Generalized Projections: a Powerful Query-Optimization Technique [*]

Venky Harinarayan **Ashish Gupta**[†]

Abstract

In this paper we introduce *generalized projections* (GP). GPs capture aggregations, groupbys, conventional projection with duplicate elimination (**distinct**), and duplicate preserving projections. We develop a technique for pushing GPs down query trees of Select-project-join queries that may use aggregations like **max**, **sum**, *etc.* and that use arbitrary functions in their selection conditions. Our technique pushes down to the lowest levels of a query tree aggregation computation, duplicate elimination, and function computation. The technique also *creates* aggregations in queries that did not use aggregation to begin with. Our technique is important since applying aggregations early in query processing can provide significant performance improvements. In addition to their value in query optimization, generalized projections unify set and duplicate semantics, and help better understand aggregations.

1 Introduction

A fundamental problem in query processing is deciding when to drop rows (tuples) and columns (attributes) of relations that do not contribute to the final result of a query. Query optimizers try to drop irrelevant rows and columns as early as possible in query processing. Removing irrelevant tuples and attributes early leads to smaller intermediate relations thus reducing the cost of the query. However there is also a cost associated with discovering and discarding irrelevant tuples and attributes. Query optimizers in commercial systems handle this trade-off in many ways. Some optimizers seek to minimize cost by using heuristics, others by exhaustively evaluating numerous alternatives in a cost based manner [S79]. Yet others use a combination of heuristics and cost based evaluation. Techniques which remove irrelevant tuples and attributes early are thus of considerable importance in query processing. A commonly used method for removing irrelevant tuples early is to push selections down the query tree [Ull89]. The further down a tree we can push a selection, the earlier in processing we discard irrelevant tuples. Thus far, pushing projections down a query tree has been used only in removing irrelevant attributes early in query processing. In this paper, we show how projections can be used to remove irrelevant tuples as well and summarize the relevant information contained in a relation.

We introduce the notion of a *generalized projection* that unifies duplicate eliminating projections (corresponds to the SQL **distinct** adjective), duplicate preserving projections, groupbys, and aggregations, in a common framework. We develop an algorithm for pushing GPs down query trees. Thus, we are able to push duplicate elimination, aggregation, and duplicate preservation in a

[*]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: {venky,agupta}@cs.stanford.edu

[†]Current address: IBM Almaden Research Center, San Jose

uniform way resulting in query execution strategies that not derivable using existing optimization techniques. Though selections and generalized projections both eliminate tuples, they do so in different ways. A selection compares each tuple of the relation with the selection predicate and discards those that do not satisfy the selection predicate. A generalized projection does not work at the level of tuples but rather at the level of a relation. The end result though is similar: both can reduce the size of relations considerably and can thus give big performance gains if applied early in query processing. As we shall see in the following sections, a generalized projection can be expressed using SQL aggregation-groupby operators. Therefore, by pushing GPs down trees, we are able to create aggregate subqueries in queries that did not originally use aggregation.

The examples below give an indication of the wide range of SQL queries to which our technique applies and illustrate both pushing down and creation of aggregations. The optimized query trees in the examples are produced using the algorithm given in this paper.

EXAMPLE 1.1 Consider the following schema that models an automobile manufacturer's database.

`cost`($\underline{M\#}$, CP) *% "\$CP" is the base cost price for model "M#"*
`factor`($\underline{M\#}$, $State$, $Factor$) *% Model "M#" has a overhead of "Factor" in "State"*
`sales`($\underline{VId}$, $M\#$, SP, $Dealer$, $State$)
% Car with ID "VId" and model "M#" was sold in "State" by "Dealer" for a sales price "\$SP"

The underlined columns state the key for each relation. On the above schema, consider a query $Q1$ that computes for each model the profit made by the car manufacturer. The following query tree represents this query. The topmost projection, not shown, outputs $M\#$ and the corresponding value for $[\mathbf{sum}(SP) - \mathbf{sum}(CP * Factor)]$.

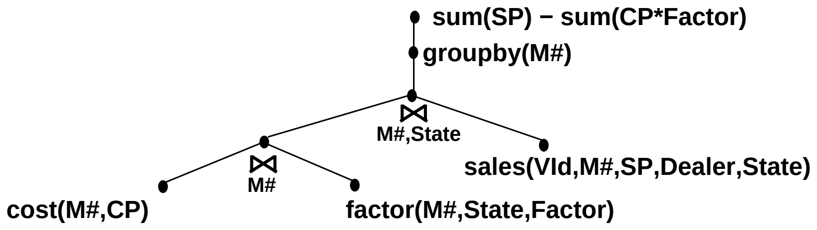

Figure 1: Find profit for company using query $Q1$

In the left branch of query $Q1$ relations `factor` and `cost` are joined to associate the multiplicative factor for each state with the base cost price of the appropriate model. The resulting intermediate relation $R(M\#, State, CP, Factor)$ is joined with relation `sales` and the total revenue is computed by summing SP. The total cost incurred by the manufacturer is computed by summing $CP*Factor$ for each car sold.

The query tree of Figure 1 does the aggregation step after all the joins are done. This is the normal way of doing aggregations in relational systems. A better option for evaluating $Q1$ is first to compute the revenues in each state for each model by aggregating relation `sales` over $(M\#, State)$. The resulting aggregate relation would be much smaller than the initial sales history table because there are far fewer states and model numbers than the total number of cars sold. The query tree for the rewritten query is in Figure 2.

Doing an early aggregation helps reduce the size of (and time taken in) subsequent joins and thus reduces the overall execution cost of the query.

Now we illustrate how aggregations can be created as a result of pushing GPs. Consider query $Q2$ executed by the market research department of the car manufacturer. $Q2$ finds all those models

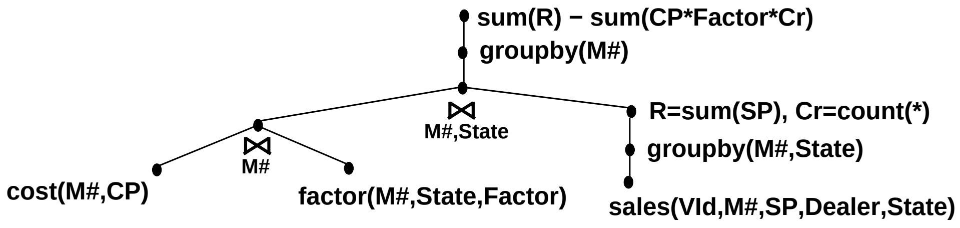

Figure 2: Query to find profit after pushing down aggregations

whose cost price after factoring in the state overhead, is more than 85% of the sales price of some car of that model sold in that state. Such models are considered low-profit. Query $Q2$ is represented by the following query tree:

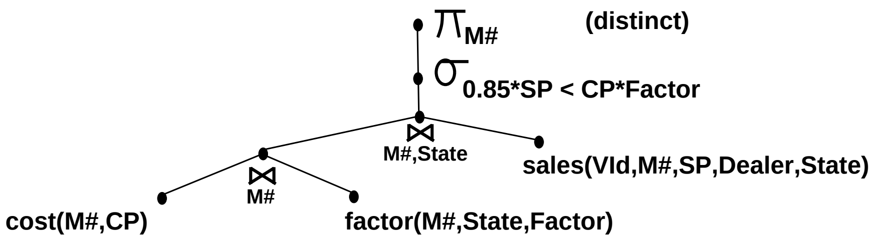

Figure 3: Find all low-profit models using query $Q2$

Query $Q2$ uses no aggregation. However, aggregation can be introduced at very low levels in the query in order to reduce the size of joins. Let us see how relation `sales` might be aggregated without affecting the answer of query $Q2$. The point to note is that all tuples of `sales` are not relevant to the query. For instance consider two tuples $t1 = (V1, miata, 14K, John, CA)$ and $t2 = (V2, miata, 13K, Sam, CA)$. Both tuples refer to cars of the same model sold in the same state. Say that $(CP * Factor)$ for a "miata" in CA is $13K$. Both tuples $t1$ and $t2$ imply that "miata" is a low-profit model. However, whenever tuple $t1$ causes "miata" to become a low-profit model, then tuple $t2$ also allows the same inference because the sales price in $t2$ is less than the sales price in $t1$. Thus, we can discard tuple $t1$ without affecting the answer to $Q2$. That is, we can introduce an aggregation operation above relation `sales` to compute for each state and each model, the minimum value of $0.85 * SP$. This minimum value determines if a model is low-profit. Figure 4 shows the query tree for query $Q2$ after the aggregation has been created. The aggregated `sales` relation has as many tuples as the product of the number of states and number of models supplied by the manufacturer. Note, a function computation has also been pushed down to a base relation.

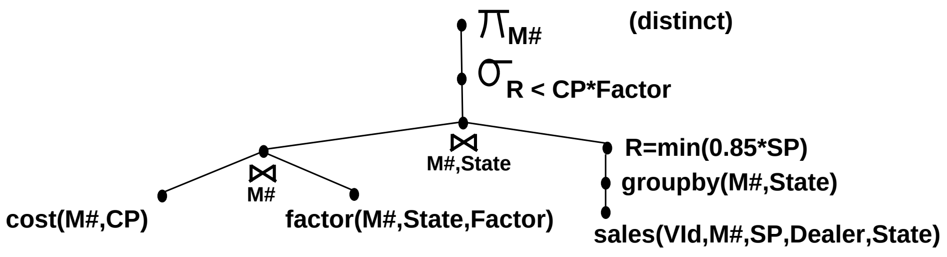

Figure 4: Query to compute low-profit cars after pushing down aggregation

$Q2$ shows that aggregations can be introduced gainfully in queries that do not use any aggregation. Queries $Q1$ and $Q2$ are representative of queries in decision support systems. These systems typically involve massive tables. The queries compute aggregate values of some attributes grouped by other attributes that denote some financial or sociological class.

This example illustrated three optimizations: pushing aggregations, introducing new aggregations in queries that did not have aggregations, and pushing function computations. All three are inferred in a uniform manner by the GP pushing algorithm we present in this paper. □

Aggregations are a way of eliminating some tuples of a relation based on some other tuples. Current optimizers do not use tuples in a relation R to discard other tuples in R. Duplicate elimination is another instance of such an optimization. Currently, **distinct** queries are computed by doing duplicate elimination at the very top of a query tree. **distinct** computations can be pushed down query trees using the algorithm we develop. We also show how to optimize standard duplicate preserving queries with no aggregations by introducing the **count** aggregate operator.

EXAMPLE 1.2 Consider query $Q3$ that for all models with base $CP > 15K$, returns the model number of the car and names of dealers who sell cars of that model. Query $Q3$ preserves duplicates. The query is represented by the left half of the following figure. This query is a simple Select-Project-Join query.

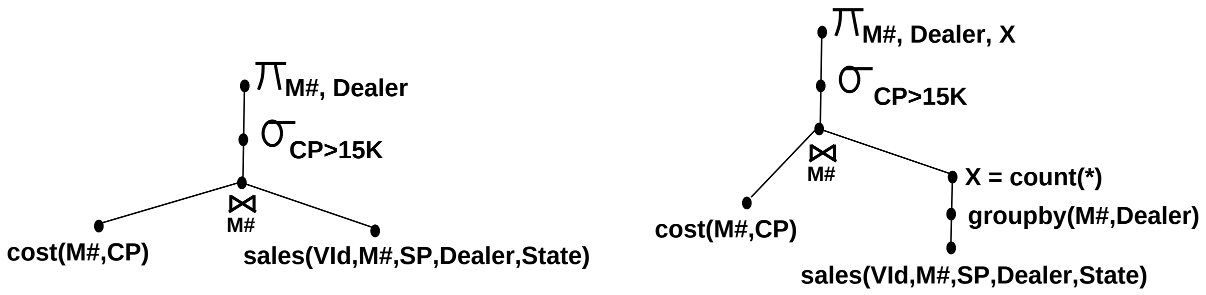

Figure 5: Compute $M\#$, $Dealer$ such that $Dealer$ sells cars of model $M\#$ and $M\#$ has $CP > 15K$

The optimized query is as shown in the right half of the above figure. In the final answer, counts represent repeated tuples. If the answer requires duplicates as output, we need to output each tuple as many times as indicated by its count. In general, if duplicates are needed in the answer then generalized projection pushing involves keeping a count of the number of duplicates at only some intermediate nodes. Our rewrite algorithm allow us to carry counts as an extra attribute for all intermediate computations. Thus, in all intermediate steps, the cost of joins and selections will be reduced because multiple tuples will be replaced by a single tuple that has an extra field. It is important to note that keeping a count does not require altering the way joins are done.

Consider a database where relation `sales` has 1 million tuples, that involves 1000 distinct dealer names, and 10 model numbers. In the optimized tree for query $Q3$, the size of the aggregated `sales` relation is at most $10,000$ assuming that each dealer sells every model. Even with this conservative assumption, the number of tuples after the join is reduced by a factor of 100. □

Aggregations are an integral part of SQL [LCW93] and are very commonly used in decision support systems. Efficient implementations of aggregation and groupby operators exist in most commercial systems. The performance impact of doing aggregations early has been recognized. For example, in the Tandem optimizer, single table aggregations are done at the disk process level whenever possible [HJT94]. While efficient single-table aggregation implementations are present, such features are not used in queries where the aggregation is done after a join. The algorithm presented in this paper allows us to push aggregations down a query tree, enabling the use of efficient single-table aggregation operators before joins are taken, as well as reducing the size of intermediate relations. Creating aggregations also makes possible significant performance gains in conventional queries executing on operational databases that do not use aggregations as often. One of the biggest benefits of our technique is that it involves no new operators: the building blocks (aggregation-groupby operators) are already present in commercial systems.

In this paper we do not investigate in detail the interaction of our GP pushing algorithm with an existing query optimizer. Query optimizer architecture varies widely and different optimizers would use the algorithm at different stages in the optimization process. In the Starburst optimizer [HFLP89], for example, the algorithm for pushing GPs could be done at query rewrite time using heuristics to determine which GPs should be pushed further down the query tree. It is also possible to use the GP pushing algorithm on query plans using a cost based scheme to decide which GPs should be used in the final phase. To avoid getting mired in details, in this paper, we push GPs down a query tree and assume the query optimizer can decide which GPs are useful to evaluate in the final tree. Note, GPs can be translated to aggregate computations, so standard optimizers have to deal with existing operators. Also, we use a query tree composed of standard relational operators to represent queries. The query tree can be mapped to an optimizers internal representation in an implementation specific manner. We do not discuss in detail issues relating to when the tree was generated in query processing or what evaluation algorithm the query optimizer uses.

We believe that generalized projections provide a new view of aggregate queries and duplicate/set projections by unifying them in a common framework. Besides providing a powerful technique for optimizing aggregate queries, this new point of view promises to yield insight into how to better understand aggregations.

Paper Outline

In the course of this paper we develop the GP pushing algorithm in stages. We start with less expressive queries and then progressively add features, enhancing the algorithm to arrive at a complete solution. We proceed as follows:

In Section 2 we describe the class of queries that we consider. In Sections 3 and 4 we give the intuition and develop the algorithm for pushing GPs down query trees for select-project-join queries that do not preserve duplicates and that have no aggregation. In Section 5 we discuss how generalized projections can be pushed down queries that do not preserve duplicates and that have **max**-**min** operators. In Section 6 we give the intuition for how GPs are pushed down queries that preserve duplicates. Then, we give the general algorithm for pushing GPs down SPJ queries that may or may not preserve duplicates and that use the aggregation operators **sum**-**avg**-**count**-**max**-**min**. In section 7 we briefly discuss how to extend the GP pushing algorithm to query trees that use union and difference.

In section 8 we discuss performance issues concerning our algorithm. We discuss the kind of queries that we think will gain from pushing generalized projection. Finally, in Section 9 we summarize the contributions of this paper and consider potential extensions of the technique introduced in this paper.

1.1 Related Work

[Day87] mentions the importance of the relative positioning of aggregate computations and joins. That paper describes algebraic operators that can represent general SQL queries and the paper also describes how to choose orderings of these operators in order to get good query plans. The paper considers the problem of discovering new join orders, and pipelining opportunities for general SQL queries. In contrast, we consider the more specific problem of pushing and creating aggregations in SQL queries that are expressible using query trees as discussed in Section 2. More recently, [YL93] consider how to push **groupby** past joins. Their work was subsequently generalized in [CS94], which discusses how to push down an aggregation operator that exists at the top of the query tree. Aggregates are not introduced in query trees that had none to begin with. Our technique also allows us to push aggregation operators past other aggregations that may be present in query trees, and thus into views defined using aggregations.

[HG95] introduces the notion of tuple subsumption. Both conventional projections and aggregation-groupby operators are tuple subsumption operators. The generalized projection we use in this paper is based on this commonality.

2 Scope of our Results

In this paper, we consider the class of queries that can be expressed by query trees of the kind used in the introduction. The permitted query trees have five types of nodes: selection nodes, projection nodes, cross-product nodes, groupby nodes, and aggregate-groupby node pairs. The topmost node is always a projection. This projection is the GP that is pushed down the query tree. Projections may preserve duplicates or discard them. That is, the projection may or may not be "**distinct**". In case an interior projection node discards duplicates, it can be replaced by a groupby node. We assume that all **distinct** projections in the query tree are replaced by **groupby**s. The query tree may also have interior duplicate-preserving projection nodes. Duplicate preserving projections in interior nodes do not affect the results of a query, when the topmost node is a projection. Their presence is purely for performance reasons, since they discard unneeded columns early and they do not interfere with our algorithm for pushing GPs in any way.

Other than duplicate-preserving projection nodes, query trees have four other types of interior nodes. Selection nodes eliminate tuples from the input relation, groupby nodes do projection+duplicate elimination, and cross-product nodes output the cross product of two input relations. For ease of exposition we do not consider join nodes explicitly since a join is a cross-product followed by a selection and can thus be expressed by cross-product and selection nodes. Aggregate-groupby node pairs have a groupby node followed by an aggregate node. An aggregate-groupby node pair produces as output a relation with one tuple for every distinct value in the input relation of the groupby attributes. The output relation has as its attributes the groupby attributes and the aggregate functions computed by the aggregate node.

The query tree we consider in this paper can represent single block SQL queries. The relations in the **from** clause can be base relations or views. The views are themselves single block SQL queries with the same structure. The query tree may thus have nested aggregates when the underlying query uses nested views.

3 Intuition for Set Semantics Queries

In this section we restrict the discussion to queries whose trees have no aggregate nodes, and where the topmost projection node eliminates duplicates. First we develop the intuition for pushing GPs down query trees for such queries and then we develop the actual algorithm for pushing GPs. In subsequent sections we extend the algorithm, first to queries that use aggregations like **max** and **min** and finally to queries that preserve duplicates and that use aggregations like **sum**. We follow this sequence because set semantics query trees are easiest to understand and form the foundation on which we build our algorithm.

Generalized Projections

The conventional projection operator is the only basic relational operator that manipulates attributes (columns) of a relation. A conventional projection takes an input relation and produces a new relation whose attributes are derived in some way from the attributes of the original relation. Aggregations in SQL are closely related to the relational projection operator. Aggregations as defined in the SQL standard [ISO92] also produce a new relation given an input relation, by manipulating attributes of the input relation.

We introduce a generalized projection operator, denoted by the symbol π, that is similar to the aggregation operator.[1] A GP takes as its argument a relation R and outputs a new relation based on the subscript of the GP. The subscript specifies the computation to be done on R. 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 $\pi_{A*B}(R)$ is written as the following SQL query:

   select $(A*B)$ **from** R **groupby** $(A*B)$.

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.

If a GP has 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 in which case multiple SQL queries may be required to express them (refer to Example 3.1). Conversely, an SQL aggregate query can be written using GPs, possibly more than one (refer to Appendix A). In the next few paragraphs we restrict ourselves to single aggregate component GPs for the sake of clarity. For such GPs 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. It is simple to observe that a GP has exactly one tuple for each value of the groupby components and thus *does not produce any duplicates in its output.* The conventional projection operator with **set** semantics fits quite naturally into this framework because the conventional projection $\pi_D^{\mathbf{distinct}}(R(D,S))$ can be represented using GPs. That is,

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

In general, a GP behaves exactly like a conventional projection with set semantics if the GP has no aggregate components, and is the same as a groupby clause in SQL. We exploit this equivalence to push duplicate elimination down query trees by pushing GPs and later replacing them with **groupby** operators. The observation that duplication elimination has been pushed down has been made earlier by researchers [CS94]. We defer to Section 6 a discussion of how to use GPs to represent conventional projections that preserves duplicates.

The algorithm provided in this paper give rules for pushing generalized projection operators down a query tree. Given that GPs can represent aggregation and projections (**distinct** and duplicate-preserving), we provide a unified method for pushing aggregates, duplicate elimination, and column elimination down a query tree. Pushing generalized projections also lets us create aggregations in query trees that do not originally use aggregation.

[1]We refer to the relational projection operator explicitly as a conventional projection

Set Queries with No Aggregate Subqueries

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 the general case 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, past 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 illustrates how.

EXAMPLE 3.1 Consider a GP π_A being pushed down a query tree, past 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. The selection predicate takes as input a relation that has (at least) three attributes A, B, C because these attributes are needed to evaluate the selection. The question is if it is possible to eliminate some tuples from the relation input to the selection node.

Eventually we want all those A values that have associated with them a B value that satisfies $\sigma_{B \geq C}$. Consider two tuples $t1 = (a, 20, c1)$ and $t2 = (a, 40, c1)$. Whenever $20 \geq c1$ then $40 \geq c1$. That is, if tuple $t1$ satisfies the selection predicate then so does tuple $t2$. Note, 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. In general, if a tuple with a non-maximum value of B for given values of A, C satisfies the selection predicate, then the tuple that has the maximum B value for the same A, C value will also satisfy the selection predicate. Thus, we can discard all non-maximum values of B and pick only the maximum value for each value of A, C. Similarly, we can argue that for each value of A, B only the minimum value of C is relevant.

At GP push down time it is not known which of the two alternatives is more attractive and we annotate GPs in a manner that allows both options to be pushed down the query tree. Thus, we push the GP $\pi_{A,\top(B),\perp(C)}$ past the selection node, where function $\top$ says that the maximum value of B is relevant and function $\perp$ says that the minimum value of C is relevant. Similar to conventional projection pushing, we keep the original GP when we push GPs. In this case we keep π_A above the selection node. □

In the above example the attributes B, C are merely filters and their actual values are not important. In general, an attribute that occurs in an arithmetic comparison $>, \geq$ is merely a "filter" and it is useful to retain only its maximum or minimum value and not all its distinct values. This property is used to introduce aggregates in query trees with arithmetic comparisons. Also note, in the above example that the structure of tuples (A, B, C) has to be maintained while computing the maximum value of B or the minimum value of C. That is, it is not correct to pick the **max** value of B and **min** value of C for a given value of A, since **max**(B) and **min**(C) might come from different tuples that have the same A value. We use the functions $\top$ and $\perp$ to denote computing the maximum and minimum value while preserving the tuple structure. $\top$ and $\perp$ are different from **max** and **min** because the latter aggregate computations break tuple associations. Below we describe how to evaluate a GP that has one $\top$ label. We discuss how to evaluate GPs with multiple aggregate components in Appendix B. Note, GPs with multiple aggregate components also can be expressed as ordinary SQL aggregate computations and can be evaluated by aggregate-groupby operators.

Consider the GP $\pi_{A,\top(B)}(R(A, B))$. The GP is equivalent to the SQL query:

select $A, \mathbf{max}(B)$ **from** R **groupby** A.

Expressions of attributes can be used in place of attributes: pushing π_A past $\sigma_{(B+2*D \geq C)}$ gives $\pi_{A,\top(B+2*D),\perp(C)}$.

The above discussion gives a flavor of how a generalized projection acquires components when it is pushed down a query tree. The acquired components may be **groupby** components like A or aggregate components like $\top(B)$. Generalized projections encounter different types of nodes as they are pushed down a query tree. Below we give the intuition for how GPs change as they are pushed past selections and cross products.

Selection Nodes: A projection $\pi_{A,\top(B)}$ when pushed past a selection $\sigma_{C=D}$ becomes $\pi_{A,C,D,\top(B)}$. Why? The original projection $\pi_{A,\top(B)}$ requires the maximum value of B for each value of A. In addition, the GP introduced below the selection predicate $\sigma_{C=D}$ should retain all the information needed for evaluating the selection $\sigma_{C=D}$ and for computing $\pi_{A,\top(B)}$. The selection $\sigma_{C=D}$ requires all distinct values of C and D. Thus, the new projection $\pi_{A,C,D,\top(B)}$ is forced to **groupby** A, C, D and computes the maximum value of B for each distinct value of A, C, D. This computation of maxima over a set of **groupby** components (A, C, D) that is a superset of the **groupby** components (A) of the original projection leads to the computation of "partial" maxima. To compute $\pi_{A,\top(B)}$ now, we need to take the maxima of all these partial maxima which is what $\pi_{A,\top(B)}$ does; in other words $\pi_{A,\top(B)}(\pi_{A,C,D,\top(B)}) = \pi_{A,\top(B)}$. [2] This reasoning also gives an indication of why aggregates, like **avg**, whose computations cannot be decomposed into "partial" computations of the same type cannot be pushed down a query tree. However, we can push such aggregations down by expressing them in terms of other aggregates that can be decomposed. For instance, **avg** can be broken down into **sum** and **count** and then pushed down (refer Section 6). When we push the projection $\pi_{A,\top(B)}$ past the arithmetic comparison $\sigma_{C \geq D}$ we get $\pi_{A,\top(B),\top(C),\perp(D)}$, as discussed earlier.

Cross Product Nodes: Let a GP $\pi_{A,A',\top(B)}$ encounter a cross product node with attribute A' going down the right branch and A, B down the left. As in the case of conventional projections, $\pi_{A'}$ is pushed down the right branch and $\pi_{A,\top(B)}$ is pushed down the left branch. Why? Consider the query tree before $\pi_{A,A',\top(B)}$ is pushed down the cross product. Let there be k tuples with a given value of $A' = a'$ in the right branch. Each tuple in the left branch is repeated k times in the cross product due to its association with $A' = a'$. Thus each B in the left branch now occurs k times in the cross product with $A' = a'$. This multiplicative effect of the cross product does not matter for operations like **max** and $\top$, since repeated occurrences of the same element in a set do not change the **max** of the set. Thus, we can push $\pi_{A,A',\top(B)}$ past the cross product with $\pi_{A'}$ going down the right branch and $\pi_{A,\top(B)}$ down the left. In other words: $\pi_{A,A',\top(B)}(\pi_{A,\top(B)} \times \pi_{A'}) = \pi_{A,A',\top(B)}$. Aggregations like **sum** do change though, when elements of a set are repeated and have to account for the multiplicative effect of the cross product.

Besides selection and cross product nodes, a query tree also has groupby and aggregate-groupby node pairs. Groupby nodes and aggregation-groupby node pairs can be rewritten as GPs. In our discussion henceforth, we assume that a query tree is rewritten to use generalized projections in place of groupby nodes and aggregate-groupby node pairs. This rewrite does not reduce the expressiveness of the language we are considering. It only makes the discussion simpler and more intuitive. Another simplification we make is that we eliminate interior projection nodes from the query-tree. Intuitively, these nodes can be eliminated because interior duplicate-preserving

[2] Assume for now that the result of a unary aggregation over column X is referred to as X. Thus, we use B to refer to the result of $\top(B)$.

projections do not affect the final result. We eliminate them to simplify the discussion. Also note that pushing GPs is strictly more powerful than pushing conventional projections. Thus, if a redundant column can be discarded safely by an interior projection, then the GP pushing algorithm also infers that the column can be discarded.

Thus, we consider pushing GPs past selection nodes, cross product nodes, and GP nodes.

In the next section we describe a two step algorithm that in its first step pushes GPs down query trees. The GPs introduced in the tree may not all improve the query plan. Thus, some GPs generated in the pushing process may have to be eliminated. The query optimizer decides which GPs it wants in the final query tree. Our algorithm eliminates unwanted GPs in its second step and does not require that any intermediate GPs generated be present in the final query tree.

4 Queries with Set Semantics

In this section we develop the GP-pushing algorithm for queries whose trees have no aggregate nodes, and where the topmost projection node eliminates duplicates. The selection nodes are a conjunction of predicates that use the operators $<, >, \leq, \geq, =, \neq$.

First, we formally define the syntax of generalized projections. The syntax is slightly different from what we have used until now. The difference is that for each aggregate function in the GP we introduce a variable to hold the result of the aggregate function.

P: $\pi_{(U,V,\ldots,W,R=Lx(X),S=Ly(Y),\ldots,T=Lz(Z))}$

Each of U, V, W, X, Y, Z may be a function of attributes. For instance, X may be $f(A, B)$ where A, B are attributes of the relation over which P is computed. Projection P is computed over an input relation that provides the columns used to define components $U, V, \ldots, W, X, Y, \ldots, Z$. The relation is grouped by $U, V, \ldots, W$. For the aggregate components $X, Y, \ldots, Z$ the result of the aggregate function $Lx, Ly, \ldots, Lz$ is assigned to the associated variable $R, S, \ldots, T$. The result of applying P to an input relation is a relation that has attributes $U, V, \ldots, W, R, S, \ldots, T$. These attributes are used by nodes and GPs that occur above P in the tree.

4.1 Strategy

GPs are incorporated into query trees using a two step process:

1. Push GPs down a query tree and annotate the query tree with a GP above each node in the tree.

2. Rewrite the annotated query tree to incorporate the GPs that the query optimizer chooses to evaluate and to eliminate all other GPs introduced in the push-down process.

The annotation is done in a a top-down pass in which GPs are pushed down a node at a time. When a GP P is pushed past node N, P and N are rewritten to use the attributes defined by the new GP Q that is introduced below N. Thus, after the pushing step GP P stays above N, albeit altered a little. The GPs can be pushed all the way down to the leaves of the tree, or to any point in the tree as dictated by the optimizer. Finally, once the query tree has been annotated with generalized projections, the query optimizer chooses only a few GPs as being advantageous to evaluate. Note, the optimizer can choose to evaluate *any* subset of the GPs introduced by the top-down pass. Also, the optimizer is not forced to evaluate any GP because it decided to evaluate some other GP. All the GPs not selected by the optimizer are removed from the tree in a bottom-up pass that rewrites the nodes and remaining GPs appropriately. The following example illustrate the two passes.

4.2 Example

Consider query $Q2$ from Figure 3 in Example 1.1 (page 3). We have modified the original query tree to express joins as cross products followed by selections. First, we consider the top-down pass that pushes generalized projections down the query tree.

Figure 6 shows the top-down pass. In the figure we write generalized projection $\pi_{A,\cdots}$ as $(A, \cdots)$. The GP at the very top of the tree is $\pi_{M\#}$. The left half of the figure shows how $\pi_{M\#}$ changes when pushed past the topmost selection node. As explained in Section 3, only the minimum value of $(0.85{*}SP)$ and maximum value of $(CP{*}Factor)$ need to be computed below the selection node. Also, $St, St', M\#, M\#'$ are added as groupby components because their values are needed in equijoins and in the answer. After the push down, the selection node is rewritten to use the output of the GP; namely the attributes A and B are used by the selection instead of $SP, CP, Factor$. The right half of the figure shows the GPs obtained after a further push down step, past the topmost cross product.

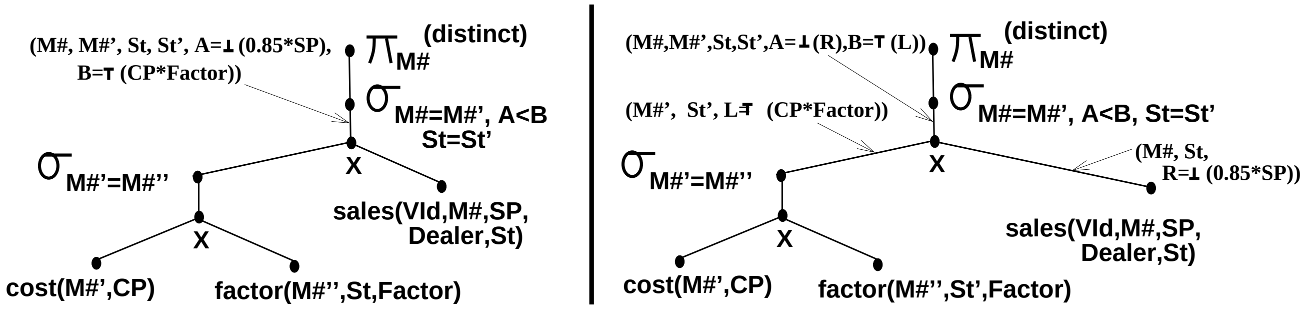

Figure 6: Pushing GPs down Query $Q2$

The GP above base relation `sales` cannot be pushed further. However, on the left branch the GP can be pushed further down as discussed in Section 4.3. For the time being, we assume that the GP push-down process stops here.

Now, the query optimizer chooses which of the GPs introduced in the tree should be evaluated. For example, say the optimizer decides to evaluate only the generalized projection above relation `sales`, *i.e.*,

$$\pi_{M\#,St,R=\top(0.85*SP)}$$

(circled in Figure 7). This GP summarizes the `sales` table which can be expected to be very large. The GP computes the minimum value of $(0.85 * SP)$ for every model and state value. This aggregate computation reduces the size of the `sales` table from the number of cars sold by the company, to a table with as many tuples as the product of the number of distinct $(M\#, St)$ value pairs.

Now, we proceed bottom-up and eliminate the other GPs from the tree. First we eliminate $\pi_{M\#',St',L=\top(CP*Factor)}$. The elimination results in the query tree shown in the left half of Figure 7. Next, the GP above the cross-product is eliminated to give the query tree in the right half of the figure. The highlighted parts of the trees show the rewriting that is done as a part of the elimination process. The GP over relation `sales` can be written as an aggregation computation as shown in Figure 4.
In the next section we describe in detail the algorithm for pushing GPs down query trees and for rewriting the query tree based on the GPs that are selected by the optimizer.

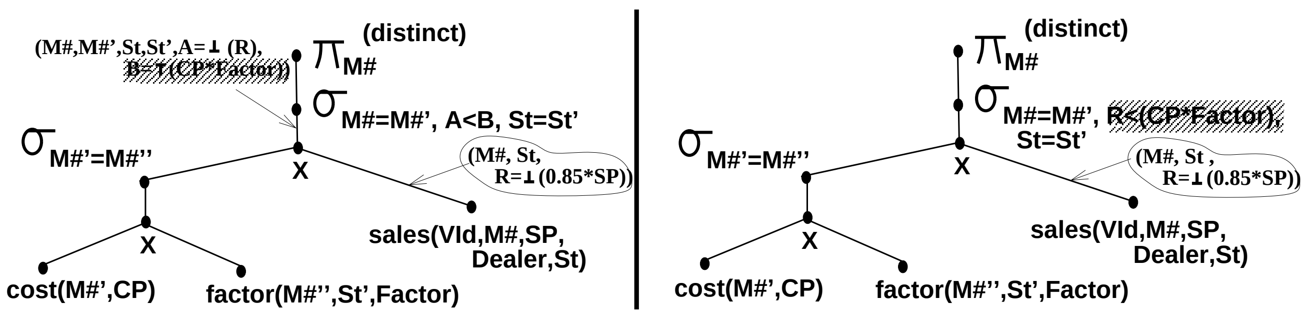

Figure 7: Eliminating GPs not selected by the query optimizer

4.3 Top-down Pass

We present the algorithm for pushing GPs down a query tree in the form of a table that gives the algebraic transformations needed for pushing GPs. In later sections, the table is enhanced to handle queries with aggregates like **sum** and to handle queries that preserve duplicates.

The topmost GP is the same as the projection node that specifies the attributes in the final answer. As a GP P is pushed down the tree, the intermediate nodes determine how P changes. As mentioned before, a query tree has three types of internal nodes, selection, cross product, and GP. In this section we are considering queries that have no aggregate subqueries. Thus, we consider GP nodes N that have only groupby components, and no aggregate components. Table 1 summarizes how to determine the GP Q below a node N when a GP is a P is pushed past N. We use the shorthand "agg" for "aggregate," "cmp" for "component," "gby" for "groupby," and "attrs" for "attributes."

Node Type	Contents of N	Add this to P to get Q	Effect on N	Effect on P
Select	$X = Y$	X and Y as gby cmps	No effect	No effect
	$X \geq Y$, $X > Y$	$X' = \top(X)$, $Y' = \bot(Y)$ 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
	$X \geq c$ $X \leq c$	$X' = \top(X)$ $X' = \bot(X)$	replace X by X' replace X by X'	same as above same as above
Cross Product	Separate GPs are pushed down the two branches L, R. Table 2 describes how to infer the GP for only branch L. GP for branch R is derived similarly.			
GP	set of gby attrs.	Copy P into Q	No effect	Eliminate P

Table 1: How to push generalized projections for **distinct** queries

The algorithm is very similar in structure to pushing conventional projections down a query tree and the only differences are the introduction of the $\top$ and $\bot$ functions and the new attribute names.

Note, each GP computes a conservative estimate of the amount of information needed above any node. Thus, we do not lose any answers by computing GPs. This claim can be proved correct by using an inductive argument.

Theorem 4.1 *Consider for a* **distinct** *query Q the corresponding query tree that uses only selection, X-product, and GP nodes, and that has a distinct projection node at the top. After GPs have*

component type	Property	Effect on P	Effect on Q_L
Groupby X	Uses attributes from only L	No effect	Add X as a gby cmp of Q_L
	Uses attributes from L, R	No effect	All attributes of L that are used in X, are made gby cmps of Q_L
aggregate $X = \top(Y)$	Uses attributes from only L	Replace $X = \top(Y)$ by $X = \top(Y')$	add $X' = \top(Y)$ to Q_L
	Uses attributes from L, R	No effect	All attributes of L that are used in Y, are made gby cmps of Q_L

Table 2: How to get the GP Q_L for branch L when GP P is pushed past a cross product

been pushed down this tree according to Table 1, the resulting tree computes the same answer as query Q. $\square$

Proof: (Outline:) GPs are pushed down one node at a time. Thus the above theorem is proved by induction over the height of the query tree. For each type of node, we prove that the answer computed by the node is unchanged after a GP is introduced below the node. ∎

Appendix B discusses some subtle but interesting issues related to evaluating GPs.

The next section discusses the bottom-up phase of rewriting query trees after the query optimizer selects the GPs that should be computed.

4.4 Bottom-up Pass

After the top-down pass associates a GP above some or all nodes of the query tree, the query optimizer decides which GPs improve the query plan. The other GPs are removed from the tree. To remove a GP Q that is rejected by the optimizer, we rewrite the node N above Q and the GP above node N. The rejected GPs are eliminated bottom-up. We illustrate the elimination process by referring back the discussion of Section 4.2:

EXAMPLE 4.1 In Figure 6 the query optimizer decides to compute only the GP above relation `sales`. The other GPs are eliminated. First we eliminate the GP $\pi_{M\#',St',L=\top(CP*Factor)}$ from the left branch. The elimination causes component L in all higher GPs to be replaced by the definition of L, namely $\top(CP*Factor)$. The left tree in Figure 7 shows the resulting tree. Next we eliminate the GP on the stem of the tree. Thus, components A and B are no longer available to higher nodes and GPs. In the selection node, A is replaced by R and B is replaced by $(CP*Factor)$. $\square$

The bottom-up rewrite pass is an exact inverse of the push-down process. Thus, the rules used to push down the generalized projections are also used to eliminate GPs.

5 Aggregates: max and min

In this section we extend the class of queries considered in Section 4. We show how to push GPs down query trees for queries that have set semantics, use arithmetic comparisons, and that may use aggregate subqueries with the **max** and **min** operators. In section 6 we generalize our results to also handle **sum**, **count**, **avg**, and other aggregate functions. We treat **max**-**min** differently from **sum**-**count**-**avg** because **max** and **min** do not need duplicate semantics in the underlying set. By contrast, a subquery that uses **sum** requires duplicate semantics in its subtree even though the top-level query may be a **distinct** query.

GPs can be pushed down queries that use **max** and **min** subqueries, with little change from the way GPs are pushed down SQL queries that use arithmetic comparison operators. Recall, we use generalized projections to write aggregate-groupby node pairs. Thus, in this section the GPs that are pushed down query trees use two more aggregate functions: **max** and **min**. There is no difference in the way GPs with functions **max** and **min** are pushed past selections and cross products. The only difference occurs when a GP P is pushed past a GP node N that has aggregate functions. Intuitively, the resulting GP Q has aggregate components corresponding to **max** and **min** computations in N. We give the algorithm for pushing GPs past **max**-**min** as a part of the complete GP pushing algorithm, in Table 3.

max seems similar to $\top$ because both $\top$ and **max** compute the maximum value of an attribute. However, to see the difference consider a GP with two components labelled **max**. When this GP is computed, the maximum for both the attributes can be computed in the same groupby operation. However, as discussed in Section 3 and Appendix B if two components are labelled $\top$ we cannot compute the GP the same way. The difference between $\top$ and **max** arises because $\top$ maintains tuple association for the different components, while **max** does not.

The bottom-up pass is exactly the same as in the case when no aggregate computations were done in the original tree. It should be noted that the query optimizer may retain some or all the preexisting generalized projections (aggregate-groupby node pairs) because unlike the GPs introduced during push-down, all preexisting GPs are not optional. It is possible for our algorithm to eliminate correctly some classes of preexisting aggregations, but we do not discuss how to eliminate preexisting aggregations in this paper.

6 Duplicates: Intuition

Consider a simple select-project-join query tree with only conventional projections (c-projections). It can be rewritten using exactly one c-projection occurring at the top of the tree. The query tree may have either duplicate or set semantics, that is, it may or may not preserve duplicates. There is no way to tell by just looking at the query tree which of these semantics is required. Since the underlying base relations have set semantics (no tuple occurs more than once), duplicates are created only when we delete columns. Thus it is the conventional projection (c-projection) that determines if set or duplicate semantics are to be followed. The joins and selections behave identically in both cases. The set semantics c-projection and the duplicate semantics c-projection are two very different operators. The set semantics c-projection does duplicate elimination and thus drops rows as well as columns; more importantly in a query tree, it does not require duplicates be preserved in the nodes below it (its input) for correctness. The duplicate semantics c-projection on the other hand just drops attributes and may produce a table with duplicates as output and thus requires that duplicates be preserved in its input. The conventional projection is overloaded with both these meanings and depending on which meaning we assign, we get the corresponding semantics for the query tree.

The output of a generalized projection is always a set, *i.e.*, there are no duplicates. It is still possible to write a duplicate semantics c-projection as a GP. For instance, the c-projection π_A that preserves duplicates is written as the GP $\pi_{A,\mathbf{count}(*)}$. There is only a syntactic difference in the results of these two projections, even though the output of the latter is a set and the output of the former may have duplicates. By dropping duplicates and incorporating a **count** column to indicate multiplicity, we do not lose any information: it is easy to go from one form to another. Thus we do not need to change the output semantics of GPs to accommodate duplicates. GPs only produce sets as outputs (no duplicates). Duplicates semantics are simulated using **count**.

It is not the output, but the requirement on the input that distinguishes the conventional set semantics and duplicate semantics projection. For instance, while aggregations like **max** do not require duplicates to be preserved in their input, aggregations like **sum** do require duplicates. It is possible to determine if the input to a generalized projection should have duplicates by looking at the components of the GP. Thus, the GP pushing algorithm should ensure that when a GP P is pushed past a node N, the resulting GP Q does not destroy duplicates if either P or N need duplicates.

For notational convenience only, we use π^{set} to denote a generalized projection that does not require duplicates to be preserved in its input. π^{dup} denotes a projection that requires duplicates to be preserved. It is important to note that the superscripts *set* and *dup* are purely syntactic sugar. The components of the generalized projection unambiguously tell us if duplicates are needed in the input: GPs are labelled π^{dup} if they have aggregation components which require preservation of duplicates in their input like **sum** and **count**; otherwise they are labelled π^{set}. Thus for example, GP $\pi_{A,\mathbf{sum}(B)}$ must have superscript *dup*, GP π_A must have superscript *set*.

In the previous sections we have concerned ourselves with pushing π^{set} down a query tree. In the following discussion we shall focus on π^{dup} and give rules for pushing it down a query tree which will enable us to push aggregations like **sum** and also create aggregations like **count** where none existed. As before, the push-down process has two phases: top-down and bottom-up.

It should be noted again that our algorithm does not require any intermediate GP nodes be present in the final tree. The query optimizer can pick whichever GPs, and thus aggregations, it wants.

6.1 Top-Down with duplicates

We discuss how to push generalized projections down a query tree that maintains duplicate semantics. The first point to note is that a duplicate preserving GP cannot *introduce* the labels $\top$ and $\perp$ because both these labels are introduced based on the intuition that duplicates do not matter. However, $\top$ and $\perp$ may already exist in a GP before it gets duplicate semantics, for instance when a distinct query has a subquery that uses **sum**. Thus, we do have to consider pushing $\top$ down trees with duplicate semantics.

First, let us see intuitively how π^{dup} behaves when pushed past selection and cross product nodes.

Selection Nodes: When we push a π^{dup} past a selection node we just acquire the attributes occurring in the selection nodes. Unlike with π^{set}, arithmetic comparisons do not help in creating aggregates. To see why, consider pushing a projection $\pi^{dup}_{A,\mathbf{count}(*)}$ past 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 past $\sigma_{C \geq D}$ 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 wanted a number. So on pushing $\pi^{dup}_{A,\mathbf{count}(*)}$ past $\sigma_{C \geq D}$ we get $\pi^{dup}_{A,C,D,\mathbf{count}(*)}$. In Table 3 we describe the syntactic changes that need to be made to the tree when a π^{dup} is pushed past a selection node.

Cross Product Nodes: As explained in Section 3, a cross product has a multiplicative effect.

If we push an aggregation computation down one branch we have to account for the multiplicative effect of the other branch. For instance, let $\pi^{dup}_{A,A',X=\mathbf{sum}(B)}$ be pushed down a cross product where attributes A, B go down the left branch and A' down the right. If we push $\pi^{dup}_{A,X=\mathbf{sum}(B)}$ down the left branch we get partial sums for B for each value of A. Note, that each B will repeat itself above the cross product due to its association with the each tuple from the right branch. Thus, if there are k tuples with $A' = a'$ then each B value on the left hand side occurs k times with $A' = a'$. If we push $\pi^{set}_{A'}$ down the right branch we lose the information that $A' = a'$ occurs k times and consider each B only once for $A' = a'$ while computing the final sum from the partial sums provided by the left branch. What is missing is the value of k and we get it by pushing a **count** down the right branch. That is, we push $\pi^{dup}_{A,L=\mathbf{sum}(B)}$ down the left branch and $\pi^{dup}_{A,R=\mathbf{count}(*)}$ down the right branch. The original projection is rewritten as $\pi^{dup}_{A,X=L*R}$. The function (multiplication:'$*$') with which we compose the count (R) with the partial aggregate computed (L) to get the total aggregate (X) depends on the aggregation (**sum**). For common aggregations like **sum** and **count** this function is $*$. In Table 5 we enumerate how to decompose (and compose) some commonly occurring aggregate functions. For Table 5 we assume that the aggregates are all single argument aggregates and that the argument is provided by the left branch of the cross product. Aggregate functions that use multiple attributes as arguments can also be decomposed when a cross product node is encountered. For instance, most statistical aggregations can be decomposed and pushed.

For query trees that have either select, cross product, or GP interior nodes, and that have either a set or duplicate preserving projection node at the top of the tree, Table 3 gives the algebraic transformations needed to push GPs down. As before, we state the algorithm in tabular form to facilitate understanding. The corresponding version of Theorem 4.1 that uses Table 3 holds for the more general trees. The bottom-up pass uses Table 5 when eliminating any GP that was obtained by decomposing aggregates at a cross product node.

Node Type	Contents of N	To get Q, add this to P	Effect on N	Effect on P
Select	X op Y	X and Y as gby cmps.	nothing	Replace each agg cmp as per Table 4
X-prod	Table 5 describes how to infer the GP for branch L for some functions. For $\top, \bot$, **max**, **min** the discussion in Table 2 applies			
GP	has only gby components	GP becomes π^{set} from π^{dup}. Rules of Section 5 become active.		
	$X = \mathbf{agg}(Y)$	If X is used in P add $X' = \mathbf{agg}(Y)$	Replace $X = \mathbf{agg}(Y)$ by $X = \mathbf{agg}(X')$	No effect
	gby(X) [3]	Add X as gby attr	nothing	nothing

Table 3: How to push GPs

[3] An aggregate component of GP P can appear as an aggregate component in Q even when the component appears as a groupby attribute in node N [HG94]. We omit discussing these further optimizations to retain clarity of the discussion.

Component form	What to add to P to get Q	Effect on P
$X = \top(Y)$	$Y' = \top(Y)$	Replace $X = \top(Y)$ by $X = \top(Y')$
$X = \mathbf{max}(Y)$	$Y' = \mathbf{max}(Y)$	Replace $X = \mathbf{max}(Y)$ by $X = \mathbf{max}(Y')$
$X = \mathbf{min}(Y)$	$Y' = \mathbf{min}(Y)$	Replace $X = \mathbf{min}(Y)$ by $X = \mathbf{min}(Y')$
$X = \mathbf{sum}(Y)$	$Y' = \mathbf{sum}(Y)$	Replace $X = \mathbf{sum}(Y)$ by $X = \mathbf{sum}(Y')$
$X = \mathbf{count}$	$Y' = \mathbf{count}$	Replace $X = \mathbf{count}$ by $X = \mathbf{sum}(X')$
$X = \mathbf{avg}(Y)$	$X_s = \mathbf{sum}(Y)$ $X_c = \mathbf{count}$	Replace $X = \mathbf{avg}(Y)$ by $X = \mathbf{sum}(X_s)/\mathbf{sum}(X_c)$

Table 4: How to push aggregate components through selections (used in Table 3)

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

Table 5: Decompositions for some commonly used aggregation functions

7 Difference and Union Queries

In this section we give only the intuition for pushing GPs past union and difference nodes, assuming that the query tree has been extended with the appropriate nodes. Note, both difference and union nodes take as input two relations that have the same set of attributes. We assume we are dealing with multiset difference and union. First we discuss briefly difference nodes.

Consider the difference of relations $R(A, B)$ and $S(A, B)$. Let the GP $\pi_{A,\top(B)}$ be present above the difference node $R-S$. That is in $R-S$ we need the maximum value of B for each value of A. It is not possible to push $\top$ down R because the tuple with maximum B value in R may also occur in relation S, and thus will not occur in the difference. Potentially no tuple in R can be eliminated before the difference because that tuple may be the only tuple in the difference of $R-S$. Thus, aggregate component $\top(B)$, when pushed past a difference, becomes groupby component B. Similarly, it is not possible to push any of the aggregate computations $\perp$, **max**, **min**, or **sum**, past difference nodes. However, it is possible to push the aggregates **count** and **avg** past difference [HG94]. Set difference is similarly handled.

Consider the union of relations $R(A, B)$ and $S(A, B)$. Let the GP $\pi_{A,\top(B)}$ be present above the union node $R \cup S$. That is, in $R \cup S$ we need the maximum value of B for each value of A. The maximum values B for each A, from relations R and S can be used to obtain the maximum value after the union. Intuitively, this is possible because the maximum of the union must be the maximum of one of the two input relations. Similarly, it is possible to push $\perp$, **max**, **sum**, **count**, and **avg**, past unions. Set based unions can be handled using a conservative approach that composes a duplicate preserving union node with a **groupby** node.

For both difference and union nodes, the bottom-up pass is more involved than before because it is necessary to ensure that the schema of the two input relations is the same. That is, the GPs that are pushed past the union and difference nodes all are not independent. Thus, it is necessary to consider the GPs selected for evaluation in the left branch of the union, while choosing the GPs that are evaluated in the right branch. Some GPs may necessarily have to be evaluated if some other GPs are selected.

The ability to push aggregations down union nodes is of importance in distributed database systems where data may be horizontally partitioned [OV91]. Pushing aggregations in such an environment helps increase parallelism since aggregation computation in different branches of a union node can be done simultaneously on different sites.

8 Performance Issues

No optimization technique reduces the cost of query execution in all cases. There are always cases where the cost of doing the optimization is greater than the benefit. In other words, the number of irrelevant tuples removed may be too small to warrant doing an operation early. For an optimization technique to be practically useful, it should provide considerable cost reduction improvements in a significant fraction of the queries executed. In this section, we identify the issues relevant to the performance of the optimization technique we outline in this paper.

Our algorithm works best on queries when the **groupby** attributes we push down do not have too many distinct values in the underlying relation. In other words, the less key-like a **groupby** attribute is, the more useful our optimization is. Most joins involve at least one non-key attribute because otherwise the relations being joined would not have been stored as separate relations in the first place. The number of distinct values of an attribute in a relation is a commonly maintained database statistic [OV91] and so the query optimizer can have a good estimate of the cost of performing the aggregate and also of the number of tuples in the output.

In typical decision support queries there is often at least one very large relation involved in the query. This relation often is time dependent: for example, the sales history of the previous year. The historic nature of data results in enormous relations. Most queries are not interested in individual tuples of this relation, but rather aggregate properties of this relation. Thus in most cases, we need to do a **groupby** on a non-key attribute of this relation. When this relation is joined with some other relation, it is often with a smaller "reference" or "algorithm" table, that need not be aggregated. In such cases, our optimization technique would reduce considerably the size of the massive table before we did a join. It can be argued that in such cases a join algorithm like a hash join could be used to achieve a similar result. However, hash joins are difficult to implement in practice and not commonly implemented. Single table aggregations being a commonly used feature of SQL exist in most systems. Our technique only requires the use of these operators. In addition, our technique works in many cases where hash joins do not do well: for instance, if two very large tables were joined. Note, decision support queries also use non-equijoin joins that can be optimized by introducing aggregations as illustrated by query $Q2$ in Example 1.1.

Indexes on the **groupby** attributes are of great help and can be used to do efficient aggregate-groupby operations. In the absence of indexes, for our technique, it would help greatly if hashing is used in doing the **groupby** rather than a sort [HJT94]. In such a scheme, we read in the large table and hash each of its tuples on the **groupby** attributes and then update the aggregation required which we maintain in the hash table. Typically our optimization will be picked if the number of distinct values of the **groupby** attributes are small. In such cases, the hash table will fit in memory and so the entire aggregation-groupby operator scans the table just once and reduces it tremendously. Single-table aggregation-groupby operators are not the focus of this paper and we do not go into any more details. It suffices to note that most commercial systems have existing efficient implementations that our technique uses.

Our optimization, when applied to query plans, potentially interferes with join ordering, since we reduce the size of the relations participating in the join. However, the technique can be used advantageously as a post join-ordering step. Note, for greater performance gains our push-down algorithm should be integrated with the join ordering module.

Our algorithm can be used in computing semijoins efficiently. Current semijoin algorithms do not treat non-equijoins any differently from equijoins. Using our algorithm we can use such non-equijoins to reduce the size of the semijoin through aggregate computation. The semijoin of a relation R with respect to a relation S and a join predicate F is given by $\pi_R(R \bowtie_F S)$ where the R subscript for the conventional projection operator π is used to indicate the attributes of R. It should be observed that the c-projection π_R has set semantics and is thus equivalent to the GP π_R. We can push this GP past the join using our algorithm and thus apply **groupby** and aggregations in computing a smaller semijoin reducer. Using aggregations can decrease the size of a semijoin greatly, when non-equijoins are present. Reducing the size of a semijoin has considerable impact in distributed query processing where semijoins are used frequently in computing joins [OV91]. Though we have only mentioned non-equijoins here, the GP pushing algorithm can be used to find efficient semijoin reducers for any simple SPJ query with aggregation and **groupby** nodes.

In summary we wish to stress again that all the building blocks required to implement our optimization are already present in most commercial systems. All we are doing, in some sense, is to link them together. Our method will be most useful when there are massive underlying relations and we are required to do joins on them. We believe that with the advent of decision support systems and the ever increasing size of databases, our optimization is viable in a practical setting.

9 Conclusions and Future Work

In this paper we introduce generalized projections (GP) that unify aggregations, groupbys, conventional projection with duplicate elimination (**distinct**), and duplicate preserving projections. We describe how to push GPs down query trees that use arbitrary aggregate functions and that may or may not preserve duplicates. This treatment allows us to create and push aggregations and duplicate elimination in a uniform way. We are able to use aggregations to substantially reduce the size of intermediate relations in duplicate preserving queries, without changing the way joins are done. We are also able to push function computations to lower levels in a query tree. We believe that our technique yields new insight into how to write and optimize SQL queries with aggregation, arithmetic, and function computations.

As future work we are looking at the following problems:

- Extend our technique to queries that use **exists**, **in**, and more involved SQL constructs.
- Carry out experimental performance studies.
- Explore how to eliminate preexisting aggregations. It is different from eliminating the optional GPs introduced by GP-pushdown.
- Integrate our technique with predicate-move-around [LMS94] to see if the tighter constraints derived by their algorithms can be used by our algorithm.
- It appears that GPs can be used for doing constraint checking and data warehousing [Z+94] efficiently. We are investigating this relationship currently.

Acknowledgements

We would like to thank Surajit Chaudhuri, Laura Haas, Jeff Ullman, and Jennifer Widom for valuable comments and suggestions.

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.

[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.

[DGK82] U. Dayal, N. Goodman, and R. H. Katz. An Extended Relational Algebra with Control over Duplicate Elimination. In *Proceedings of the ACM Symposium on Principles of Database Systems, 1982*, pages 117-123.

[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.

[HG94] Venky Harinarayan and Ashish Gupta. Generalized Projections: a Powerful Query-Optimization Technique. Technical report, Department of Computer Science, Stanford.

[HG95] Venky Harinarayan and Ashish Gupta. Optimization Using Tuple Subsumption. To appear in *ICDT 95*, January 1995.

[HJT94] F. Ho, R. Jain and J. Troisi. An Overview of NonStop SQL/MP. In *Tandem Systems Review*, pages 6-17, July 1994.

[ISO92] ISO. Database Language SQL ISO/IEC. Document ISO/IEC 9075:1992. Also available as ANSI Document ANSI X3.135-1992, 1992.

[LCW93] H. Lu and C. C. Chan and K. K. Wei. A Survey of Usage of SQL. In *SIGMOD Record*, Vol 22, No. 4, 1993.

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

[OV91] Tamer M. Oszu and P. Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[S79] P. G. Selinger et. al . Access Path Selection in a Relational Database Management System. In *Proceedings of ACM SIGMOD 1979 International Conference on Management of Data*, pages 23-34, 1994.

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

[YL93] W. P. Yan and P. A. Larson. *Performing Group-By Before Join.* In *International Conference on Data Engineering*, 1993.

[Z+94] Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, and Jennifer Widom. View update anomalies in a warehousing architecture. unpublished manuscript, October 1994.

A Writing SQL Aggregate Queries Using GPs

We show here how to represent SQL aggregate-groupby queries using GPs. Consider an SQL query Q that we wish to write as a GP P. First we write all the attributes occurring in the **groupby** clause of Q as the groupby components of P. We then append to P the aggregate computations that occur in the **select** clause of Q. The resulting GP P is equivalent to the given SQL query Q if the **groupby** attributes of Q all occur in the **select** clause of Q. Otherwise, we have to project out those **groupby** attributes of Q that do not occur in the **select** clause. That is, we need to follow GP P with a conventional projection. Since we can express conventional projections using GPs, we can write any SQL aggregate query using GPs.

B Evaluating GPs

B.1 How should generalized projection of the form $\pi_{D,\top(S),\bot(A)}$ be evaluated?

It can be shown that $\top(E) = \bot(-E)$, where E is any expression involving attributes. So $\pi_{A,\top(B),\bot(C)}$ can be rewritten as $\pi_{A,\top(B),\top(-C)}$ which is

E: **select** A, **max**$(B, -C)$ **from** R **groupby** A.

max(B, C) is not the same as **max**(B), **max**(C). **max**(B, C) is the set of all B, C tuples $\{b_1, c_1\}$, such that there is no other tuple $\{b_2, c_2\}$, where $b_2 \geq b_1$ and $c_2 \geq c_1$. The following example illustrates this point:

EXAMPLE B.1 Let GP E be evaluated on relation emp$(Dept, Sal, Age)$ that has tuples $(toy, 21K, 10yrs)$, $(toy, 15K, 8yrs)$, and $(toy, 21K, 9yrs)$. GP E computes the least age for the most highly paid employee in each department and returns as answer the pair of tuples emp$(toy, 21K, 9yrs)$ and $emp(toy, 15K, 8yrs)$. If instead we computed **select** D, **max**(S), **max**$(-A)$ **from** R **groupby** D, we would get as answer the tuple emp$(toy, 21K, 8yrs)$. □

Thus, if several aggregate components use functions $\bot$ and $\top$, they all cannot be computed as a part of the same groupby. Rather, only one aggregate computation can be done at a time while treating all other components as groupby components. In order to compute a GP with k aggregate computations, k aggregate computations have to be cascaded where each step computes one aggregate while treating all other components as grouping attributes. It is not necessary to evaluate all the k aggregate computation step; any or all of them can be omitted without compromising correctness. That is, any or all the aggregation components of a generalized projection can be replaced by groupby components.

B.2 Can one GP be transformed into another GP?

EXAMPLE B.2 Consider the table $r(A, B, C, D)$ with the GP $P : \pi_{(D,U=\bot(A),V=\top(A),W=\top(B+C))}$. To compute the component $U = \bot(A)$, we find the minimum value of column A for each value of the triple $(A, B + C, D)$. This computation computes the minimum value for A while grouping by A and thus is a useless computation. Thus, the GP is better replaced by $\pi_{(A,D,W=\top(B+C))}$. The GP and node immediately above P are rewritten to use A in place of U and V because U and V are no longer computed by P. □

There are some obvious cases when a GP should be replaced by another. For instance, if a GP has both $U = \top(A)$ and $V = \top(A)$, then U and V can be both replaced by only one component. If $U = \top(A)$ and $V = \bot(A)$ both occur in P then U and V could be replaced by A as a groupby component. There are other simple rules that can be made available to a query optimizer to transform one GP into another GP.