) checks whether all aggregate components of $GP_{bot}(Q)$ are derivable from the aggregate components of $GP(V)$. Since $Y = \mathbf{sum}(sale\_amt)$ of $GP_{bot}(Q)$ can be obtained from $total = \mathbf{sum}(sale\_amt)$ of $GP(V)$, this test succeeds.

Step 4 identifies conjuncts in $\sigma_l(Q)$ that are not implied by selection conditions in $\sigma_l(V)$ and need to be pulled up past $GP_{bot}(Q)$. Note 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 on the right-hand side of Figure 8.

Step 5 checks that $\sigma_h(Q) \Rightarrow \sigma_h(V)$. Since $\sigma_h(Q)$ and $\sigma_h(V)$ are both empty this is true.

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`. The left-hand side of Figure 9 shows the original query tree with subtree Q replaced by 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 on the right-hand side of Figure 9. □

5.4 Enhancements

For the sake of clarity we omitted several enhancements that we have made to the basic algorithm presented in Section 5.2. The enhancements extend the algorithm to cover additional cases when Q' can be derived. Even with the enhancements the algorithm is not complete due to the undecidability of the implication problem for the class of aggregations we consider [RSSS94]. However, the algorithm handles a very large class of queries and views that include the common cases.

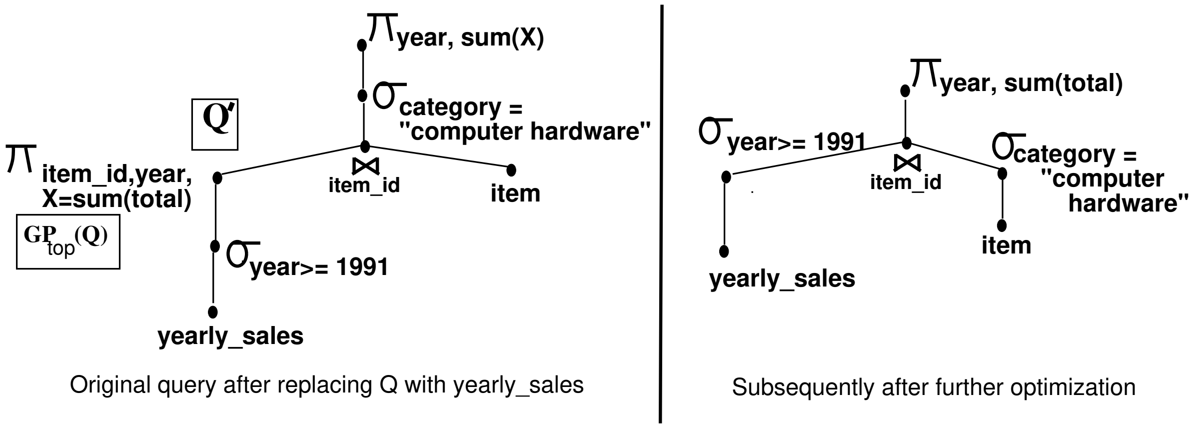

Figure 9: Query tree after replacing subtree Q with Q' and then after further optimization

- Remove *useless* aggregate components from $GP(Q)$, *i.e.*, aggregate components whose result syntactically is not used by any other operator. This step is performed just before Step 1.
- Add grouping components to $GP(Q)$. Attributes in $\sigma_l(Q)$ that are either equated to a constant or to some grouping component of $GP(Q)$, can be added as grouping components to $GP(Q)$. Thus, we can avoid splitting $GP(Q)$ if all grouping components of $GP(V)$ that are not grouping components of $GP(Q)$ can added to $GP(Q)$. This step is performed just before Step 3.
- Remove grouping components from $GP(Q)$ in order to handle the case when $GP(Q)$ has more grouping components than $GP(V)$. Grouping components in $GP(Q)$ that do not appear in $GP(V)$ can be removed from $GP(Q)$ if they are equated to a constant or to another grouping component by $\sigma_l(Q)$. This step is performed just before Step 3.

In addition to the above listed enhancements that we have incorporated into the algorithm, more are possible. For instance, we could also consider semantically useless aggregate components in addition to syntactically useless aggregations. Furthermore, selection conditions that are pushed below $GP(V)$ or $GP(Q)$ in Step 1 may have to be pulled up later. We could minimize the movement of selection conditions by executing earlier some tests that are currently executed later in the algorithm.

6 GPs and Query Optimization

6.1 GP Transformations

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 [HFLP89] as our example optimizer and discuss how and where our transformations can be used. The Starburst optimizer works in two phases: query rewrite and plan optimization. The plan optimizer, works on each `SELECT` block as a unit and finds the cheapest plan in a cost based manner. Plan optimization is not done across blocks and so the central philosophy of the query rewrite phase is to attempt to merge `SELECT` blocks [PHH92] and thus increase the alternative plan available to the plan optimizer. In particular, by moving all the joins into one block, the plan optimizer can consider all possible join orders and methods. Merging `SELECT` blocks can provide order-of-magnitude improvements in query evaluation performance [PHH92]. Merging `SELECT` blocks essentially involves pulling up projections (SELECT) in query trees (QGMs in Starburst), handling duplicates correctly. In the presence of aggregations the Starburst optimizer cannot merge blocks.

The GP pull-up technique we've presented in this paper enables us to merge blocks even when they involve aggregation. It can thus be used in the query rewrite phase of Starburst. In particular we can rewrite query trees considered in Section 3.3 in a normal form such that all the joins in the tree are in one block. Then the plan optimizer can consider all possible join orders and methods yielding significant performance benefits, just as with SELECT block merge. Importantly, since we treat aggregations as generalized projections, it is easy to change the existing SELMERGE rule in [PHH92], to derive the corresponding rule for aggregations. We do not provide the details in this paper, since our focus here is on transformations. The coalescing rules given in this paper can also be used in the query rewrite phase to reduce computation by merging aggregates.

The push-down transformation can be used by the plan optimizer in a cost based manner to optimize plans after the join ordering has been determined. Using push-down, Aggregations can be placed in the tree in positions that are exponential in the number of nodes in the tree. It is infeasible for a plan optimizer to consider each of these alternatives while picking the cheapest. So a heuristic based approach is needed in determining the placement of aggregations. An example of such a heuristic is given in [CS94]. The detailed algorithm we give in [HG94] enables the plan optimizer to pick any set of placements for the aggregations using heuristic it desires.

6.2 The Expand Operator

When a relation $R(A, B)$ (base or intermediate) has many duplicate tuples, we can compute the number of duplicates for each tuple value in a new attribute X, by doing $\pi_{A,B,X=\mathbf{count}(*)}$. The original relation R is obtained by applying the expand operator $\mathtt{e}_X$ to the result. If the expand operator is pulled up the tree then the intermediate relations have fewer tuples (albeit an extra attribute for the count). This technique of computing duplicate values early in query processing is analogous to eliminating duplicates early and may be useful when there are many duplicate tuples. Doing such duplicate computation also helps with pre-existing sort nodes in query plans. By computing a count of duplicate tuple values during sorting, using a technique called "early aggregation" [G93], we reduce the I/O done when many duplicates are present.

7 Related Work

Answering queries using materialized views has received much attention in the past [LY85, YL87, LMSS94, CKPS95]. However, existing work has focussed on SPJ views and queries that do not use aggregation. Aggregates are an important extension to the previously considered queries because aggregations are at the heart of decision support and warehousing. Most data warehousing architectures have aggregate materialized views called summary tables, and the ability to use such views instead of the massive base tables promises significant performance gains.

The query transformations we give unify and properly subsume the push-down transformations given in [CS94] and the pull-up transformations given in [Day87]. 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 GPs different from duplicate-sensitive GPs, we can infer more powerful transformation rules for duplicate-insensitive GPs. For example, we can pull up two duplicate-insensitive GPs 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] also provides a heuristic to push aggregations down in a cost-based manner. Another related work is [YL94] which uses functional dependencies to move aggregations past joins.

8 Conclusions and Future Work

In this paper we present a new framework for reasoning with **groupby** and aggregation in **select-from-where-groupby-having** SQL queries. We generalize **distinct** projections to yield the notion of "generalized projections" that capture **groupby** and aggregation computations. GPs also capture arithmetic comparison operators that are not expressible as usual 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.

We derive transformation rules for aggregate queries 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.

We give an algorithm for a hitherto unsolved problem, namely, how to answer an aggregate query using one or more materialized aggregate views instead of some of the base relations. This algorithm is very useful in applications like decision support and data visualization. The algorithm is developed using the new transformation rules that we obtain using the GP framework.

8.1 Future Work

We plan to focus our attention on important problems where solutions are known for SPJ queries and extend them to queries involving aggregation. This extension is especially important given the increasing popularity of applications like decision support that make heavy use of aggregations. As a first step we plan to extend the results we have on query containment for queries with aggregation.

We also plan to investigate the following:

- Examples show that the rules to move GPs allow us to extend the predicate move-around techniques for query optimization [LMS94].
- In this paper we discuss how the transformation rules may be incorporated into a conventional query optimizer. We plan to conduct experimental studies to determine the performance benefits of the incorporation.
- Use views to answer queries that belong to an even more general subclass of SQL with aggregation. Similarly, for containment.

Acknowledgements

We thank Prof. Jeff Ullman and Prof. Jennifer Widom for valuable feedback on the ideas presented in this paper. We also thank Anand Rajaraman, Surajit Chaudhuri, and Waqar Hasan for discussing with us various issues in the paper.

References

[CS94] S. Chaudhuri and Kyuseok Shim. Including Group-By in Query Optimization. In *Proceedings of the Twentieth International Conference on Very Large Databases (VLDB)*, pages 354–366, Santiago, Chile, 1994.

[CKPS95] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, Kyuseok Shim. Query Optimization in the presence of Materialized Views. To appear in *Proc. of 11th IEEE International Conference on Data Engineering (ICDE-11)*, Taipei, 1995.

[Day87] U. Dayal. Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers. In *Proceedings of the Thirteenth International Conference on Very Large Databases (VLDB)*, pages 197–28, Brighton, England, 1987.

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

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

[HFLP89] Laura M. Haas, J. C. Freytag, Guy M. Lohman, and Hamid Pirahesh. Extensible query processing in starburst. In *Proceedings of ACM SIGMOD 1989 International Conference on Management of Data*, pages 377–388, Portland, OR, May 1989.

[HG95] Venky Harinarayan and Ashish Gupta. Optimization Using Tuple Subsumption. In *ICDT 95*, Prague, January 1995.

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

[LMS94] Alon Levy and Inderpal Singh Mumick and Yehoshua Sagiv. Query Optimization by Predicate Movearound. In *VLDB 1994*, pp: 96-107.

[LMSS94] Alon Y. Levy, Alberto O. Mendelzon, Yehoshua Sagiv, and Divesh Srivastava. Answering queries using views. To appear in *PODS*, 1995.

[PHH92] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *SIGMOD*, pages 39-48, 1992.

[LY85] P. A. Larson and H.Z. Yang. Computing queries from derived relations. In *Proceedings of the 11th International VLDB Conference*, pages 259–269, 1985.

[RSU95] Anand Rajaraman, Yehoshua Sagiv, and Jeffrey Ullman. Answering queries using templates with binding patterns. To appear in *PODS*, 1995.

[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.

[Ull89] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, New York, 1989.

[YL87] H. Z. Yang and P. A. Larson. Query transformation for PSJ-queries. In *Proceedings of the 13th International VLDB Conference*, pages 245–254, 1987.

[YL94] W. P. Yan and P. A. Larson. Performing Group-By Before Join. In *International Conference on Data Engineering*, 1994.

A Pushing duplicate-insensitive GPs down query trees

N is the node we're pushing the GP P past. Q is the new GP created (in the case of cross products, we have two new GPs, Q_{left} and Q_{right}. If the new GP Q is exactly the same as the old GP P, then we eliminate P from the tree (independent of the new form of P).

Node	Contents of N	New GP Q	Effect on N	Effect on P
Select	$X = Y$	add X, Y to P as gby cmps	No effect	rewrite agg. attribute to use new att. created
	$X = c$	add X to P as gby comp	No effect	same as above
	$X \geq Y$, $X > Y$	add $X' = \top(X)$, $Y' = \bot(Y)$ as Agg cmps	Replace X, Y by X', Y'	Replace each agg cmp $A = \top(B)$ by $A = \top(B')$. Add $B' = \top(B)$ to Q
	$X \geq c$ $X \leq c$	add $X' = \top(X)$ $X' = \bot(X)$	replace X by X' replace X by X'	same as above same as above
Cross		Q_{left} = left branch att. of P Q_{right} = right branch att. of P		Drop P

Table 2: Pushing duplicate insensitive GPs

B Rules for pushing selection conditions past *GP*s and splitting *GP*s

B.1 Rules for Pushing Selection Conditions Past *GP*

We discuss how to push selection conditions from a conjunction of selection conditions "sigmaBLO", past a generalized projection "GP," to the selection conditions "sigmaAB" below the GP. sigmaAB and sigmaBLO are conjunctive conditions where each condition is referred to as "C." The form of C is discussed in the following table.

We push conditions from sigmaAB past the GP and add them to sigmaBLO (and if possible, remove predicates from sigmaAB). The following table summarizes the rules for this pushdown: The rules for pulling up selection conditions are the converse of the rules for pushing down selection conditions.

B.2 How to split a GP into two GPs

Consider a GP $GP1$ that has groupby attributes $A(GP1)$ and a GP $GP2$ that has groupby attributes $A(GP2)$, and let $A(GP1) \subseteq A(GP2)$. We give rules to split $GP1$ into two GPs, GP_{bot} and GP_{top} such that GP_{bot} has groupby attributes $A(GP2)$ and GP_{top} has groupby attributes $A(GP1)$.

Form of "C"	Additional restrictions		Add to sigmaBLO	Remove from sigmaAB
X OP a	X is a groupby attribute of GP OP is any operator		C	C
X OP Y	X and Y are groupby attributes of GP		C	C
X OP a	X is the only aggregate defined in GP			
	X's defn in GP	OP		
	$X = \mathbf{max}(B)$	$=$	$B \geq a$	
	$X = \mathbf{min}(B)$	$=$	$B \leq a$	
	$X = \mathbf{max}(B)$	$>$	$B > a$	$X > a$
	$X = \mathbf{max}(B)$	$\geq$	$B \geq a$	$X \geq a$
	$X = \mathbf{min}(B)$	$<$	$B < a$	$X < a$
	$X = \mathbf{min}(B)$	$\leq$	$B \leq a$	$X \leq a$

Table 3: Rules to push selection condition "C" down, past GP

GP_{bot} and GP_{top} are obtained according to Table 4. Each aggregate computation in $GP1$ is broken up as follows to produce the aggregate computations that go into GP_{bot} and GP_{top}.

Aggregation in $GP(Q)$	Aggregation in GP_{bot}	Aggregation(s) in GP_{top}
$X = \mathbf{max}(D)$	$X' = \mathbf{max}(D)$	$X = \mathbf{max}(X')$
$X = \mathbf{min}(D)$	$X' = \mathbf{min}(D)$	$X = \mathbf{min}(X')$
$X = \mathbf{sum}(D)$	$X' = \mathbf{sum}(D)$	$X = \mathbf{sum}(X')$
$X = \mathbf{count}(*)$	$X' = \mathbf{count}(*)$	$X = \mathbf{sum}(X')$
$X = \mathbf{avg}(D)$	$X' = \mathbf{sum}(D)$, $Y' = \mathbf{count}(*)$	$X = \mathbf{sum}(X')/\mathbf{sum}(Y')$

Table 4: Rules to split a GP into two cascaded GPs