GPIVOT: Efficient Incremental Maintenance of Complex ROLAP Views *

Songting Chen and Elke A. Rundensteiner
Department of Computer Science
Worcester Polytechnic Institute
Worcester, MA 01609-2280
(chenst | rundenst)@cs.wpi.edu

Abstract

Data warehousing and on-line analytical processing (OLAP) are essential for decision support applications. Common OLAP operations include for example drill down, roll up, pivot and unpivot. Typically, such queries are fairly complex and are often executed over huge volumes of data. The solution in practice is to use materialized views to reduce the query cost. Utilizing materialized views that incorporate not just traditional simple SELECT-PROJECT-JOIN operators but also complex OLAP operators such as pivot and unpivot is crucial to improve the OLAP query performance but as of now unexplored topic. In this work, we demonstrate that the efficient maintenance of views with pivot and unpivot operators requires the definition of more generalized operators, which we call GPIVOT and GUNPIVOT. We propose rewriting rules, combination rules and propagation rules for such operators. We also design a novel view maintenance framework for applying these rules to obtain an efficient maintenance plan. Extensive experimental evaluation reveals the efficiency of our proposed maintenance techniques. Our query transformation rules are thus dual purpose serving both view maintenance and query optimization. This paves the way for the inclusion of the GPIVOT and GUNPIVOT into any DBMS engine. Extensively performance study reveals the effectiveness of our proposed maintenance strategies.

1 Introduction

Data warehousing and on-line analytical processing (OLAP) are essential for decision support applications and have been a focus by both the research and industrial communities [4]. A data warehouse stores historical, summarized and consolidated data, important for complex trend analysis applications. The data in the data warehouse is typically multidimensional. Example dimensions for sales data are the product, location and time dimensions. Many complex transformations need to be supported, including *drill down*, *roll up*, *slice and dice* and *pivot*, in order to perform online analysis on such multidimensional data [4].

*This work was supported in part by NSF grant #IIS 9988776.

Relational database engines [15, 16] have been extended to natively support these OLAP operations in order to achieve better performance. One well-known example is the extension of the relational engine with CUBE and ROLLUP operators [10] to support multidimensional aggregation. Making such operators explicit to a relational database engine provides excellent optimization opportunities [9]. Another example is the inclusion of PIVOT and UNPIVOT operators into Microsoft SQL Server [8, 16] for efficient execution and optimization.

Beyond OLAP applications, such pivot and unpivot operators have also been shown to be useful for sparse dataset processing by storing such data in vertical format [2].

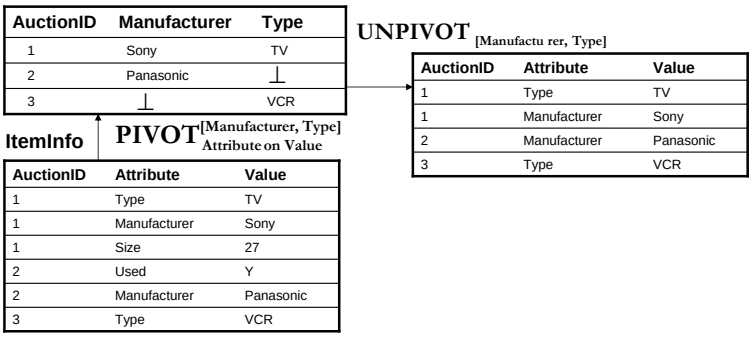


Figure 1: PIVOT and UNPIVOT Operators

For example, in Figure 1, the table $ItemInfo$ stores the attributes of each auction. Since there might be thousands of different item attributes while individual item may just have few of them, if we were to store the *ItemInfo* table *horizontally*, i.e., devoting one column to each auction attribute, we may have a table with thousands of columns filled with numerous NULL values. Hence, such data are instead stored in the vertical format. In other words, the attribute names are treated explicitly as data values and are stored pairwise with their corresponding attribute values.

The pivot operator transforms the vertical data into horizontal format. More precisely, we pivot the column 'Value' by the column 'Attribute'. Only the values of 'Manufacturer' and 'Type' are specified to be of interest indicated by the superscript '[Manufacturer, Type]'. They will be converted into column names of the pivoted output table. '$\perp$' means *empty* entry. The unpivot operator converts column names into data values in a reverse fashion.

The benefits of native support of pivot and unpivot by the query engine are multi-fold [8]. One, we can optimize a query containing pivot/unpivot by moving these operators around the algebra tree. Two, strategies for optimizing the execution of such operators can also be devised.

Despite these execution and optimization strategies, these operators are still potentially costly to evaluate especially when applied to huge volumes of data in data warehousing scenarios. Usage of materialized views to further improve the query performance is a commonly accepted strategy. However, one critical issue, the incremental maintenance of such views remains unsolved, making the refresh cost (i.e., always recomputation) intolerable.

We propose to take an algebraic approach [11] towards the incremental maintenance of views with pivot and unpivot operators. The benefits of tackling the incremental maintenance of such ROLAP views at the algebra level are that first the result is not tied to any particular query language. Second, the correctness of our solution can also easily be shown. In summary, the main contributions of this work are:

- We propose a novel framework for incremental maintenance of views with pivot and unpivot operators. We analyze the basic propagation rules for pivot and show how to obtain an efficient maintenance plan. In order to achieve this, we demonstrate that the transformation of the view query is a necessary step. To our knowledge, this is the first work on efficient maintenance of views with pivot and unpivot, an important class of ROLAP views which are of great interest in practice.

- In order to achieve such query transformation, we propose a *generalized pivot operator* GPIVOT, which not only has more powerful semantics but also can be used to merge multiple pivot operators based on our *combination rules*. These combination rules are useful for both view maintenance and query optimization.

- We propose the *pullup rules* for GPIVOT operators in order to move the GPIVOT operators to the top of the query tree. These rules as well as the corresponding pushdown rules are also useful for both view maintenance and query optimization.

- We propose the *propagation rules* for GPIVOT and its reverse operator GUNPIVOT for the maintenance of ROLAP views. The output of our techniques is a maintenance plan, which can be optimized by a cost-based optimizer using our proposed query transformation rules.

- We demonstrate that these propagation rules may not be efficient when the GPIVOT interacts with other operators in the view query, such as SELECT and GROUPBY. We design special

propagation rules by taking such interactions into consideration to derive a more efficient maintenance plan.

- We also propose the rewriting rules for GUNPIVOT, the reverse operator for GPIVOT.
- We formally prove the correctness of the rewriting and propagation rules for GPIVOT and GUNPIVOT.
- The extensive performance evalutions confirms the effectiveness of our proposed techniques for efficient view maintenance.

Overall, our solution fits nicely into the existing maintenance framework for aggregate views [15, 17]. This makes our maintenance solution easily integrable into these systems. Our query transformation rules serve a dual purpose, namely, both for view maintenance and for query optimization. This paves the way to include the GPIVOT and GUNPIVOT operators into the query engine.

The organization of the rest paper is as follows. Section 2 studies the basic propagation rules for pivot. Section 3 presents the overview of our proposed solution for view maintenance. We define the GPIVOT and GUNPIVOT operators and the combination rules in Section 4. The rewriting rules for GPIVOT and GUNPIVOT are described in Section 5. We propose the propagation rules for GPIVOT and GUNPIVOT and design a novel maintenance framework for applying these rules to obtain an efficient maintenance plan in Section 6. Section 7 presents the results of performance study. Section 8 reviews the related work and Section 9 concludes the paper.

2 Basis on PIVOT and UNPIVOT

2.1 PIVOT and UNPIVOT Operators

We first define the PIVOT and UNPIVOT operators. Assume V is a table with the schema (K, A, B) where K denotes possibly multiple columns and A, B are one column each. The PIVOT operator is defined in Equation (1) [1]. It takes columns A and B as input parameters and $[A_1, ..., A_m]$ as output parameters, where A_i are values of column A. The result of PIVOT converts these column

[1]Except for the NULL handling, the PIVOT and UNPIVOT operators defined in this paper are similar to v2h/h2v and FOLD/UNFOLD operators in [2, 8, 14].

values A_i into column names.

$$\text{PIVOT}^{[A_1,...,A_n]}_{A \text{ on } B}(V) = [\sqsupset\!\!\bowtie\!\!\sqsubset^n_{i=1} \pi_{K,B}(\sigma_{A=`A'_i}(V))] \quad (1)$$

Here $\sqsupset\!\!\bowtie\!\!\sqsubset$ means full outerjoin and is used to find each A_i values for K. Such (K, A_i) value pair may not always exist in table V, hence an outerjoin is required. The missing value will then be denoted as '$\perp$'. An example of PIVOT is depicted in Figure 1. Note that in order to have the results meaningful, the columns K, A together must form the *key* of table V. Then the key for the pivoted output table is K.

Now we assume that the table H has the schema $(K, A_1, ..., A_n)$, where K denotes multiple columns and each A_i is one column. The UNPIVOT is defined in Equation (2) with columns $[A_1, ..., A_n]$ as the input parameters. K usually assumes to be the key of table H in practice although it is not required for the applicability of unpivot.

$$\text{UNPIVOT}_{[A_1,...,A_n]}(H) = [\cup^n_{i=1} \pi_{K,`A'_i,A_i}(\sigma_{A_i \neq \perp} H)] \quad (2)$$

One example of UNPIVOT is given in Figure 1, where the columns names 'Manufacturer' and 'Type' are converted into data values.

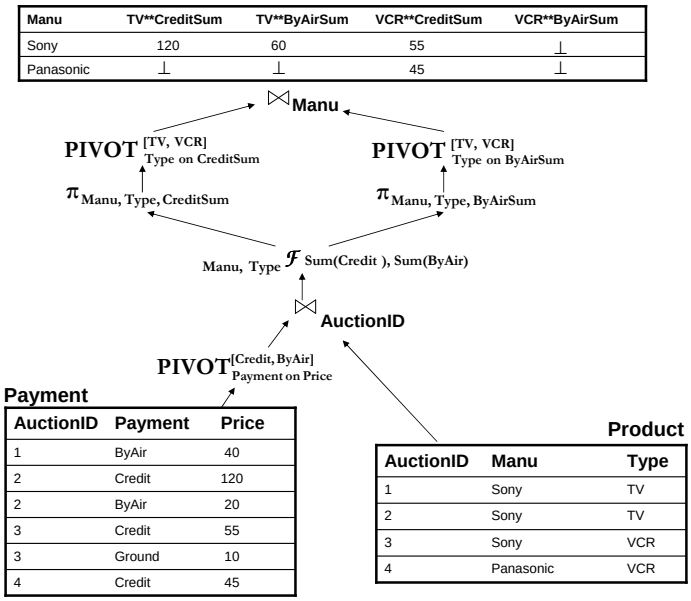


Figure 2: A Sample ROLAP View

Figure 2 depicts an example view composed of relational algebra and pivot operators. In this example, the vertical table Payment stores the different types of payment information. It is first pivoted to output the prices of type Credit and ByAir. Then an equi-join is performed with the Product table. After that, we compute the total Credit and ByAir payments for each manufacturer and type. For this, we use the notation $\mathcal{F}$ [9] to specify the group-by columns (Manu, Type) and the aggregation list (sum(Credit), sum(ByAir)). The aggregate results are pivoted again in order to provide a crosstab view of the summary data. We will show in this paper a strategy for generating an efficient maintenance plan for complex ROLAP views such as this one.

2.2 Basic Propagation Rules

As a first step to study the incremental maintenance of views with pivot and unpivot operators, Figure 3 depicts some rules for how to propagate changes through the pivot operator [2]. Assume some data were inserted into the *ItemInfo* table. The first rules, which we call *insert/delete propagation rules*, propagate these changes through the pivot operator as one positive delta (insert) and one negative delta (delete) to the original pivoted result. Here the negative delta are the old output tuples affected by the source inserts. The positive delta are the new output tuples introduced by the source inserts. The second rules, which we call *update propagation rules*, first perform a left outer-join between the pivoted delta, PIVOT(ΔI), and the original result, PIVOT(I). Then from the join result, the unmatched tuples will be *inserted* and the matched tuples will be *updated*.

2.3 Discussion of Propagation Rules

We note that both propagation rules in Figure 3 access the original pivoted result, PIVOT(I). If the pivot is an intermediate operator in the query plan, then re-evaluating this intermediate result PIVOT(I) or even just partially re-evaluating it by predicate pushdown could still be fairly expensive. Moreover, the update propagation rules are not applicable in this case unless the intermediate results are materialized. This may be prohibitively expensive. In comparison, if the pivot is the *last* operator in the query plan, then PIVOT(I) represents the materialized view itself. In this case, we can avoid the re-evaluation of PIVOT(I). Instead we could perform a join between the delta and the materialized view itself. Hence, these propagation rules can be more

[2]The propagation rules for unpivot are relatively simple and will be discussed in Section 6, together with the detailed formalism.

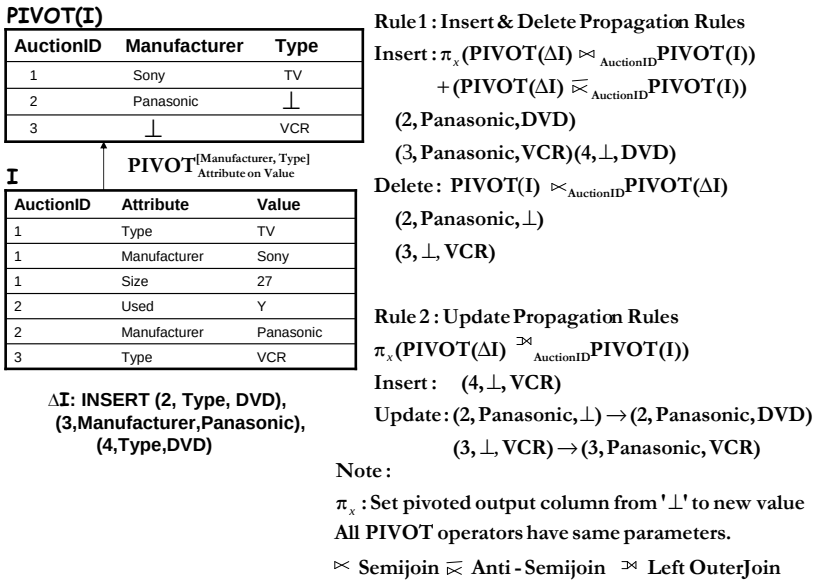


Figure 3: Propagating Changes through PIVOT

efficiently applied when the pivot is the *last* operator in the query plan.

Second, even when the pivot is the *last* operator in the query plan, there are still some differences between these two types of propagation rules. For the insert/delete rules, the tuples to be deleted might be re-inserted again with just a few column changes. In Figure 3, (2,Panasonic,⊥) and (3,⊥,VCR) are deleted and re-inserted as (2,Panasonic,DVD) and (3,Panasonic,VCR). In comparison, the update rules can make in-place changes of these rows by a SQL update statement. Such deletion and then re-insertion generally introduces more CPU and I/O cost than the update approach.

Based on the observations above, we conclude that in order to derive an efficient maintenance plan, (1) the pivot should be the *last* operator in the query plan and (2) the update propagation rules are preferred to the insert/delete propagation rules. In fact, similar heuristics have also been employed in prior view maintenance work. For example, the propagation rules for GROUPBY can also use either insert/delete or update operations [18]. The update propagation rules are preferable as suggested in [18] and in fact are the ones incorporated into many commercial systems [3, 15]. The update propagation rules also require the GROUPBY to be the last operator in the query plan. These heuristics for GROUPBY are the same as ours for pivot.

3 Our Overall Approach

In this work, we propose a systematic way to efficiently and incrementally maintain both aggregate or non-aggregate materialized views containing pivot and unpivot operators.

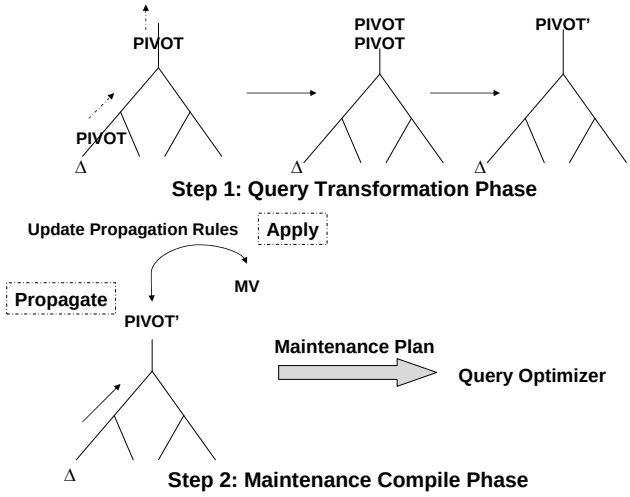


Figure 4: Solution Overview

There might be multiple pivot operators in the query algebra tree (see the example query in Figure 2). Except for the top pivot in the tree, other intermediate pivot operators may not propagate changes efficiently. Hence, as shown in Figure 4, the first step of our solution is to pull the pivot operators up to the **top** of the algebra tree if possible by query rewriting rules and combine them into a **single extended** pivot operator, which we call *Generalized PIVOT (GPIVOT)*.

The second step is to construct the maintenance plan based on the transformed query algebra tree. The resulting maintenance plan contains two phases. The *propagate phase* propagates the deltas through each operator to the top of the tree to compute the *final delta*. Here, we can apply the existing propagation rules for relational operators [11]. The *apply phase* applies the update propagation rules for this extended pivot operator. Together, this two-phase processing of *propagate* and *apply* phases fit nicely into the traditional aggregate view maintenance framework [15, 17]. This makes our solution easily integrable into existing systems. Note that the result of this compile phase is a maintenance query plan. Thus it is optimizable by a query optimizer. For example, we now may want to push down or split the top pivot operator for execution. Such decision can be made by a cost-based optimizer.

Note that for those intermediate pivot operators that cannot be pulled up, we have to apply the

insert/delete propagation rules in order to be able to propagate the changes through them. The resulting maintenance plan may still outperform the full recomputation approach. This also makes our solution complete in the sense that it is capable of maintaining any ROLAP views.

4 Combining Multiple PIVOTs

4.1 GPIVOT and GUNPIVOT: Generalized PIVOT and UNPIVOT

In this section, we will first describe how to combine multiple pivot operators. We will show that the resulting pivot operator, which we call *Generalized PIVOT (GPIVOT)*, is a natural extension of the simple pivot in Equation (1) with more powerful semantics. Its definition is in Equation (3). Here we assume that the table V has schema $(K, A_1, A_2, ..., A_m, B_1, B_2, ..., B_n)$, where K denotes possibly multiple columns and A_i, B_j denote one column each [3]. $(K, A_1, ..., A_m)$ must form a key for pivot applicability. Similar to the simple pivot operator, the input parameters for GPIVOT are the columns $[A_1, ..., A_m]$ and $[B_1, ..., B_n]$. The output parameters for GPIVOT are $[(a_1^1, ..., a_m^1), ..., (a_1^p, ..., a_m^p)]$, which are values of columns $(A_1, ..., A_m)$.

$$\text{GPIVOT}^{[(a_1^1,...,a_m^1),...,(a_1^p,...,a_m^p)]}_{[A_1,...,A_m] \text{ on } [B_1,...,B_n]}(V) = [\exists\!\!\!\!\sqsubset_{i=1}^{p} \pi_{K,B_1,...,B_n}(\sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)}(V))]\ ^{4} \quad (3)$$

An example of GPIVOT is shown in Figure 5. Here '$\{Sony, Panasonic\} \times \{TV, VCR\}$' means that any combination of the given manufacturer and type values will be output. Unlike the simple pivot, the GPIVOT output column names now need special treatment. We use the simple protocol of naming the pivoted output columns as '$a_1^i * * a_2^i * * ... * * a_m^i * * B_j$' [5]. Note that the GPIVOT operator is able to pivot *multiple* measurements based on *multiple* dimensions, a rather common and highly useful operation [7] for multi-dimensional databases.

The GUNPIVOT operator is designed to decode the column names in the reverse way (Equation (4)). Here we assume the table H has schema $(K, a_1^1 * * ... a_m^1 * * B_1, ..., a_1^1 * * ... a_m^1 * * B_n, ..., a_1^p * * ... a_m^p * * B_1, ..., a_1^p * * ... a_m^p * * B_n)$, where K can be multiple columns and each $a_1^i * * ... a_m^i * * B_j$ is one column. One example is in Figure 5.

[3] This input table schema will be used in the rest paper for GPIVOT.

[4] For simplicity, we assume GPIVOT will output all $(B_1, ..., B_n)$ for each $(a_1^i, ..., a_m^i)$. We can add an additional projection to remove unwanted columns. Such projection can be pushed into the GPIVOT execution for optimization.

[5] Alternatively, we can use a separate table to store such column name information.

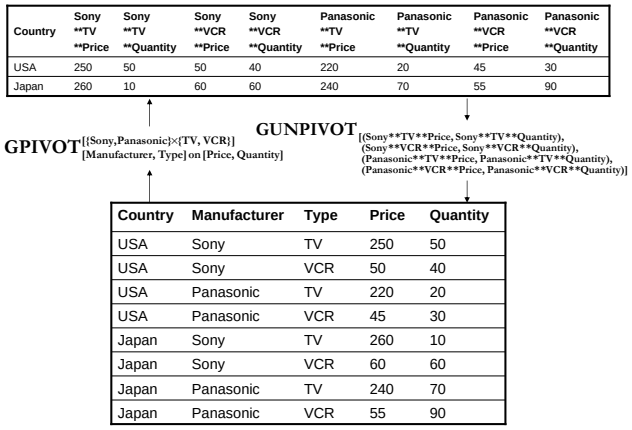


Figure 5: Example for GPIVOT and GUNPIVOT

$$
\begin{aligned}
&\text{GUNPIVOT}_{[(a_1^1 **...a_m^1 **B_1,...,a_1^1 **...a_m^1 **B_n),} \\
&\qquad\qquad ... \\
&\qquad\qquad (a_1^p **...a_m^p **B_1,...,a_1^p **...a_m^p **B_n)]}(H) \\
&= [\cup_{i=1}^{p} \pi_{K,`a_1^i{}',...,`a_m^i{}',a_1^i **...a_m^i **B_1,...,a_1^i **...a_m^i **B_n} \\
&\qquad\qquad (\sigma_{any\ a_1^i **...a_m^i **B_j\ \neq \perp} H)]
\end{aligned} \tag{4}
$$

Since PIVOT is a special case of GPIVOT and UNPIVOT a special case of GUNPIVOT, in the rest of this paper we will only consider GPIVOT and GUNPIVOT. The results obviously apply to PIVOT and UNPIVOT as well.

4.2 Combining Multiple GPIVOTs

4.2.1 Multicolumn PIVOT

The first combination rule for GPIVOT is called *multicolumn pivot*. Take the view in Figure 2 for example, both the total sum of Credit and the total sum of ByAir are pivoted by first pivoting each of them individually and then joining the respective results. We propose to combine these two pivot operators into one that simply pivots both 'CreditSum' and 'ByAirSum' columns by 'Type' column as $GPIVOT^{[TV,VCR]}_{Type\ on\ [CreditSum,ByAirSum]}$. This combination rule for GPIVOT is formally defined in Equation (5), assuming the same schema of table V.

$$\text{GPIVOT}^{[\{(a_1^i,...,a_m^i)\}]}_{[A_1,...,A_m]\ on\ [B_1,...,B_n]}(V) =$$
$$\text{GPIVOT}^{[\{(a_1^i,...,a_m^i)\}]}_{[A_1,...,A_m]\ on\ [B_1,...,B_j]}(\pi_{K,A_1,...,A_m,B_1,...,B_j}V) \bowtie_K$$
$$\text{GPIVOT}^{[\{(a_1^i,...,a_m^i)\}]}_{[A_1,...,A_m]\ on\ [B_{j+1},...,B_n]}(\pi_{K,A_1,...,A_m,B_{j+1},...,B_n}V) \quad (5)$$

Proof for Equation (5): By GPIVOT definition in Equation (3), we have

$$\text{GPIVOT}^{[\{(a_1^i,...,a_m^i)\}]}_{[A_1,...,A_m]\ on\ [B_1,...,B_j]}(\pi_{K,A_1,...,A_m,B_1,...,B_j}V) =$$
$$\rtimes\!\!\sqsubset^p_{i=1}\pi_{K,B_1,...,B_j}(\sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)}(\pi_{K,A_1,...,A_m,B_1,...,B_j}V))$$
$$\text{GPIVOT}^{[\{(a_1^i,...,a_m^i)\}]}_{[A_1,...,A_m]\ on\ [B_{j+1},...,B_n]}(\pi_{K,A_1,...,A_m,B_{j+1},...,B_n}V) =$$
$$\rtimes\!\!\sqsubset^p_{i=1}\pi_{K,B_{j+1},...,B_n}(\sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)}(\pi_{K,A_1,...,A_m,B_{j+1},...,B_n}V))$$
$$\text{GPIVOT}^{[\{(a_1^i,...,a_m^i)\}]}_{[A_1,...,A_m]\ on\ [B_1,...,B_n]}(\pi_{K,A_1,...,A_m,B_1,...,B_n}V) =$$
$$\rtimes\!\!\sqsubset^p_{i=1}\pi_{K,B_1,...,B_n}(\sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)}(\pi_{K,A_1,...,A_m,B_1,...,B_n}V))$$

In other words, we need to prove the following:

$$[\rtimes\!\!\sqsubset^p_{i=1}\pi_{K,B_1,...,B_j}(\sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)}(\pi_{K,A_1,...,A_m,B_1,...,B_j}V))]$$
$$\bowtie_K\ [\rtimes\!\!\sqsubset^p_{i=1}\pi_{K,B_{j+1},...,B_n}(\sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)}(\pi_{K,A_1,...,A_m,B_{j+1},...,B_n}V))]$$
$$= [\rtimes\!\!\sqsubset^p_{i=1}\pi_{K,B_1,...,B_n}(\sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)}(\pi_{K,A_1,...,A_m,B_1,...,B_n}V))] \quad (5.1)$$

(1) Since both sides of Equation (5.1) have a key K in their output, we first show that both sides output the same set of key values. The left side of Equation (5.1) outputs key set: $\delta_K(\sigma_{(A_1,...,A_m)=(a_1^1,...,a_m^1)\vee...\vee(A_1,...,A_m)=(a_1^p,...,a_m^p)}(V))$, where δ means project under set semantics (i.e., select distinct). The right side of Equation (5.1) output key set: $\delta_K(\sigma_{(A_1,...,A_m)=(a_1^1,...,a_m^1)\vee...\vee(A_1,...,A_m)=(a_1^p,...,a_m^p)}(V)) \bowtie \delta_K(\sigma_{(A_1,...,A_m)=(a_1^1,...,a_m^1)\vee...\vee(A_1,...,A_m)=(a_1^p,...,a_m^p)}(V)) = \delta_K(\sigma_{(A_1,...,A_m)=(a_1^1,...,a_m^1)\vee...\vee(A_1,...,A_m)=(a_1^p,...,a_m^p)}(V))$. Hence, both sides generate the same set of key values.

(2) Next we show that for a given value k_1, both sides of Equation (5.1) yields the same output tuple. Assume for a given K value k_1, a set of rows $\{r_1, ..., r_p\}$ are defined as: $r_i = \pi_{K,B_1,...,B_n}(\sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)\ AND\ K=k_1}(V))$. Note that there must be *at most* one such tuple in V that satisfies the above condition, because $(K, A_1, ..., A_m)$ forms the key of table V. If table V does not contain any tuple that satisfies the above condition, then we let $r_i = (k_1, \perp, ..., \perp)$. Based on this definition, for a given key value k_1, the output of the right side of Equation (5.1) is $\bowtie \{r_i\}$. While the output of the left side of Equation (5.1) is $[\bowtie \{\pi_{K,B_1,...,B_j}(r_i)\}] \bowtie [\bowtie \{\pi_{K,B_{j+1},...,B_n}(r_i)\}] = \bowtie \{r_i\}$ since $\pi_{K,B_1,...,B_j}(r_i) \bowtie \pi_{K,B_{j+1},...,B_n}(r_i) = r_i$. Hence for a given value k_1, both sides of Equation (5.1)

yields the same output tuple.

By (1) and (2), we thus reach the conclusion that Equation (5) always holds. ∎

4.2.2 PIVOT Composition

The second rule is to combine two adjacent GPIVOT operators, when *all* the pivoted output columns of the first pivot are the input parameters of the second pivot. This may occur when the user wants to pivot the measurements by more than one dimension. One simple example is shown in Figure 6. On the left side of the figure, the second pivot takes *all* the output columns of the first pivot as the columns to be further pivoted on. These two operators can also be combined into one operator by combining their parameters as shown on the right side of the figure.

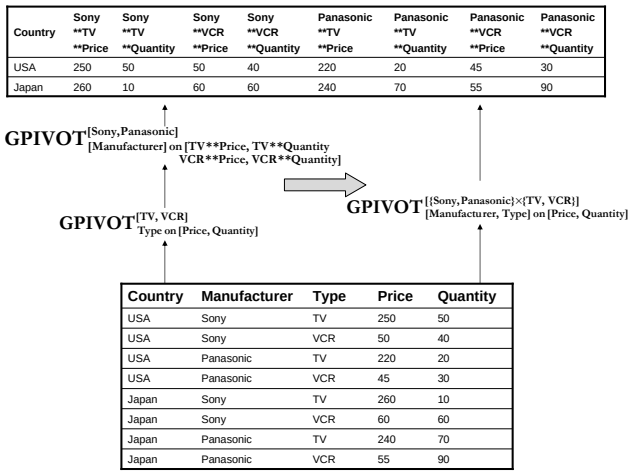


Figure 6: Composition of GPIVOT

Equation (6) formally defines this rule. We assume the same table V and $\{(a_1^i, ..., a_l^i)\}$ as the output values of $(A_1, ..., A_l)$ and $\{(a_{l+1}^i, ..., a_m^i)\}$ as the output values of $(A_{l+1}, ..., A_m)$. Here the second GPIVOT takes *all* the output columns of the first GPIVOT, i.e., '$\{(a_{l+1}^i, ..., a_m^i)\} \times \{B_j\}$', as the input parameters.

$$\begin{aligned}\text{GPIVOT}^{[\{(a_1^i,...,a_l^i)\}\times\{(a_{l+1}^i,...,a_m^i)\}]}_{[A_1,...,A_l,A_{l+1},...,A_m]\ on\ [B_1,...,B_n]}(V) = \\ \text{GPIVOT}^{[\{(a_1^i,...,a_l^i)\}]}_{[A_1,...,A_l]\ on\ [\{(a_{l+1}^i,...,a_m^i)\}\times\{B_j\}]}(\text{GPIVOT}^{[\{(a_{l+1}^i,...,a_m^i)\}]}_{[A_{l+1},...,A_m]\ on\ [B_1,...,B_n]}(V))\end{aligned} \quad (6)$$

Proof for Equation (6): We assume the output values for $(A_1, ..., A_l)$ are $\{(a_1^1, ..., a_l^1), ..., (a_1^{p_1}, ..., a_l^{p_1})\}$ and the output values for $(A_{l+1}, ..., A_n)$ are $\{(a_{l+1}^1, ..., a_n^1), ..., (a_{l+1}^{p_2}, ..., a_n^{p_2})\}$.

(1) Since both sides of Equation (6) have a key K in their output, we first show that both sides output the same set of key values. The left side of Equation (6) outputs key set: $\delta_K(\sigma_{[(A_1,...,A_l)=(a_1^1,...,a_l^1) \vee ... \vee (A_1,...,A_l)=(a_1^{p_1},...,a_l^{p_1})] \wedge [(A_{l+1},...,A_m)=(a_{l+1}^1,...,a_m^1) \vee ... \vee (A_{l+1},...,A_m)=(a_{l+1}^{p_2},...,a_m^{p_2})]}(V))$, where δ means project under set semantics (i.e., select distinct). The right side of Equation (6) output key set: $\delta_K(\sigma_{(A_1,...,A_l)=(a_1^1,...,a_l^1) \vee ... \vee (A_1,...,A_l)=(a_1^{p_1},...,a_l^{p_1})}(\delta_{K,A_1,...,A_l}(\sigma_{(A_{l+1},...,A_m)=(a_{l+1}^1,...,a_m^1) \vee ... \vee (A_{l+1},...,A_m)=(a_{l+1}^{p_2},...,a_m^{p_2})}(V))$. By pushing down the selection, we get $\delta_K(\sigma_{[(A_1,...,A_l)=(a_1^1,...,a_l^1) \vee ... \vee (A_1,...,A_l)=(a_1^{p_1},...,a_l^{p_1})] \wedge [(A_{l+1},...,A_m)=(a_{l+1}^1,...,a_m^1) \vee ... \vee (A_{l+1},...,A_m) =(a_{l+1}^{p_2},...,a_m^{p_2})]}(V))$. Hence, both sides generate the same set of key values.

(2) Next we show that for a given value k_1, both sides of Equation (6) yields the same output tuple. A set of rows $\{r_{11}, ..., r_{p_1 p_2}\}$ are defined as: $r_{ij} = \pi_{K,B_1,...,B_n}(\sigma_{(A_1,...,A_l)=(a_1^i,...,a_l^i) \ AND \ (A_{l+1},...,A_n) =(a_{l+1}^j,...,a_n^j) \ AND \ K=k_1}(V))$, where $i = 1..p_1$ and $j = 1..p_2$. If table V does not contain any tuple that satisfies the above condition, then we let $r_{ij} = (k_1, \perp, ..., \perp)$. Based on this definition, for a given key value k_1, the output of the left side of Equation (6) is $\bowtie \{r_{ij}\}$. For the right side of Equation (6), we have:

$$\text{GPIVOT}^{[\{(a_1^i,...,a_l^i)\}]}_{[A_1,...,A_l] \ on \ [\{(a_{l+1}^i,...,a_m^i)\} \times \{B_j\}]}(\text{GPIVOT}^{[\{(a_{l+1}^i,...,a_m^i)\}]}_{[A_{l+1},...,A_m] \ on \ [B_1,...,B_n]}(V)) =$$
$$\exists\!\!\bowtie\!\!\sqsubset_{i=1}^{p_1} \pi_{K,C_1,...,C_{p_2 \times n}}(\sigma_{(A_1,...,A_l)=(a_1^i,...,a_l^i)}[\exists\!\!\bowtie\!\!\sqsubset_{j=1}^{p_2} \pi_{K,A_1,...,A_l,B_1,...,B_n}(\sigma_{(A_{l+1},...,A_n)=(a_{l+1}^j,...,a_n^j)}(V))]).$$

Here $C_1, ..., C_{p_2 \times n}$ are the pivoted output columns for the first GPIVOT. By pushing down the first selection, we get:

$$\exists\!\!\bowtie\!\!\sqsubset_{i=1}^{p_1} \pi_{K,C_1,...,C_{p_2 \times n}}([\exists\!\!\bowtie\!\!\sqsubset_{j=1}^{p_2} \pi_{K,A_1,...,A_l,B_1,...,B_n}(\sigma_{(A_{l+1},...,A_n)=(a_{l+1}^j,...,a_n^j) \ AND \ (A_1,...,A_l)=(a_1^i,...,a_l^i)}(V))]).$$

Hence, for a given K value k_1, the output tuple is $\bowtie_{i=1}^{p_1} [\bowtie_{j=1}^{p_2} r_{ij}] = \bowtie \{r_{ij}\}$. Thus both sides of Equation (6) yield the same output tuple for any value of K.

By (1) and (2), we thus reach the conclusion that Equation (6) always holds. ∎

4.2.3 Completeness of Combination Rules

We now study the combination rules for any two adjacent GPIVOT operators in general. In particular, we consider the possible parameters for the second GPIVOT.

Assume the final output has schema $(K, \{A_i\})$, where $\{A_i\}$ are the pivoted output columns. We

note that if the two adjacent GPIVOT operators can be combined into one, then the following three observations must hold: (1) The *key* K in the final output table must be *part* of the key in the original table based on the GPIVOT definition in Section 4.1; and (2) the pivoted output column names must have the same structure as the GPIVOT definition in Section 4.1, i.e., 'a_1^i**..**a_m^i**B_j'; and (3) the data values in the original table are *not* lost, i.e., they either still remain as data values or become column names in the final pivoted output.

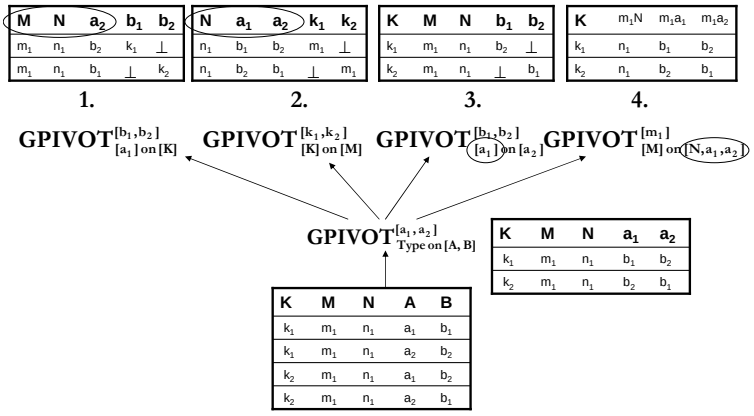


Figure 7: Example for Combining Two Adjacent GPIVOTs

Figure 7 depicts some examples for two adjacent GPIVOTs. In the first case, the pivoted output column a_1 is used to pivot the column K. The key in the output is (M, N, a_2), which cannot be part of the key in the original table since a_2 is not even a column there. More generally, the pivoted output columns for the first GPIVOT must *all* be used for the second GPIVOT. Otherwise, some of the pivoted output columns will form the key for the second GPIVOT. This makes it impossible for the combination of these two GPIVOTs. The reason is that the pivoted output columns are only data values in the original table and cannot form a key, which violates observation (1) mentioned above. We can thus quickly determine that the second case in Figure 7 is not possible for combination either, since the pivoted output columns of the first GPIVOT appear as part of the key in the final output.

Now assume the second GPIVOT pivots columns $(X_1, ..., X_m)$ by columns $(Y_1, ..., Y_n)$. We know that the pivoted output columns of the first GPIVOT must *all* be contained in $\{X_i\}$ and $\{Y_j\}$. If $\{X_i\}$ contains any of the pivoted output column as the third case in Figure 7, then their columns names (which are part of the original data, such as a_1 in this example) will be *lost* in the final output, which violates observation (3). We thus reach the conclusion that $\{Y_j\}$ must contain all

pivoted output columns of the first GPIVOT.

When $\{Y_j\}$ only contains all the pivoted output columns of the first GPIVOT, the two GPIVOTs can be combined as in Equation (6). The last possibility is when $\{Y_j\}$ contains extra columns besides all the previous pivoted output columns, as the fourth case in Figure 7. In this case, the two GPIVOTs cannot be combined either, since the pivoted output column names cannot have the same structure, which violates observation (2).

As a final remark, we can apply the combination rules developed in this section in the query graph in order to reduce the number of pivots. The combination rule in Section 4.2.1 increases the columns (measurements) to be pivoted on, while the rule in Section 4.2.2 increases the columns (dimensions) to be pivoted by. It is important to note that these combination rules not only help for incremental view maintenance but are also beneficial for optimization of queries, even those with only simple pivots.

4.3 Splitting GPIVOT

The split rules for GPIVOT can easily be derived based on the combination rules. For example, the Equation (5) and (6) can be used to split the GPIVOT defined on the left side of the equation to the expression on the right side.

There are also some interesting splitting rule for parallel processing of GPIVOT (similar to the parallel processing of simple pivot in [8]). That is, we compute the GPIVOT sub-results for each node and then combine them together to generate the final output. This is very similar to the standard local/global aggregation for parallel aggregate processing. The GPIVOT sub-results at each node can be combined using the propagation rules under *insert* case in Section 6.1, as we will elaborate later.

5 Rewriting Rules for GPIVOT and GUNPIVOT

As motivated in Section 2.3, the efficient view maintenance requires us to pull the GPIVOT operators up the view query tree in order to apply the update propagation rules. In this section, we will study the pullup as well as the pushdown rules for GPIVOT. We will also present the rewriting rules for GUNPIVOT.

5.1 Pullup Rules for GPIVOT

Figure 8 describes one general principle for the GPIVOT pullup rules. Assume there is one operator "Op" above the GPIVOT. Since the output of the pulled up GPIVOT' must contain a key due to the nature of pivot operators, a prerequisite for the pullup applicability is that the operator "Op" must also preserve a key.

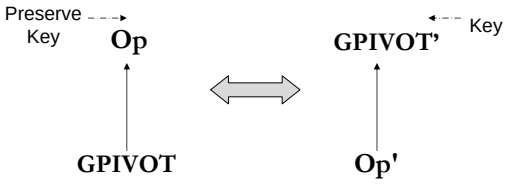


Figure 8: Prerequisite for GPIVOT Pullup: Key Preservation

5.1.1 Pullup GPIVOT through SELECT

While selection pushdown is trivial for most relational operators, it is complex for GPIVOT. If the selection condition is defined on non-pivoted output columns, then we can push it down without any changes such as the condition $\sigma_{Country='USA'}$ in Figure 9.

However, if the selection condition involves pivoted output columns and is null-intolerant (i.e., is false when NULL), then pushing down the selection results in multiple self-joins. For instance, in Figure 9, in order to push down the condition $\sigma_{Sony**TV**Price>200}$, we first find the country with its Sony TV price larger than 200 and then do a join with the original table to find other information about these countries. That is: $\pi_{Country}(\sigma_{Manu=Sony\ \wedge\ Type=TV\ \wedge\ Price>200}(V)) \bowtie V$. More self-joins are required if more pivoted output columns are involved. Formally, assume a selection predicate over two pivoted output columns as: $\sigma_{a_1^{i_1}**\ldots a_m^{i_1}**B_{l_1}\ op\ a_1^{i_2}**\ldots a_m^{i_2}**B_{l_2}}(GPIVOT^{[\{(a_1^i,\ldots,a_m^i)\}]}_{[A_1,\ldots,A_m]\ on\ [B_1,\ldots,B_n]}(V))$. Here 'op' is any comparison operator. This predicate can be pushed down based on the rule below.

$$\sigma_{a_1^{i_1}**\ldots a_m^{i_1}**B_{l_1}\ op\ a_1^{i_2}**\ldots a_m^{i_2}**B_{l_2}}(GPIVOT^{[\{(a_1^i,\ldots,a_m^i)\}]}_{[A_1,\ldots,A_m]\ on\ [B_1,\ldots,B_n]}(V)) =$$
$$GPIVOT^{[\{(a_1^i,\ldots,a_m^i)\}]}_{[A_1,\ldots,A_m]\ on\ [B_1,\ldots,B_n]}($$
$$\pi_K[\sigma_{(A_1,\ldots,A_m)=(a_1^{i_1},\ldots,a_m^{i_1})}(V) \bowtie_{K^1=K^2\ \wedge\ B_{l_1}^1\ op\ B_{l_2}^2}\ \sigma_{(A_1,\ldots,A_m)=(a_1^{i_2},\ldots,a_m^{i_2})}(V)] \bowtie V) \qquad (7)$$

Proof for Equation (7):

$\sigma_{a_1^{i_1}**...a_m^{i_1}**B_{l_1}\ op\ a_1^{i_2}**...a_m^{i_2}**B_{l_2}}(GPIVOT^{[\{(a_1^i,...,a_m^i)\}]}_{[A_1,...,A_m]\ on\ [B_1,...,B_n]}(V))$.

$= (\pi_K[\sigma_{(A_1,...,A_m)=(a_1^{i_1},...,a_m^{i_1})}(V) \bowtie_{K^1=K^2\ \wedge\ B^1_{l_1}\ op\ B^2_{l_2}} \sigma_{(A_1,...,A_m)=(a_1^{i_2},...,a_m^{i_2})}(V)]) \bowtie$
$(GPIVOT^{[\{(a_1^i,...,a_m^i)\}]}_{[A_1,...,A_m]\ on\ [B_1,...,B_n]}(V))$.

By pushing down the join condition (since it is on key column), we have:

$= GPIVOT^{[\{(a_1^i,...,a_m^i)\}]}_{[A_1,...,A_m]\ on\ [B_1,...,B_n]}($
$\pi_K[\sigma_{(A_1,...,A_m)=(a_1^{i_1},...,a_m^{i_1})}(V) \bowtie_{K^1=K^2\ \wedge\ B^1_{l_1}\ op\ B^2_{l_2}} \sigma_{(A_1,...,A_m)=(a_1^{i_2},...,a_m^{i_2})}(V)] \bowtie V)$. ∎

Note that when $i_1 = i_2$, i.e., the columns have the same prefix, then the first join can be avoided as $GPIVOT(\pi_K[\sigma_{(A_1...A_m)=(a_1^{i_1}...a_m^{i_1})\ \wedge\ B_{l_1}\ op\ B_{l_2}}(V)] \bowtie V)$.

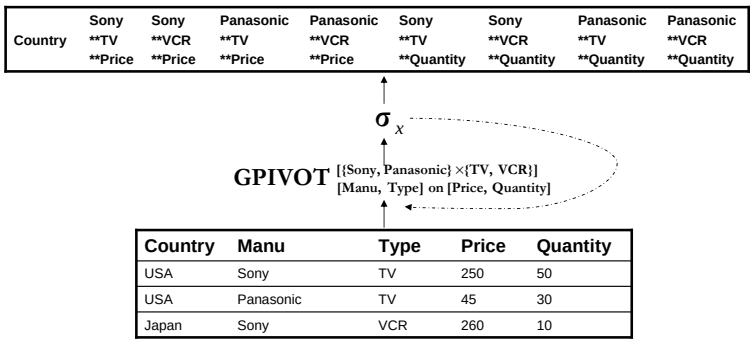


Figure 9: Pullup through SELECT

The above rules can be easily extended to handle predicates with even more pivoted output columns and complex conjunctive or disjunctive conditions. To handle more pivoted output columns, we need to perform more self-joins. Each join is to find one pivoted output column. The final join result provides the key values that satisfy the condition. Conjunctive and disjunctive conditions can be achieved by unioning or intersecting these key values.

However, the benefit of pulling GPIVOT up is likely offset by such multiple self-joins since propagating changes through multiple self-join expressions is non-trivial, i.e., generating multiple join terms [12]. One alternative to address this potential performance problem is that for those conditions that result in multiple self-joins if pushed down, we pull *both* the SELECT and GPIVOT up the query tree and design special update propagation rules. We will describe this technique in Section 6.3.2.

5.1.2 Pullup GPIVOT through PROJECT

In this work, we consider negative project, i.e., removal of columns. The project operator that drops the non-pivoted output columns can be pushed down unless this project violates the prerequisite of the key preservation. For example, the drop of the 'Country' column above the GPIVOT in Figure 9 cannot be pushed down since the output no longer contains a key. We have to use the insert/delete propagation rules for this pivot. The project operator that drops the pivoted output columns need careful treatment. E.g., $\pi_{\neg VCR}(GPIVOT^{[TV,VCR]}_{Type\ on\ Price}) \neq GPIVOT^{[TV]}_{Type\ on\ Price}$. The reason is that the left part of equation will output TV with $\perp$ while the right part not. We may need to use the insert/delete propagation rules for this GPIVOT. This in fact also suggest not to remove the pivoted output columns in the materialized view definition, which also increases the opportunities to utilize this view to answer queries.

5.1.3 Pullup GPIVOT through JOIN

Guided by the same principle, the join result should also preserve a key in order to pull up the GPIVOT. In general, both operands having a key must hold. While this requirement seems restrictive, however in data warehousing senarios the majority of the joins are between the fact tables and the dimension tables on their keys and foreign keys, respectively. Thus they fall into this category.

The rules of pulling up GPIVOT through the JOIN operator are similar to those for the SELECT operator. If the join condition is not on the pivoted output columns, then we can pull up the GPIVOT without change. An example is shown in Figure 10. That is, since the AuctionID is the non-pivoted output column, the GPIVOT can be pulled up (the pullup through GROUPBY in the figure will be explained later). When the join condition involves pivoted output columns, pushing down the join operator results in multiple self-joins. This again is similar to the situation for the SELECT operator.

Assume a join is $GPIVOT(A) \bowtie_{\sigma_1\ \wedge\ \sigma_2} B$, where σ_1 is the join condition involving only non-pivoted output columns of $GPIVOT(A)$ and σ_2 is the join condition involving the pivoted output columns. We can pull up the GPIVOT as $\sigma_2(GPIVOT(A \bowtie \sigma_1 B))$. Then we can pull both σ_2 and the GPIVOT up the algebra tree as in Section 5.1. Note that if the join condition σ_1 is empty, then the pullup of the GPIVOT results in a Cartesian product of the underlying tables. If the join condition is $\sigma_1\ \vee\ \sigma_2$, then we cannot split these two conditions. For those cases, we instead choose

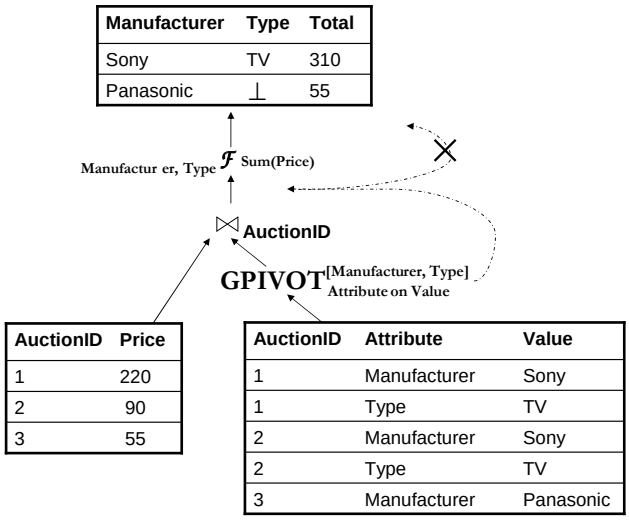


Figure 10: Pullup through Join and GROUPBY

the insert/delete propagation rules for this GPIVOT.

5.1.4 Pullup GPIVOT through GROUPBY

The applicability of pulling up GPIVOT through GROUPBY depends on how the GROUPBY uses the pivoted output columns. In particular, if the pivoted output columns are group-by columns, then we cannot pullup the GPIVOT. If the pivoted output colomns are used to compute the aggregate, then we can pullup the GPIVOT.

Figure 10 depicts an example when we cannot pullup the GPIVOT. While the GPIVOT in the figure is successfully pulled upon through the join operator, it cannot be further pulled up through the GROUPBY denoted by $\mathcal{F}$ in the figure. The reason is that the group-by columns, e.g., 'Sony' and 'TV', are two values originating from the same column 'Value'. There is no good way to achieve such *multi-value* grouping on a single column.

The lower pivot in Figure 2 is an example that can be pulled up through the GROUPBY. That is, the aggregate functions are over the pivoted output columns 'Credit' and 'ByAir'. In this case, we can pull up the GPIVOT by modifying both the GROUPBY and GPIVOT's parameters, i.e., by adding the pivot parameter 'Payment' into the group-by columns and by aggregating over the 'Price' column. The rewritten GPIVOT will take the aggregate results as input parameters. The lower part of the query tree up to the GROUPBY in Figure 2 can thus be rewritten as in Figure 11.

Formally, assume the same table V and the same parameters for the GPIVOT. The GROUPBY

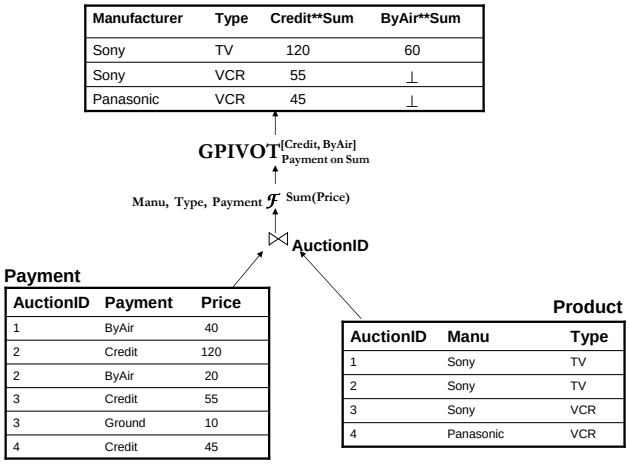


Figure 11: Pullup through GROUPBY

operator $\mathcal{F}$ takes $K' \subseteq K$ as group-by columns and computes any aggregate function f over the pivoted output columns $\{(a_1^i, ..., a_m^i)\} \times \{B_j\}$. This pull up rule is given in Equation (8).

$$
\begin{aligned}
&{}_{K'}\mathcal{F}_{f(\{(a_1^i,...,a_m^i)\}\times\{B_j\})}(GPIVOT^{[\{(a_1^i,...,a_m^i)\}]}_{[A_1...A_m]\ on\ [B_1...B_n]}(V)) = \\
&\qquad GPIVOT^{[\{(a_1^i,...,a_m^i)\}]}_{[A_1,...,A_m]\ on\ [f(B_1),...,f(B_n)]}({}_{K',A_1,...,A_m}\mathcal{F}_{f(B_1),...,f(B_n)}(V))\ ^{6}
\end{aligned}
\tag{8}
$$

Proof for Equation (8): (1) Since both sides of Equation (8) have a key K' in their output, we first show that both sides output the same set of key values. The left side of Equation (8) outputs key set: $\delta_{K'}(\delta_K(\sigma_{(A_1,...,A_m)=(a_1^1,...,a_m^1)\vee...\vee(A_1,...,A_m)=(a_1^p,...,a_m^p)}(V)) = \delta_{K'}(\sigma_{(A_1,...,A_m)=(a_1^1,...,a_m^1)\vee...\vee(A_1,...,A_m)=(a_1^p,...,a_m^p)}(V))$, where δ means project under set semantics (i.e., select distinct). The right side of Equation (8) output key set: $\delta_{K'}(\sigma_{(A_1,...,A_m)=(a_1^1,...,a_m^1)\ \vee...\vee(A_1,...,A_m)=(a_1^p,...,a_m^p)}(\delta_{K',A_1,...,A_m}(V)) = \delta_{K'}(\sigma_{(A_1,...,A_m)=(a_1^1,...,a_m^1)\vee...\vee\ (A_1,...,A_m)\ =(a_1^p,...,a_m^p)}(V))$. Hence, both sides generate the same set of key values.

(2) Next we show that for any key value k_1' for K', both sides of Equation (8) generate the same row. We further assume that there are K values $k_1, ..., k_p$ in V that contains k_1' and $k_l \in \delta_K(\sigma_{(A_1,...,A_m)=(a_1^1,...,a_m^1)\vee...\vee(A_1,...,A_m)\ =(a_1^p,...,a_m^p)}(V))$, $l = 1..p$. Then for some $i \leq m$ and $j \leq n$, we let $r_l = \pi_{B_j}(\sigma_{K=k_l\wedge(A_1,...,A_m)=(a_1^i,...,a_m^i)}(V)), l = 1..p$. If there is no satisfied row in V, we let $r_l = \perp$.

We first consider the left side of Equation (8). In particular, we consider the column $f(a_1^i *$

[6] We assume that the aggregate function f will not take $\perp$ into account, i.e., treat $\perp$ as NULL value. Note that special treatment is required for COUNT. That is, if the COUNT function encounters a group with all $\perp$ value, it should output $\perp$ instead of 0.

$*a_2^i ... * *a_m^i * *B_j)$ for a given k_1'. The data to be aggregated are $\{r_l\}, l = 1..p$, i.e., the column outputs $f(\{r_l\})$.

Now we consider the right side of Equation (8), which equals $\supset\!\!\!\times\!\!\!\sqsubset_{i=1}^{q} \pi_{K', f(B_1),...,f(B_n)}(\sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)}$ $({}_{K',A_1,...,A_m}\mathcal{F}_{f(B_1),...,f(B_n)}(V))) = \supset\!\!\!\times\!\!\!\sqsubset_{i=1}^{q} \pi_{K', f(B_1),...,f(B_n)}({}_{K',A_1,...,A_m}\mathcal{F}_{f(B_1),...,f(B_n)}(\sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)}V))$. Hence, for a given value k_1' and $(a_1^i, ..., a_m^i)$, the inner GROUPBY for column B_j computes $f(\pi_{B_j}(\sigma_{K'=k_1' \wedge (A_1,...,A_m)=(a_1^i,...,a_m^i)}(V)))$. Here, we note that $\sigma_{K'=k_1' \wedge (A_1,...,A_m)=(a_1^i,...,a_m^i)}(V)$ must equal to $\cup_{l=1}^{p}(\sigma_{K=k_l \wedge (A_1,...,A_m)=(a_1^i,...,a_m^i)}(V))$. Otherwise, if there exists an extra row $(k_{p+1}, a_1^i, ..., a_m^i, ..)$ in V, where k_{p+1} also contains k_1', this row will then qualify the definition of $\{k_l\}$ and thus should have already been included.

Hence, the inner GROUPBY for column B_j actually computes: $f(\pi_{B_j}(\cup_{l=1}^{p}(\sigma_{K=k_l \wedge (A_1,...,A_m)=(a_1^i,...,a_m^i)}(V)))$. Or in other words, the output is $f(\sigma_{r_l \neq \perp}\{r_l\})$. The next GPIVOT will output either $f(\sigma_{r_l \neq \perp}\{r_l\})$ or $\perp$ if $\sigma_{r_l \neq \perp}\{r_l\}$ is empty.

Thus if the aggregate function f disregards $\perp$ value, then $f(\{r_l\}) = f(\sigma_{r_l \neq \perp}\{r_l\})$. A special requirement is that when all $\{r_l\}$ is $\perp$, $f(\{r_l\})$ should output $\perp$. For COUNT, this means that it should output $\perp$ instead of 0.

By (1) and (2), we thus establish the proof for Equation (8). ∎

5.1.5 Pullup GPIVOT through GUNPIVOT

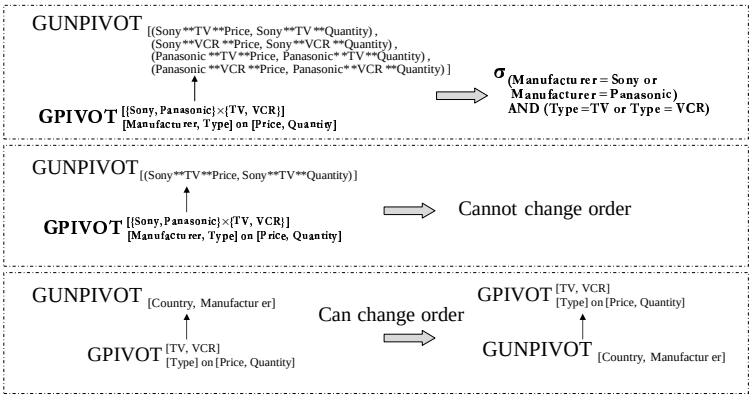
Given two adjacent GPIVOT and GUNPIVOT, if the unpivot takes *all* pivoted output columns as input parameters, then these two operators may cancel each other. For example, in the first case of Figure 12, these two operators cancel each other, replaced by a simple select condition. Formally, we have:

$$GUNPIVOT_{[(X_i \times \{B_j\})]}(GPIVOT^{[\{X_i\}]}_{[A_1...A_m] \ on \ [B_1...B_n]}(V)) = (\sigma_s(V)) \ ^{7} \qquad (9)$$

Proof for Equation (9): First, obviously, the GUNPIVOT outputs the table schema same as table V. Next, for each row $(k_1, a_1, ..., a_m, b_1, ..., b_n)$ in V, 1) if $(a_1, ..., a_m)$ does not equal to any X_i, then both sides of Equation (9) will not include it. 2) if $(a_1, ..., a_m)$ does equal to some X_i, then GPIVOT outputs $(k_1, ..., b_1, ..., b_n, ...)$, with column name for column b_i as '$a_1 * *.. * *a_m * *b_i$'. The

[7] Here $(X_i \times \{B_j\})$ are all the pivoted output columns with $(A_1, ..., A_i)$ value as X_i. σ_s is a disjunctive predicate on $(A_1...A_m)$, i.e., they equal to any X_i.

next GUNPIVOT outputs row $(k_1, a_1, ..., a_m, b_1, ..., b_n)$ [8]. Hence both sides of Equation (9) contain that row. By 1),2), Equation (9) always holds. ∎



Country	Manufacturer	Type	Price	Quantity
USA	Sony	TV	250	50
USA	Sony	VCR	50	40
USA	Panasonic	TV	220	20
Japan	Sony	VCR	60	60
Japan	Panasonic	TV	240	70

Figure 12: Pullup through GUNPIVOT

In the second case, note that the GUNPIVOT now only partially uses the pivoted output columns. Their order cannot be changed, since GUNPIVOT has to use the output of GPIVOT. Also the semantics of such operations is problematic in practice. As can be seen in the figure, the result will have some 'Sony' as column names and some as column values.

Finally, if the parameters between GPIVOT and GUNPIVOT have no overlap, as the third case in Figure 12, then their order can be reversed. Formally, we assume table V has schema $(K, G_1, ..., G_l, A_1, ..., A_m, B_1, ...B_n)$, where K denotes possibly multiple columns while G_i,A_i and B_i denote one column each. $(K, G_1, ..., G_l, A_1, ..., A_m)$ together form the key of table V.

$$GUNPIVOT_{[\{G_i\}]}(GPIVOT^{[\{X_i\}]}_{[A_1...A_m] \ on \ [B_1...B_n]}(V)) = GPIVOT^{[\{X_i\}]}_{[A_1...A_m] \ on \ [B_1...B_n]}(GUNPIVOT_{[\{G_i\}]}(V)) \quad (10)$$

Proof of Equation (10): Assume a row in V as $v = (k_1, g_1, ..., g_l, a_1, ..., a_m, b_1, ..., b_n)$. We further assume that applying $GUNPIVOT_{[\{G_i\}]}$ on this row will output p rows $(k_1, h^1_1, ..., h^1_q, a_1, ..., a_m, b_1, ..., b_n)$,..., $(k_1, h^p_1, ..., h^p_q, a_1, ..., a_m, b_1, ..., b_n)$.

[8] We assume not all $(b_1, ..., b_n)$ are $\perp$. If not, the predicate σ_s should be extended to choose those rows whose $\{B_i\}$ not all $\perp$.

Hence, for this row v, the right side of Equation (10) will first output p rows $(k_1, h_1^1, ..., h_q^1, a_1, ..., a_m, b_1, ..., b_n)$, ..., $(k_1, h_1^p, ..., h_q^p, a_1, ..., a_m, b_1, ..., b_n)$. The next GPIVOT also outputs p rows as $(k_1, h_1^1, ..., h_q^1, ..., b_1, ..., b_n, ...)$, ..., $(k_1, h_1^p, ..., h_q^p, ..., b_1, ..., b_n, ...)$, with each b_i column's name as 'a_1**..a_m**B_i'.

The left side of Equation (10) will first output $(k_1, g_1, ..., g_l, ..., b_1, ..., b_n, ...)$, with each b_i column's name as 'a_1**..a_m**B_i'. The next GUNPIVOT outputs p rows as $(k_1, h_1^1, ..., h_q^1, ..., b_1, ..., b_n, ...)$, ...,$(k_1, h_1^p, ..., h_q^p, ..., b_1, ..., b_n, ...)$. The reason is that the output of GUNPIVOT is determined by $(g_1, ..., g_l)$.

Thus, both sides of Equation (10) generates same output for each input row v. Hence Equation (10) always holds. ∎

5.2 Pushdown Rules for GPIVOT

5.2.1 Push GPIVOT Down SELECT

We now present the rules for pushing GPIVOT down the SELECT operator. Similarly, if the select condition is on key column, such as '$\sigma_{country=USA}$' in Figure 13, then we can push GPIVOT down the SELECT operator without change.

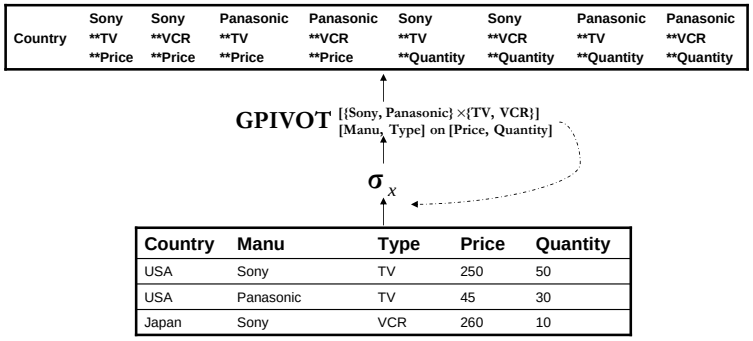


Figure 13: Pulldown through SELECT

If the select condition is on the columns to be pivoted by, such as '$\sigma_{Type=TV}$', then the pushdown results in a PROJECT, which turns all 'VCR' related columns into $\perp$, followed by a SELECT, which removes the rows that contain only '$\perp$' columns. More precisely, it becomes '$\sigma_{not\ all\perp}(\pi_{country,\ Sony**TV**Price, Sony**TV**Quantity,\ Panasonic**TV**Price, Panasonic**TV**Quantity, \perp, \perp, \perp, \perp})$'.

If the select condition is on the columns to be pivoted on, such as '$\sigma_{Price=250}$', then the pushdown results in a PROJECT, which sets the '$**Price$' column and the '$**Quantity$' column with the same prefix to $\perp$ if the '$**Price$' column does not equal 250, followed by a SELECT, which also removes the

rows that contain only $\perp$ columns. More precisely, it becomes '$\sigma_{not\ all \perp}(\pi_{country, case(Sony**TV**Price,}$ $_{Sony**TV**Quantity),}$[9] $_{case(Sony**VCR**Price,Sony**VCR**Quantity),\ case(Panasonic**TV**Price,Panasonic**TV}$ $_{**Quantity),\ case(Sony**VCR**Price,Sony**VCR**Quantity))}$'.

Formally, we have the following pushdown rule in Equation (11), assuming the same table schema V. Here the case expression, $case(a_1^{i_1} * *... * *a_m^{i_1} * *B_1, ..., a_1^{i_1} * *... * *a_m^{i_1} * *B_n)$, outputs $(a_1^{i_1} * *... * *a_m^{i_1} * *B_1, ..., a_1^{i_1} * *... * *a_m^{i_1} * *B_n)$ only when $a_u^{i_1} = x \wedge a_1^{i_1} * *... * *a_m^{i_1} * *B_v = y$. Otherwise, it outputs $(\perp, ..., \perp)$. Note that here '$a_u^{i_1} = x$' is a *higher* order predicate that the column name should contain x.

$$GPIVOT^{[\{(a_1^i,...,a_m^i)\}]}_{[A_1,...,A_m]\ on\ [B_1,...,B_n]}(\sigma_{A_u=x \wedge B_v=y}(V)) =$$
$$\sigma_{not\ all\ \perp}(\pi_{K,\{case(a_1^{i_1} * *... * *a_m^{i_1} * *B_1,...,a_1^{i_1} * *... * *a_m^{i_1} * *B_n)\}})(GPIVOT^{[\{(a_1^i,...,a_m^i)\}]}_{[A_1,...,A_m]\ on\ [B_1,...,B_n]}(V)) \quad (11)$$

Proof for Equation (11): (1) First we prove that both sides of Equation (11) generate the same set of key values. The left side outputs key value set as: $\delta_K(\sigma_{A_u=x \wedge B_v=y}(V))$. Or in other words, it outputs a key value k_1 iff there exists at least one row in V that satisfies $_{K=k_1 \wedge A_u=x \wedge B_v=y}$.

The right side outputs a key value k_1 iff there exists at least one column $a_1^{i_1} * *... * *a_m^{i_1} * *B_v$ that satisfies $a_u^{i_1} = x$ and $a_1^{i_1} * *... * *a_m^{i_1} * *B_v = y$. The original row in V that corresponds to this column must then satisfy $K = k_1 \wedge A_u = x \wedge B_v = y$. Hence, both sides generate the same set of key values.

(2) Next we prove that for each key value k_1, both sides generate the same row. For any column $a_1^{i_1} * *... * *a_m^{i_1} * *B_j$, the left side of Equation (11) outputs $\pi_{B_j}(\sigma_{K=k_1 \wedge A_1=a_1^{i_1} \wedge ... \wedge A_u=a_u^{i_1}=x ... \wedge A_m=a_m^{i_1} \wedge B_v=y}(V))$. The right side of Equation (11) outputs $\pi_{B_j}((\sigma_{a_u^{i_1}=x \wedge B_v=y})(\sigma_{K=k_1 \wedge A_1=a_1^{i_1} \wedge ... \wedge A_u=a_u^{i_1} ... \wedge A_m=a_m^{i_1}}(V)))$. Hence, both sides output the same value for any column $a_1^{i_1} * *... * *a_m^{i_1} * *B_j$.

By (1) and (2), we know that Equation (11) always holds. ∎

Note that this rule can also easily be extended to handle more complex conditions, such as disjunctive conditions. For example, if the condition in left side of Equation (11) is $\sigma_{A_u=x \vee B_v=y}(V)$, then the condition in case expression on the right side becomes $a_u^{i_1} = x \vee a_1^{i_1} * *... * *a_m^{i_1} * *B_v = y$.

5.2.2 Push GPIVOT Down PROJECT

Similarly, we also consider negative project, i.e., removal of columns. One example is shown in Figure 14. As can be seen, if the GPIVOT is pushed down, then there are two rows regarding 'USA'.

[9] Here case(column1, column2) is a case expression that if column1 does not equal to 250, then it outputs $(\perp, \perp)$, otherwise it outputs (column1,column2).

In comparison, if the GPIVOT is not pushed down, then there is one row regarding 'USA'. Hence, GPIVOT generally cannot be pushed down project, unless the removed columns are functionally determined, e.g., if 'Country → Year' holds, then we can pushdown the GPIVOT.

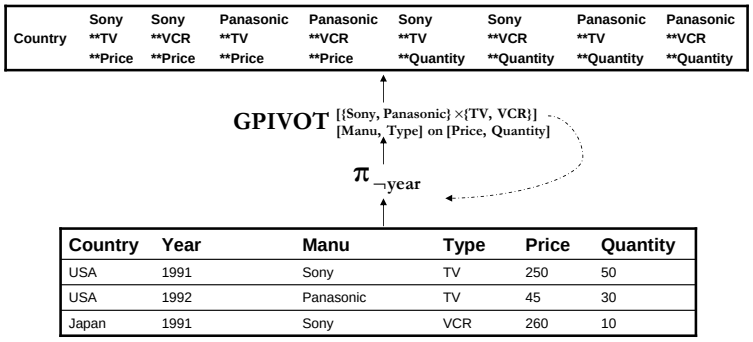


Figure 14: Pushdown through PROJECT

5.2.3 Push GPIVOT Down JOIN

Clearly, if GPIVOT takes parameter columns from both of the join tables, then we have to perform GPIVOT after the join. Now assume GPIVOT(V$\bowtie_{\sigma_1}$A), where σ_1 is the join condition. We further assume the same table schema $(K, A_1, ..., A_n, B_1, ..., B_m)$ for V, (K_2, X, Y) for A and the GPIVOT takes all parameter columns from table V. In this case, the pushdown rules are quite similar to those in Section 5.2.1

First, if the join condition σ_1 is on the key column K of table V, e.g., $K = K_2$, then we can push GPIVOT down the join without change. Second, if the join condition σ_1 is on the column to be pivoted on of table V, e.g., $B_2 = X$, then the pushdown result is $\pi_{K,\{case(a_1^i**..a_n^i**B_1,...,a_1^i**..a_n^i**B_m)\},K_2,X,Y}($ $GPIVOT(V) \bowtie_{a_1^1**..a_n^1**B_2=X \vee ... \vee a_1^p**..a_n^p**B_2=X} A$). More precisely, we apply a check between each $a_1^i * *..a_n^i * *B_2$ column and X column. If $a_1^i * *..a_n^i * *B_2 \neq X$, then we set all $a_1^i * *..a_n^i * *B_j$ columns to $\perp$ (the case expression). Finally, if the join condition σ_1 is on the column to be pivoted by of table V, e.g., $A_1 = Y$, then after we push down the GPIVOT, we need to apply a check between the column name $a_1^i * *..a_n^i * *B_j$ and the column value Y. This however requires the query language extended with such a higher order feature [14].

5.2.4 Push GPIVOT Down GROUPBY

Now assume the GROUPBY operator has group by columns $\{A_i\}$, aggregate columns $\{B_i\}$ with functional dependency $\{A_i\}\rightarrow\{B_i\}$. Due to this functional dependency, the GPIVOT operator has to pivot some A_i columns on B_i columns for applicability.

We also note that the input to the GROUPBY operator may contain duplicates. In this case, we cannot push GPIVOT down GROUPBY, since GPIVOT requires the input to contain a key. When there is a key in the input to the GROUPBY operator, Equation (8) can be applied in a reverse fashion in order to push down the GPIVOT.

5.2.5 Push GPIVOT Down GUNPIVOT

Given two adjacent GUNPIVOT and GPIVOT, they may also cancel each other when the GPIVOT takes the GUNPIVOT output columns as parameters. As can be seen in the first case in Figure 15, these two operators also cancel each other, resulting in a simple selection. Formally, assume table H has schema $(K, a_1^1 **..a_m^1 **B_1, ..., a_1^p **..a_m^p **B_n)$, where K denotes possibly multiple columns and is the key.

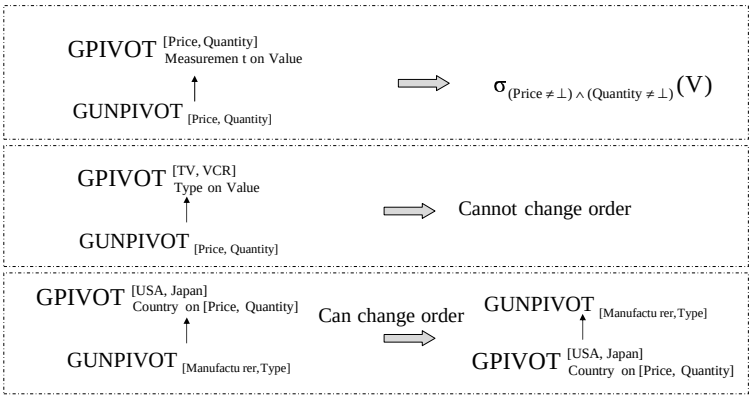
$$GPIVOT^{[\{a_1^i,...,a_m^i\}]}_{[A_1...A_m]\ on\ [B_1...B_n]}(GUNPIVOT_{[\{(a_1^i **a_m^i **B_1,...,a_1^i **a_m^i **B_n)\}]}(H)) = (\sigma_s(H))\ ^{10} \qquad (12)$$

Proof for Equation (12): First, obviously, the GUNPIVOT outputs the table schema same as table H. Next, for each row $h = (k_1, c_1, ..., c_{pn})$ in H, 1) if $(c_1, ..., c_{pn})$ all equal to $\perp$, then both sides of Equation (12) will not include it. 2) if not all $(c_1, ..., c_{pn})$ equal to $\perp$, then GPIVOT outputs a set of rows $\{(k_1, a_1^i, ..., a_m^i, b_1, ..., b_n)|$ if not all b_j equals $\perp$,i=1..p $\}$. The next GPIVOT takes these rows as input and outputs row $(k_1, c_1, ..., c_{pn})$. Hence both sides of Equation (12) contain that row. By 1),2), Equation (12) always holds. ∎

In the second case of Figure 15, note that the GPIVOT now only partially uses the pivoted output columns. Their order cannot be changed, since GPIVOT has to use the output of GUNPIVOT.

Finally, if the parameters between GPIVOT and GUNPIVOT have no overlap, as the third case in Figure 15, then their order can be reversed. This essentially is the reverse application of Equation (10) in Section 5.1.5.

[10] Here σ_s is a disjunctive predicate on $a_1^i **a_m^i **B_1, ..., a_1^i **a_m^i **B_n$, i.e., they do not all equal $\perp$.



Country	Manufacturer	Type	Price	Quantity
USA	Sony	TV	250	50
USA	Sony	VCR	50	40
USA	Panasonic	TV	220	20
Japan	Sony	VCR	60	60
Japan	Panasonic	TV	240	70

Figure 15: Push GPIVOT down GUNPIVOT

5.3 Pullup Rules for GUNPIVOT

In this section, we will present the rules for pulling up GUNPIVOT. We assume the input to GUNPIVOT contains a key, as such key usually exists in practice.

5.3.1 Pull GUNPIVOT through SELECT

There are three cases for pulling GUNPIVOT through SELECT. We now refer the unpivoted output columns that originated from column values as *value columns* and refer the unpivoted output columns that originated from column names as *name columns*. For example, in Figure 16, 'Type' is name column while 'Price' is value column.

First, if the selection condition is defined on non-unpivoted output columns, then we can push it down without any changes such as the condition $\sigma_{Country='USA'}$ in Figure 16.

Second, if the select condition is on the value column, e.g., $\sigma_{Price=150}$, then pushing this select down results in a project that changes the columns. In the above example, it becomes $\pi_{Country,case(Sony**TV**Price),}$

$_{case(Sony**VCR**Price),case(Panasonic**TV**Price),case(Panasonic**VCR**Price)}$. Here $case(column1)$ is a case expression that outputs $column1$ if $column1 = 150$, otherwise it outputs $\perp$.

Third, if the select condition is on the name column, e.g., $\sigma_{Type=TV}$, then pushing this select down results in a project that removes columns. In the above example, it becomes $\pi_{\neg(Sony**VCR**Price,}$

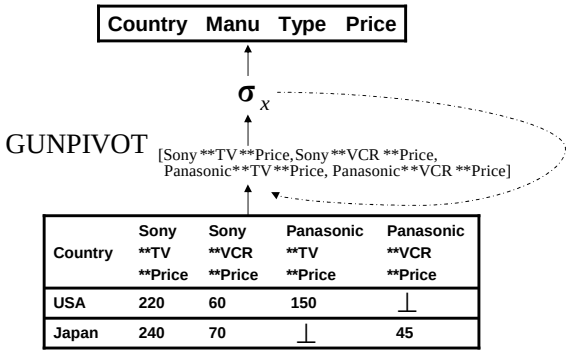


Figure 16: Pull GUNPIVOT through SELECT

$Panasonic**VCR**Price)$.

Formally, we assume table H with schema $(K, a_1^1**...a_m^1**B_1, ..., a_1^1**...a_m^1**B_n, ..., a_1^p**...a_m^p**B_1, ..., a_1^p**...a_m^p**B_n)$, where K can be multiple columns and each $a_1^i**...a_m^i**B_j$ is one column. We further assume a selection predicate as: $\sigma_{A_p=X \wedge B_q=y}(GUNPIVOT_{[\{(a_1^i**...a_m^i**B_1,...,a_1^i**...a_m^i**B_n)\}]}(H))$.

$$\sigma_{A_p=X \wedge B_q=y}(GUNPIVOT_{[\{(a_1^i**...a_m^i**B_1,...,a_1^i**...a_m^i**B_n)\}]}(H)) =$$
$$GUNPIVOT_{[\{(a_1^i**..**X..a_m^i**B_1,...,a_1^i**..**X..a_m^i**B_n)\}]}($$
$$\pi_{K,\{case(a_1^i**..**X..a_m^i**B_1,...,a_1^i**..**X..a_m^i**B_n)\}}(H)) \quad [11] \qquad (13)$$

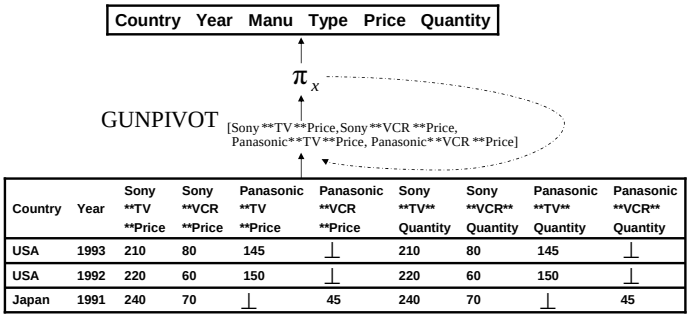
Proof of Equation (13): The proof of this rule is straightforward. The left side of Equation (13) outputs a row $r = (k_1, a_1, a_2, ..., a_m, b_1, ..., b_n)$ iff $a_p = X \wedge b_q = y$. On the right side of Equation (13), the case expression actually removes the rows that do not satisfy $a_p = X \wedge b_q = y$. Hence they are equivalent. ∎

If the condition is disjunctive, e.g., $\sigma_{\sigma_1 \vee \sigma_2}$, then we can first rewrite it to $\sigma_{\sigma_1}(GUNPIVOT) \cup \sigma_{\sigma_2}(GUNPIVOT)$. After that, we push the two select conditions down individual GUNPIVOT using the above rules.

5.3.2 Pull GUNPIVOT through PROJECT

Similarly, we also consider negative project, i.e., removal of columns. There are also three cases for pulling GUNPIVOT through PROJECT. First, if the project is to remove the non-unpivoted columns, such as $\pi_{\neg Country}$ in Figure 17, we can push the project down without changes.

[11] Here $case(a_1^i**..**X..a_m^i**B_1, ..., a_1^i**..**X..a_m^i**B_n)$ is a case expression that if $a_1^i**..**X..a_m^i**B_q = y$,



Country	Year	Sony **TV **Price	Sony **VCR **Price	Panasonic **TV **Price	Panasonic **VCR **Price	Sony **TV** Quantity	Sony **VCR** Quantity	Panasonic **TV** Quantity	Panasonic **VCR** Quantity
USA	1993	210	80	145	$\perp$	210	80	145	$\perp$
USA	1992	220	60	150	$\perp$	220	60	150	$\perp$
Japan	1991	240	70	$\perp$	45	240	70	$\perp$	45

Figure 17: Pull GUNPIVOT through PROJECT

Second, if the project is to remove the *value column* from the GUNPIVOT output, e.g., $\pi_{\neg Price}$ in Figure 17, then pulling up GUNPIVOT results in a project that removes all price related columns, i.e., $\pi_{\neg(Sony**TV**Price,Sony**VCR**Price,Panasonic**TV**Price,Panasonic**VCR**Price)}$.

Third, if the project is to remove the *name column* from the GUNPIVOT output, e.g., $\pi_{\neg Manu}$ in Figure 17, then pulling up GUNPIVOT requires to modify the column names, i.e., removing 'Sony' and 'Panasonic' from the column names.

5.3.3 Pull GUNPIVOT through JOIN

The rules for pulling GUNPIVOT above JOIN is quite similar to those for SELECT in Section 5.3.1. First, if the join predicate is on the non-unpivoted columns, then we can pull GUNPIVOT above the join without changes.

Second, if the join predicate is on the *value columns* from the output of GUNPIVOT, then the GUNPIVOT pullup results in a join followed by a project. Formally, we assume table H with schema $(K, a_1^1**...a_m^1**B_1, ..., a_1^1**...a_m^1**B_n, ..., a_1^p**...a_m^p**B_1, ..., a_1^p**...a_m^p**B_n)$, where K can be multiple columns and each $a_1^i**...a_m^i**B_j$ is one column. Table T has schema (K_1, K_2). As usual, we assume the output of GUNPIVOT(H) has schema $(K, A_1, ..., A_m, B_1, ..., B_n)$. We further assume a join predicate as: $GUNPIVOT_{[\{(a_1^i**...a_m^i**B_1,...,a_1^i**...a_m^i**B_n)\}]}(H) \bowtie_{B_l=K_1} T$.

$$GUNPIVOT_{[\{(a_1^i**...a_m^i**B_1,...,a_1^i**...a_m^i**B_n)\}]}(H) \bowtie_{B_l=K_1} T =$$
$$GUNPIVOT_{[\{(a_1^i**...a_m^i**B_1,...,a_1^i**...a_m^i**B_n)\}]}(\pi_{K,\{case(a_1^i**...a_m^i**B_1,...,a_1^i**...a_m^i**B_n)\},K_1,K_2}($$
$$H \bowtie_{a_1^1**...a_m^1**B_l=K_1 \vee ... \vee a_1^p**...a_m^p**B_l=K_1} T)) \quad [12] \quad (14)$$

then output $(a_1^i**..**X..a_m^i**B_1, ..., a_1^i**..**X..a_m^i**B_n)$, otherwise output $(\perp,...,\perp)$.

[12] Here case$(a_1^i**...a_m^i**B_1, ..., a_1^i**...a_m^i**B_n)$ is a case expression that if $a_1^i**...a_m^i**B_l = K_1$, then output

Proof of Equation (14): The proof of this rule is straightforward. The left side of Equation (14) outputs a row $r = (k, a_1, a_2, ..., a_m, b_1, ..., b_n, k_1, k_2)$ iff $b_l = k_1$. On the right side of Equation (14), the case expression actually removes the rows that do not satisfy $b_l = k_1$. Hence we conclude that they are equivalent. ∎

Third, if the join predicate is on the *name columns* from the output of GUNPIVOT, e.g., $A_l = K_2$ in the above example. Then the pullup of GUNPIVOT requires a join between the column value K_2 and the column name '$a_1^i * *... * *a_m^i * *B_j$'. This also requires a higher order feature of the query language [14].

5.3.4 Pull GUNPIVOT through GROUPBY

By first unpivoting a table and then performing aggregation, we are able to do *horizontal aggregation* [14]. As can be seen from the example in Figure 18, all the prices regarding 'USA' have been summed up even they appear as several columns in the same row.

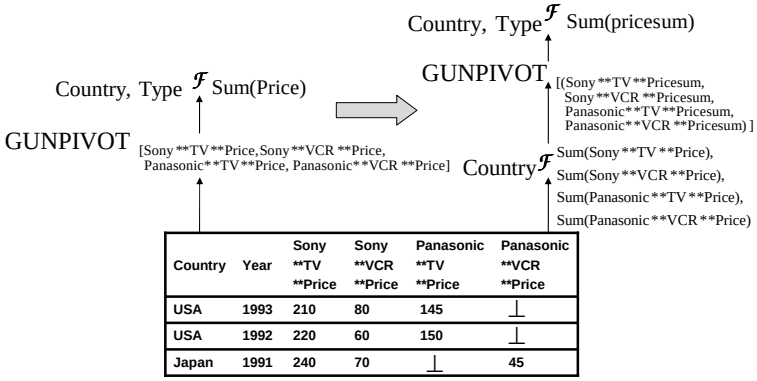


Figure 18: Pull GUNPIVOT through GROUPBY

In this case, pulling up GUNPIVOT results in a two-level aggregation as shown in Figure 18. In particular, we first aggregate all price-related columns and then unpivot individual sum totals and finally re-aggregate over these subtotals. This rule is formally described in Equation (15), assuming the same table schema H, K' as a subset of columns $(K, A_1, ..., A_m)$ and $K'' = K \cap K'$. For simplicity, we also assume here f is sum or count. We can easily extend f to *distributive* or *algebraic* functions [10].

$(a_1^i * *...a_m^i * *B_1, ..., a_1^i * *...a_m^i * *B_n)$, otherwise output $(\perp, ..., \perp)$.

$$_{K'}\mathcal{F}_{f(B_j)}(GUNPIVOT_{[\{(a_1^i **...a_m^i **B_1,...,a_1^i **...a_m^i **B_n)\}]}(H)) =$$
$$_{K'}\mathcal{F}_{f(FB_j)}(GUNPIVOT_{[\{(a_1^1 **...a_m^1 **FB_j,...,a_1^p **...a_m^p **FB_j)\}]}(_{K''}\mathcal{F}_{\{f(a_1^i **...a_m^i **B_j)\ as\ `a_1^i **...a_m^i **FB'_j\}}(H)) \quad (15)$$

Proof of Equation (15): Assume the groupby columns are $K' = (K'', A_{l_1}, ..., A_{l_p})$, with $K'' \subseteq K$ and $A_{l_i} \in \{A_1, ..., A_m\}$. On the right side of Equation (15), the inner groupby computes $a_1^i * *...a_m^i * *FB_j$, which is equivalent to compute f group by $(K'', A_1, ..., A_m)$ on the unpivoted data. Note that since $(K'', A_1, ..., A_m)$ is a superset of the next group by columns, namely, $(K'', A_{l_1}, ..., A_{l_p})$, such a two-level aggregation is already known in [5]. ∎

Next, note that even if the groupby operator does not use any output columns of GUNPIVOT, we still cannot remove the GUNPIVOT operator because it will affect the cardinality of the input.

Finally, if the groupby operator aggregates over the *name columns* from the GUNPIVOT output, e.g., *max(Type)* in the above example, then we cannot push it down since we are not able to aggregate over column names. If the groupby operator takes the *value columns* from the GUNPIVOT output, e.g., group by Price in the above example, then we also cannot push it down since we cannot group same values in different columns.

5.4 Pushdown Rules for GUNPIVOT

5.4.1 Push GUNPIVOT down SELECT

There are two cases for pushing GUNPIVOT down SELECT. First, if the select condition on the non-unpivoted columns, then we can push the GUNPIVOT operator down without changes, such as $\sigma_{Country=USA}$ in Figure 19.

Second, if the select condition is on the columns to be unpivoted, e.g., $\sigma_{Sony**TV**Price=220}$, then pushing GUNPIVOT down results in a self-join, i.e., $\pi_{Country}(\sigma_{Sony**TV**Price=220}(T)) \bowtie GUNPIVOT(T)$. Formally, assume a selection predicate over two output columns to be unpivoted as: $GUNPIVOT_{[\{(a_1^i **...a_m^i **B_1,...,a_1^i **...a_m^i **B_n)\}]}(\sigma_{a_1^{i_1} **...a_m^{i_1} **B_{l_1}\ op\ a_1^{i_2} **...a_m^{i_2} **B_{l_2}}(H))$. Here 'op' is any comparison operator.

$$GUNPIVOT_{[\{(a_1^i **...a_m^i **B_1,...,a_1^i **...a_m^i **B_n)\}]}(\sigma_{a_1^{i_1} **...a_m^{i_1} **B_{l_1}\ op\ a_1^{i_2} **...a_m^{i_2} **B_{l_2}}(H)) =$$
$$\pi_K(\sigma_{a_1^{i_1} **...a_m^{i_1} **B_{l_1}\ op\ a_1^{i_2} **...a_m^{i_2} **B_{l_2}}(H)) \bowtie GUNPIVOT_{[\{(a_1^i **...a_m^i **B_1,...,a_1^i **...a_m^i **B_n)\}]}(H) \quad (16)$$

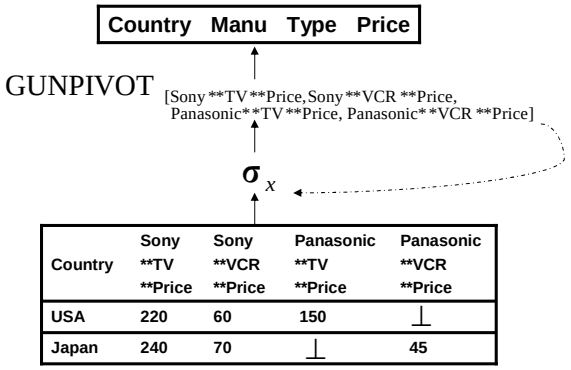


Figure 19: Push GUNPIVOT Down SELECT

Proof of Equation (16): The proof of this rule is straightforward. On the right side of Equation (16), since the join is on the key K, by Section 5.3.3, we can push it in as:

$$GUNPIVOT_{[\{(a_1^i **...a_m^i **B_1,...,a_1^i **...a_m^i **B_n)\}]}(\pi_K(\sigma_{a_1^{i_1}**...a_m^{i_1}**B_{l_1}\ op\ a_1^{i_2}**...a_m^{i_2}**B_{l_2}}(H)) \bowtie (H)) =$$
$$GUNPIVOT_{[\{(a_1^i **...a_m^i **B_1,...,a_1^i **...a_m^i **B_n)\}]}(\sigma_{a_1^{i_1}**...a_m^{i_1}**B_{l_1}\ op\ a_1^{i_2}**...a_m^{i_2}**B_{l_2}}(H)). \quad \blacksquare$$

5.4.2 Push GUNPIVOT down PROJECT

Similarly, we also consider negative project, i.e., removal of columns. Note that the GUNPIVOT operator will not take the removed columns as parameters. Hence it is always possible to push the GUNPIVOT down. For example, we can pull $\pi_{\neg Country}$ up in Figure 20. Note that we can also pull $\pi_{\neg Sony**TV**Price}$ up. The reason is that in this case, this 'Sony**TV**Price' column will not appear in the GUNPIVOT parameter column list.

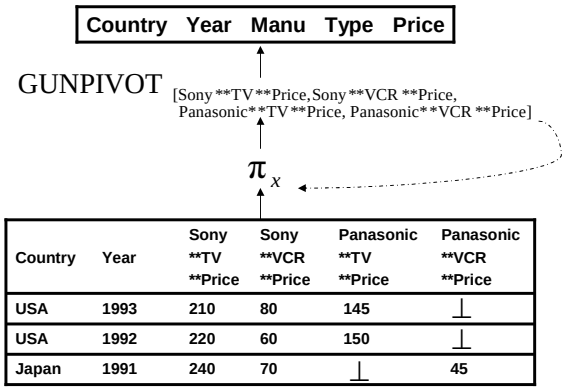


Figure 20: Push GUNPIVOT Down PROJECT

5.4.3 Push GUNPIVOT down JOIN

There are two cases for pushing GUNPIVOT down JOIN. First, if the join condition is on the non-unpivoted columns, then we can push GUNPIVOT down the join.

Second, if the join condition is on the columns to be unpivoted, then the pushdown results in self-joins. Formally, assume the same table schema H and table T with schema (K_1, K_2).

$$GUNPIVOT_{[\{(a_1^i **...a_m^i **B_1,...,a_1^i **...a_m^i **B_n)\}]}(H \bowtie_{a_1^{i_1} **...a_m^{i_1} **B_l = K_1} T) =$$
$$\pi_K(H \bowtie_{a_1^{i_1} **...a_m^{i_1} **B_l = K_1} T) \bowtie GUNPIVOT_{[\{(a_1^i **...a_m^i **B_1,...,a_1^i **...a_m^i **B_n)\}]}(H) \qquad (17)$$

Proof of Equation (17): The proof of this rule is straightforward. On the right side of Equation (17), since the join is on the key K, by Section 5.3.3, we can push it in as:

$$GUNPIVOT_{[\{(a_1^i **...a_m^i **B_1,...,a_1^i **...a_m^i **B_n)\}]}(\pi_K(H \bowtie_{a_1^{i_1} **...a_m^{i_1} **B_l = K_1} T) \bowtie (H)) =$$
$$GUNPIVOT_{[\{(a_1^i **...a_m^i **B_1,...,a_1^i **...a_m^i **B_n)\}]}(H \bowtie_{a_1^{i_1} **...a_m^{i_1} **B_l = K_1} T). \quad \blacksquare$$

5.4.4 Push GUNPIVOT down GROUPBY

First, if GUNPIVOT unpivots the aggregate columns as shown in Figure 21, then we can push GUNPIVOT down the groupby. Formally, assume the groupby operator computes, $(K, f(B_1), f(B_2), ..., f(B_n))$, where K are the group by columns and $f(B_i)$ is to compute function f over column B_i. The GUNPIVOT unpivots $(f(B_1), f(B_2), ..., f(B_n))$ and outputs *name columns* C_N, *values columns* C_V.

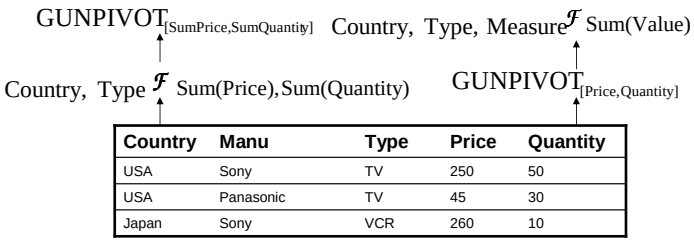
$$GUNPIVOT_{[\{f(B_i)\}]}({}_K\mathcal{F}_{\{f(B_i)\}}(T)) =_{K,C_N} \mathcal{F}_{\{f(C_V)\}}(GUNPIVOT_{[\{B_i\}]}(T)) \ ^{13} \qquad (18)$$

Proof of Equation (18): Assume for a given group by value k_1, there are a set of rows $\{t_1, ..., t_p\}$= $\{(k_1, b_1^1, ..., b_n^1), ..., (k_1, b_1^p, ..., b_n^p)\}$ in T with that group value (we ignore other columns). The left side of Equation (18) first computes $(k_1, f(b_1^j), ..., f(b_n^j))$ and unpivots to a set of rows as $\{(k_1, c_v^i, f(b_i^j))\}$, where c_v^i are the corresponding *name columns*.

The right side of Equation (18) first unpivots $\{t_1, ..., t_p\}$ to $\{(k_1, c_v^1, b_1^1), ..., (k_1, c_v^n, b_n^1), ..., (k_1, c_v^1, ..., b_n^p)$, $..., (k_1, c_v^n, ..., b_n^p)\}$ (note that some row may not exist if $b_i^j = \perp$). The next group by on (k_1, c_v^i) com-

[13] Function f should disregard $\perp$.

putes $\{(k_1, c_v^i, f(b_i^j))\}$, same as the left side. ∎



Country	Manu	Type	Price	Quantity
USA	Sony	TV	250	50
USA	Panasonic	TV	45	30
Japan	Sony	VCR	260	10

Figure 21: Push GUNPIVOT Down GROUPBY

Second, if GUNPIVOT unpivots any groupby columns, e.g., [Country,Type], then we cannot push it down. The reason is that after pushing GUNPIVOT down, we cannot perform groupby on *multi-values* from the same column. Note that this is overall quite similar to the rules for pulling up GPIVOT in Section 5.1.4.

6 Incremental View Maintenance

We now propose the propagation rules for GPIVOT and GUNPIVOT. In particular, we will show how to utilize the combination, rewriting and propagation rules together to obtain an efficient maintenance plan.

6.1 Types of ROLAP Views

In this work, we consider both aggregate and non-aggregate views containing GPIVOT and GUNPIVOT operators. We assume a key exists in the materialized view as prerequisite for enabling efficient maintenance. The reason is that if we can successfully move the GPIVOT operator to the top of the query tree (or a SELECT/GPIVOT pair on top of the query tree), then a key can be obtained from the output of GPIVOT [14]. Actually most existing view maintenance work [12, 17] also has this assumption. If there is no key in the view, i.e., it contains duplicates, the *count algorithm* [12] maintains the multiplicity of each tuple. This is equivalent to have a GROUPBY ALL operator on top of the view query. The key then would correspond to all columns. In addition, when there is a key in the view, we can use SQL update/delete statement to apply the changes

[14] In fact, our insert/delete propagation rules for GPIVOT and GUNPIVOT can be used to maintain views with duplicates. Hence this requirement is just for efficiency purposes.

efficiently. Most commercial DBMSs [3, 15] require the views to contain a key (or using rowid to arbitrarily form a key) for the above reasons. Hence our proposed techniques are applicable to the majority of the views in practice.

In this work, we also assume the GPIVOT above the GROUPBY is to pivot the aggregate results based on the group-by columns, e.g., pivot the total sales for each product type. This is common for most OLAP applications since the user often pivots the measurements by various dimensions [4]. In comparison, pivoting product type based on total sales is often problematic. The reason is that the functional dependency, measurements → dimensions, often does not hold. This makes the pivot not applicable.

6.2 Propagation Rules for GPIVOT and GUNPIVOT

The insert/delete propagation rules for GPIVOT and GUNPIVOT are depicted in Figure 22. Here '1.' means the first join operand and '2.' means the second join operand. These propagation rules are applicable to any parameters. The update propagation rules for GPIVOT are depicted in Figure 23. Note that we assume bag semantics in this paper, i.e., $\uplus$ as bag insert and $\dot{-}$ as bag delete.

$$\mathbf{GUNPIVOT}(\mathbf{H} \uplus \Delta\mathbf{H}) = \mathbf{GUNPIVOT}(\mathbf{H}) \uplus \mathbf{GUNPIVOT}(\Delta\mathbf{H})$$
$$\mathbf{GUNPIVOT}(\mathbf{H} \dot{-} \nabla\mathbf{H}) = \mathbf{GUNPIVOT}(\mathbf{H}) \dot{-} \mathbf{GUNPIVOT}(\nabla\mathbf{H})$$

$$\begin{aligned}\mathbf{GPIVOT}(\mathbf{V} \uplus \Delta\mathbf{V}) = {} & \mathbf{GPIVOT}(\mathbf{V}) \\ & \dot{-}\ \mathbf{GPIVOT}(\mathbf{V}) \ltimes_{\mathbf{K}} \mathbf{GPIVOT}(\Delta\mathbf{V}) \\ & \uplus\ \pi_{K, f(1.x, 2.x)}(\mathbf{GPIVOT}(\Delta\mathbf{V}) \bowtie_{\mathbf{K}} \mathbf{GPIVOT}(\mathbf{V})) \\ & \uplus\ \mathbf{GPIVOT}(\Delta\mathbf{V}) \overline{\ltimes}_{\mathbf{K}} \mathbf{GPIVOT}(\mathbf{V})\end{aligned}$$
$$\begin{aligned}\mathbf{GPIVOT}(\mathbf{V} \dot{-} \nabla\mathbf{V}) = {} & \mathbf{GPIVOT}(\mathbf{V}) \\ & \dot{-}\ \mathbf{GPIVOT}(\mathbf{V}) \ltimes_{\mathbf{K}} \mathbf{GPIVOT}(\nabla\mathbf{V}) \\ & \uplus\ \sigma_y(\pi_{K, g(1.x, 2.x)}(\mathbf{GPIVOT}(\nabla\mathbf{V}) \bowtie_{\mathbf{K}} \mathbf{GPIVOT}(\mathbf{V})))\end{aligned}$$

$f(1.x, 2.x)$: case when $1.x = '\perp'$ then $2.x$ else $1.x$ end (x is pivoted output column)
$g(1.x, 2.x)$: case when $1.x \neq '\perp'$ then $'\perp'$ else $2.x$ end (x is pivoted output column)
σ_y: $\sigma_{\text{any pivoted output column } x \neq '\perp'}$

Figure 22: Insert/Delete Propagation Rules for GPIVOT and GUNPIVOT

Proofs for Propagation Rules in Figure 22 and 23: We first prove the rules in Figure 22. The correctness of the GUNPIVOT rules can be shown as follows:

(1) $GUNPIVOT_{[(a_1^1 ** ... a_m^1 ** B_1, ..., a_1^1 ** ... a_m^1 ** B_n), ..., (a_1^p ** ... a_m^p ** B_1, ..., a_1^p ** ... a_m^p ** B_n)]}(H \uplus \triangle H)$

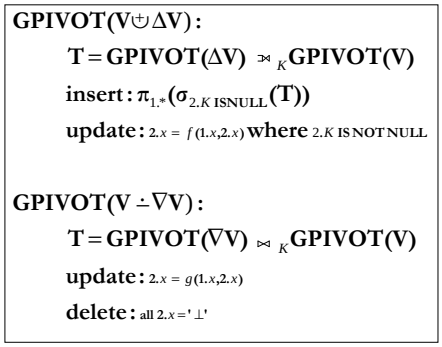


Figure 23: Update Propagation Rules for GPIVOT

$$= [\cup_{i=1}^{p} \pi_{K,`a_1^i{}',...,`a_m^i{}',a_1^i{**}...a_m^i{**}B_1,...,a_1^i{**}...a_m^i{**}B_n}(\sigma_{any\ a_1^i{**}...a_m^i{**}B_j\ \neq\perp}(H \uplus \triangle H))$$

$$= [\cup_{i=1}^{p} \pi_{K,`a_1^i{}',...,`a_m^i{}',a_1^i{**}...a_m^i{**}B_1,...,a_1^i{**}...a_m^i{**}B_n}(\sigma_{any\ a_1^i{**}...a_m^i{**}B_j\ \neq\perp}(H)] \uplus$$
$$[\cup_{i=1}^{p} \pi_{K,`a_1^i{}',...,`a_m^i{}',a_1^i{**}...a_m^i{**}B_1,...,a_1^i{**}...a_m^i{**}B_n}(\sigma_{any\ a_1^i{**}...a_m^i{**}B_j\ \neq\perp}(\triangle H)]$$

$$= GUNPIVOT_{[(a_1^1{**}...a_m^1{**}B_1,...,a_1^1{**}...a_m^1{**}B_n),...,(a_1^p{**}...a_m^p{**}B_1,...,a_1^p{**}...a_m^p{**}B_n)]}(H)$$
$$\uplus\ GUNPIVOT_{[(a_1^1{**}...a_m^1{**}B_1,...,a_1^1{**}...a_m^1{**}B_n),...,(a_1^p{**}...a_m^p{**}B_1,...,a_1^p{**}...a_m^p{**}B_n)]}(\triangle H)$$

Similarly, we can prove the GUNPIVOT rules under delete case.

(2.1) Next, we show the correctness of the rules for GPIVOT. We first prove the insert case. Given a key value k_1, we define a set of rows $\{r_i\}$ as $r_i = \pi_{K,B_1,...,B_n}(\sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)\ AND\ K=k_1}(V \uplus \triangle V))$. If there is no row that satisfies the condition for a particular i, we set $r_i = (k_1, \perp, ..., \perp)$. We also define other two sets of rows $\{p_i\}$ and $\{q_i\}$ as $p_i = \pi_{K,B_1,...,B_n}(\sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)\ AND\ K=k_1}(V))$ and $q_i = \pi_{K,B_1,...,B_n}(\sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)\ AND\ K=k_1}(\triangle V))$. Similarly, we also set p_i or q_i as $(k_1, \perp, ..., \perp)$ if no row satisfies. Based on the above definition, the following function f must hold for any i: $r_i = f(p_i, q_i)$, where $f(p_i, q_i)$ equals either p_i if $q_i = (k_1, \perp, ..., \perp)$ or equals q_i if $q_i \neq (k_1, \perp, ..., \perp)$. The reason is that $(K, A_1, ..., A_m)$ forms the key, thus at most one row exists in either V or $\triangle V$, but not both.

Based on the above definition, the output of $GPIVOT(V \uplus \triangle V)$ for key k_1 is $\bowtie \{r_i\}$. The output of $GPIVOT(V)$ for key k_1 is $\bowtie \{p_i\}$. The output of $GPIVOT(\triangle V)$ for key k_1 is $\bowtie \{q_i\}$.

If $\{p_i\}$ contain only $(k_1, \perp, ..., \perp)$ tuples, then the original output does not contain such a row with key k_1 based on the GPIVOT definition. In this case, if $\{q_i\}$ contain any non-empty tuples, then $\bowtie \{r_i\} = \bowtie \{f(p_i, q_i)\} = \bowtie \{q_i\}$. This proves the anti-join in Figure 22.

If both $\{p_i\}$ and $\{q_i\}$ contain any non-empty tuples, then $\bowtie \{r_i\} = \bowtie \{f(p_i, q_i)\}$. This proves the join term for generating new row in Figure 22. Obviously, the original row with key k_1 has to

be deleted in this case. This proves the delete term in Figure 22.

Hence the propagation rules hold under the insert case.

(2.2) We now prove the propagation rules for GPIVOT under the delete case. Given a key value k_1, we define a set of rows $\{r_i\}$ as $r_i = \pi_{K,B_1,...,B_n}(\sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)\ AND\ K=k_1}(V \dot{-} \bigtriangledown V))$. If there is no row that satisfies the condition for a particular i, we set $r_i = (k_1, \perp, ..., \perp)$. We also define other two sets of rows $\{p_i\}$ and $\{q_i\}$ as $p_i = \pi_{K,B_1,...,B_n}(\sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)\ AND\ K=k_1}(V))$ and $q_i = \pi_{K,B_1,...,B_n}(\sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)\ AND\ K=k_1}(\bigtriangledown V))$. Similarly, we also set p_i or q_i as $(k_1, \perp, ..., \perp)$ if no row satisfies. Based on the above definition, the following function g must hold for any i: $r_i = g(p_i, q_i)$, where $g(p_i, q_i)$ equals either p_i if $q_i = (k_1, \perp, ..., \perp)$ or equals $(k_1, \perp, ..., \perp)$ if $q_i \neq (k_1, \perp, ..., \perp)$. The reason is that since $(K, A_1, ..., A_m)$ forms the key, if one row is deleted from V, then the same row no longer exists in $V \dot{-} \bigtriangledown V$.

If both $\{p_i\}$ and $\{q_i\}$ contain any non-empty tuples, then the resulting row becomes $\bowtie \{r_i\} = \bowtie \{g(p_i, q_i)\}$. The original row with key k_1 has to be deleted in this case. This proves the delete term in Figure 22. We insert the new row only when not all $\{r_i\}$ equal $(k_1, \perp ..., \perp)$. This proves the insert term in Figure 22.

Hence the propagation rules also hold under the delete case.

(3) The correctness of the update propagation rules in Figure 23 can be proven by showing that they are equivalent to the insert/delete propagation rules. ∎

Figure 24 describes a simple example to show how to use our rewriting rules and propagation rules to obtain an efficient maintenance plan. Assume a materialized view is defined that first pivots the 'Items' table and then joins the results with the 'Payment' table.

First let us assume two tuples are inserted into the 'Items' table. Figure 25 depicts the change propagation by naively applying the insert/delete rules for GPIVOT. We can see that the propagation of GPIVOT generates one insert delta and one delete delta. Each of them will be joined with the 'Payment' table. The final maintenance plan is shown at the bottom of the figure. It involves several GPIVOT and joins. Also the results show that we have to delete existing view tuples and reinsert them with a few column changes.

Figure 26 depicts an alternative maintenance plan achieved by our GPIVOT pullup techniques. First, the GPIVOT operator is pulled up above the join and becomes the top of the query tree.

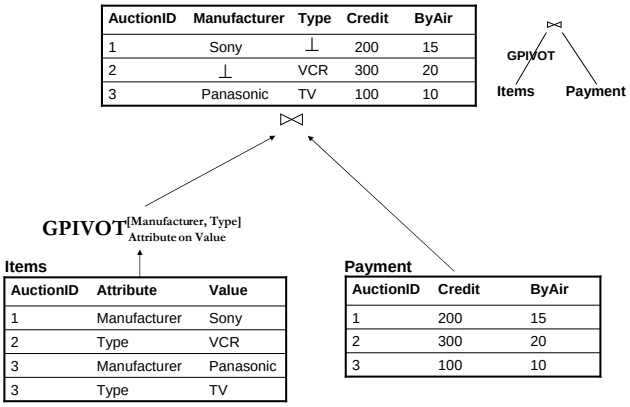


Figure 24: A Simple View with GPIVOT

The *propagation phase* propagates the source deltas through JOIN ($\Delta I \bowtie P$) and then GPIVOT to compute the *final delta* ($GPIVOT(\Delta I \bowtie P)$). The *apply phase* uses the update propagation rules for GPIVOT by evaluating a left outer-join between the final delta and the view (MV) to insert new tuples and make appropriate changes. The final plan (depicted at the bottom of the figure) is obviously more efficient than the one in Figure 25. Note that such GPIVOT pullup is only necessary when the GPIVOT is on the delta propagation path. For example, the maintenance of some inserts on 'Payment' table need not pull up the GPIVOT.

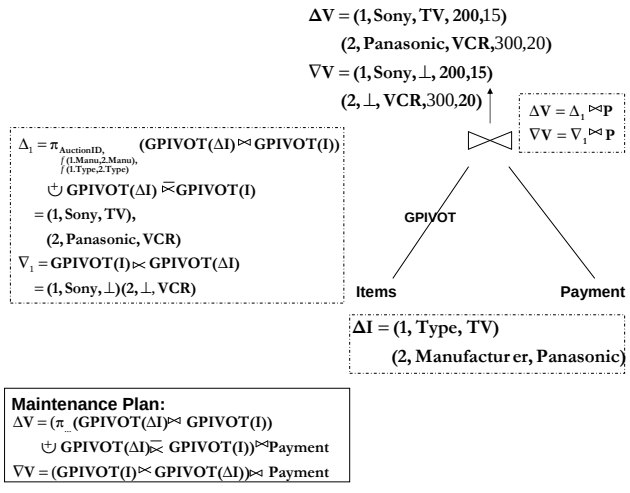


Figure 25: Maintenance without GPIVOT Pullup

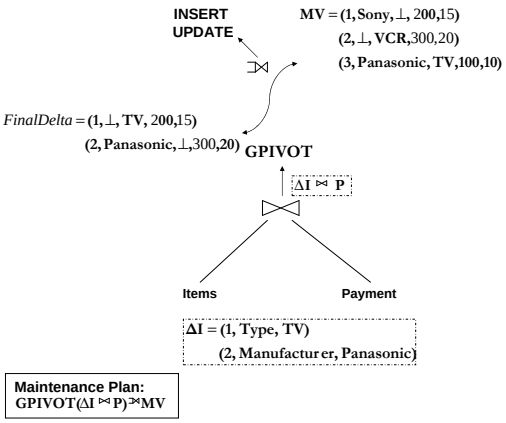


Figure 26: Maintenance with GPIVOT Pullup

6.3 Update Propagation Rules for Multiple Operators

Note that the propagation rules in Section 6.2 can be used to maintain any arbitrary view expressions. However, the update propagation rules for GPIVOT in Figure 23 require (1) the GPIVOT to be at the top of the algebra tree, and (2) the insert/delete changes to the input ΔV to be available. For some views, it might be either expensive to move the GPIVOT to top of the algebra tree or too expensive to compute the delta changes to the input. In this section, we propose to solve this problem by developing some alternative update propagation rules.

6.3.1 Update Propagation Rules for GPIVOT over GROUPBY

When GPIVOT is above a GROUPBY operator, then the input ΔV to GPIVOT is no longer trivial to compute. The reason is that we have to use the insert/delete propagation rules for GROUPBY [18]. This is inefficient since we may need to recompute some groups.

We propose to combine the GPIVOT update propagation rules and GROUPBY update propagation rules as depicted in Figure 27. Here we assume $\mathcal{F}(\mathcal{V})$ has schema $(K, A_1, ..., A_m, B_1, B_2, ...B_n)$, where $K, A_1, ..., A_m$ are group by columns, $B_1, ..., B_n$ are aggregate function columns including $count(*)$. For simplicity, we only consider SUM and COUNT in this paper. It is not hard to extend to support AVG or other algebraic functions [10]. The GPIVOT operator is assumed to pivot $B_1, ..., B_n$ by $A_1, ..., A_m$ as described in Section 6.1.

Note that when the count column, $a_1^i * *...a_m^i * *B_j = 0$, i.e., B_j is count(*) column, then all the output columns with the same prefix $a_1^i * *...a_m^i$ become empty, i.e., $a_1^i * *...a_m^i * *B_k = \perp$ for any

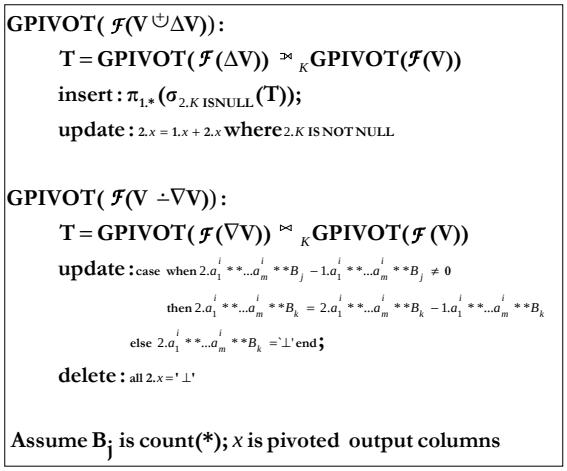


Figure 27: Update Propagation Rules for GPIVOT over GROUPBY

k. The reason is that no more item exists for this subgroup. When all the output columns become $\perp$, i.e., all subgroups are deleted, then this row should be deleted from the view.

Proofs for Propagation Rules in Figure 27: We first prove the insert case.

(1) Given a key value k_1, we define a set of rows $\{r_i\}$ as $r_i =_{K,f_1,...,f_n} \mathcal{F}(\sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)\ AND\ K=k_1}(V \uplus \triangle V))$, where each f_j is an aggregate function. If there is no row that satisfies the condition for a particular i, we set $r_i = (k_1, \perp, ..., \perp)$. We also define other two sets of rows $\{p_i\}$ and $\{q_i\}$ as $p_i =_{K,f_1,...,f_n} \mathcal{F}(\sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)\ AND\ K=k_1}(V))$ and $q_i =_{K,f_1,...,f_n} \mathcal{F}(\sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)\ AND\ K=k_1}(\triangle V))$. Similarly, we also set p_i or q_i as $(k_1, \perp, ..., \perp)$ if no row satisfies.

Based on the above definition, the output of $GPIVOT(\mathcal{F}(V \uplus \triangle V))$ for key k_1 is $\bowtie \{r_i\}$. The output of $GPIVOT(\mathcal{F}(V))$ for key k_1 is $\bowtie \{p_i\}$. The output of $GPIVOT(\mathcal{F}(\triangle V))$ for key k_1 is $\bowtie \{q_i\}$.

By applying the propagation rules for GROUPBY in [18], we have: if $p_i = (k_1, \perp, ..., \perp)$, then $r_i = q_i$; if $p_i \neq (k_1, \perp, ..., \perp)$, then $r_i = (k_1, p_i.f_1 + q_i.f_1, ..., p_i.f_n + q_i.f_n)$, assuming each aggregate function f_j is either sum or count.

If $\{p_i\}$ contains only $(k_1, \perp, ... \perp)$ tuples, then the original output does not contain such a row with key k_1. In this case, if $\{q_i\}$ contains any non-empty tuples, then $\bowtie \{r_i\} = \bowtie \{q_i\}$. This proves the insert term in Figure 27.

If both $\{p_i\}$ and $\{q_i\}$ contain any non-empty tuples, then $\bowtie \{r_i\} = \bowtie \{(k_1, p_i.f_1 + q_i.f_1, ..., p_i.f_n + q_i.f_n)\}$. This proves the update term for generating new row in Figure 27.

Hence the propagation rules hold under the insert case.

(2) Next, we prove the delete case. Given a key value k_1, we define a set of rows $\{r_i\}$ as $r_i =_{K,f_1,...,f_n} \mathcal{F}(\sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)\ AND\ K=k_1}(V \dot{-} \bigtriangledown V))$, where each f_j is an aggregate function. If there is no row that satisfies the condition for a particular i, we set $r_i = (k_1, \perp, ..., \perp)$. We also define other two sets of rows $\{p_i\}$ and $\{q_i\}$ as $p_i =_{K,f_1,...,f_n} \mathcal{F}(\sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)\ AND\ K=k_1}(V))$ and $q_i =_{K,f_1,...,f_n} \mathcal{F}(\sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)\ AND\ K=k_1}(\bigtriangledown V))$. Similarly, we also set p_i or q_i as $(k_1, \perp, ..., \perp)$ if no row satisfies.

Based on the above defintion, the output of $GPIVOT(\mathcal{F}(V \dot{-} \bigtriangledown V))$ for key k_1 is $\bowtie \{r_i\}$. The output of $GPIVOT(\mathcal{F}(V))$ for key k_1 is $\bowtie \{p_i\}$. The output of $GPIVOT(\mathcal{F}(\bigtriangledown V))$ for key k_1 is $\bowtie \{q_i\}$.

By applying the propagation rules for GROUPBY in [18], we have $r_i = (k_1, p_i.f_1 - q_i.f_1, ..., p_i.f_n - q_i.f_n)$, assuming each aggregate function f_j is either sum or count. Furthermore, if the count(*) column, say $p_i.f_j - q_i.f_j = 0$, then $r_i = (k_1, \perp, ..., \perp)$.

Thus, $\bowtie \{r_i\} = \bowtie \{(k_1, p_i.f_1 - q_i.f_1, ..., p_i.f_n - q_i.f_n)\}$. If the count(*) column $r_i.f_j = p_i.f_j - q_i.f_j = 0$, then we need to set $r_i = (k_1, \perp, ..., \perp)$. This proves the update term in Figure 27. If the resulting $\{r_i\}$ contain only $(k_1, \perp\ ..., \perp)$ tuples, then we should delete this row from the pivoted output based on the GPIVOT definition. This proves the delete term in Figure 27.

Hence the propagation rules also hold under the delete case. ∎

We now use the motivating example (Figure 2) in Section 2.1 to describe how to use these update propagation rules to efficiently maintain this view. We first pullup and combine multiple pivot operators in the query. For example, the top two pivots can be combined into one operator, denoted as $G_1 = GPIVOT^{[TV,VCR]}_{Type\ on\ [CreditSum,ByAirSum]}$ as described in Section 4.2. The lower pivot can be pulled up through JOIN and GROUPBY denoted as $G_2 = GPIVOT^{[Credit,ByAir]}_{Payment\ on\ [Sum]}$ using the rewriting rules in Section 5. Then we combine G_1 and G_2 using the composition rules in Section 4.3. Since the original view only contains SUM, we also need to add COUNT(*) into the view definition in order to make the view incrementally maintainable (Figure 28).

Then we construct the maintenance plan based on this rewritten view query. We can propagate the changes through the algebra tree (propagation phase) and apply the GPIVOT/GROUPBY update propagation rules in Figure 27 (apply phase) in order to maintain the view. Note that in

Figure 28 since the resulting VCR**Credit**Cnt for Panasonic equals 0, both VCR**Credit**Cnt and VCR**Credit**Sum will be set to ⊥. Consequently, since all the pivoted output columns of 'Panasonic' become empty, this row can be deleted from the view. In comparison, the update propagation rules in Figure 23 require significant recomputation of the group-by operator.

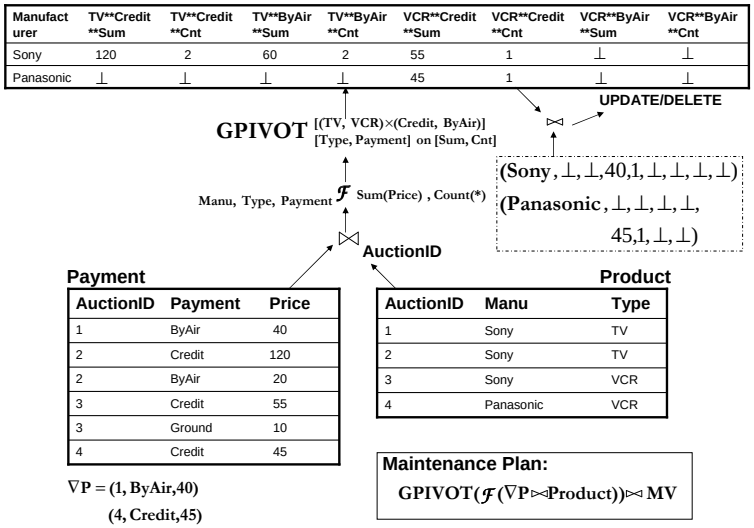


Figure 28: Maintenance of View in Figure 2

6.3.2 Update Propagation Rules for SELECT over GPIVOT

As mentioned, the other problem with the update propagation rules in Figure 23 is that it might be expensive to move the GPIVOT to the top of the algebra tree. For example, the rewriting rules in Section 5.1 show that the pullup of GPIVOT through SELECT may result in multiple self-joins. The propagation through multiple self-joins generates multiple join terms. This can be quite inefficient (Section 7). In this section, we propose alternative update propagation rules for SELECT (σ_c) on top of a GPIVOT as depicted in Figure 29.

We first describe the simple delete case. Figure 30 depicts a simple view query with a SELECT above the GPIVOT. To maintain the deletes on the 'Items' table as shown in Figure 31, the pullup of GPIVOT above this selection generates multiple self-joins. Alternatively, as described in Section 5.1, we pull both SELECT and GPIVOT up to the top of the query tree. Then we propagate the changes below the GPIVOT operator. The apply phase uses the update propagation rules in Figure 29. It first performs a join between the final delta and the view. We delete the view tuple that no longer satisfies the select condition. This is stricter than deleting the view tuple with all

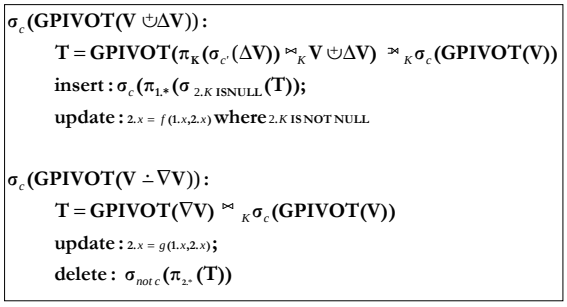


Figure 29: Update Propagation Rules for SELECT over GPIVOT

entries empty as in Figure 23. For example, the resulting tuple (3,Panasonic,⊥,300,20) will be deleted from the view since it no longer satisfies the condition. Such tuple will be retained if there is no such SELECT on top of GPIVOT.

The intuition behind this idea is that the deleted source tuples may cause an existing view tuple to no longer satisfy the condition, which can be removed by the postponed selection filtering during the apply phase, such as the auction 3 in our example. Or they may cause an existing view tuple to update some of its columns but still satisfy the condition, such as auction 1. Lastly, they may affect some pivot output tuples that originally do not satisfy the condition hence are not in the view, such as auction 2. An important observation is that if an original pivot output tuple does not satisfy the condition, then after some deletion it still will not satisfy the condition (if the condition is null-intolerant). The join between the delta and the view, as a side product, effectively removes such tuples.

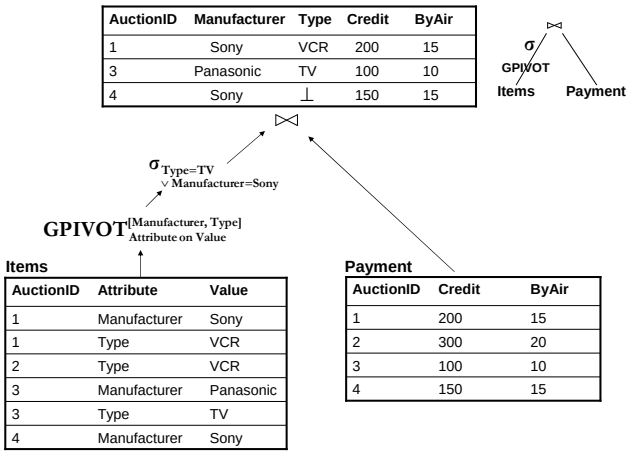


Figure 30: Views with SELECT over GPIVOT

In comparison, the source inserts may cause an originally unsatisfied tuple to now satisfy the

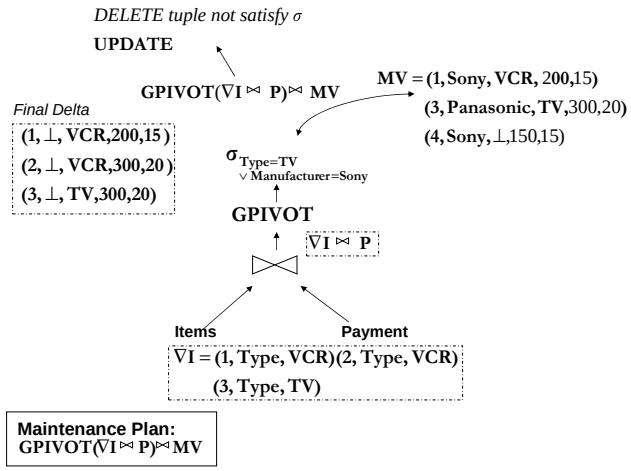


Figure 31: Maintenance with SELECT over GPIVOT

select condition. If so, we have to find some other tuples that are not in the view originally in order to construct a new view tuple. E.g., in order to maintain the insert $(2, Manufacturer, Sony)$, we have to locate the source tuple $(2, Type, VCR)$ to generate a new view tuple. The maintenance plan generated based on the rules in Figure 29 is shown below.

$GPIVOT((\pi_{ID}(\sigma_{c'}(\Delta I) \bowtie P) \bowtie I \uplus \Delta I) \bowtie P) \square\!\!\bowtie MV$,

where $\sigma_{c'} = \sigma_{(Attribute=Type \wedge Value=TV) \vee (Attribute=Manufacturer \wedge Value=Sony)}$.

Note that here σ'_c is to push some of simple selection predicates down in order to reduce the join size. The rationale is that only those deltas that are related to the columns referenced in the select predicate may change the result of the predicate, and consequently generate new view tuples. We now formally prove these rules below.

Proofs for Propagation Rules in Figure 29: We first prove the delete case.

(1) Given a key value k_1, we define a set of rows $\{r_i\}$ as $r_i = \pi_{K,B_1,...,B_n}(\sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)\ AND\ K=k_1}(V \dot{-} \bigtriangledown V))$. If there is no row that satisfies the condition for a particular i, we set $r_i = (k_1, \perp, ..., \perp)$. We also define two other sets of rows $\{p_i\}$ and $\{q_i\}$ as $p_i = \pi_{K,B_1,...,B_n}(\sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)\ AND\ K=k_1}(V))$ and $q_i = \pi_{K,B_1,...,B_n}(\sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)\ AND\ K=k_1}(\bigtriangledown V))$. Similarly, we also set p_i or q_i as $(k_1, \perp, ..., \perp)$ if no row satisfies.

Based on the above definition, the output of $GPIVOT(V \dot{-} \bigtriangledown V)$ for key k_1 is $\bowtie \{r_i\}$. The

output of $GPIVOT(V)$ for key k_1 is $\bowtie \{p_i\}$. The output of $GPIVOT(\bigtriangledown V)$ for key k_1 is $\bowtie \{q_i\}$. Similar to the proof for the rules in in Figure 22, the following function g must hold: $\bowtie \{r_i\} = \bowtie \{g(p_i, q_i)\}$.

Then we consider the following two cases:

i) If the original row $\bowtie \{p_i\}$ satisfies the condition σ_c, then $\sigma_c(\bowtie \{p_i\}) = \bowtie \{p_i\}$, since $\bowtie \{p_i\}$ represents one row. Or in other words, the original output contains the row $\bowtie \{p_i\}$. Hence, we can compute the new row by $\sigma_c(\bowtie \{r_i\}) = \sigma_c(\bowtie \{g(p_i, q_i)\})$, If the resulting row $\bowtie \{r_i\}$ statisfies the condition σ_c, then we perform updates using function G. If the resulting row $\bowtie \{r_i\}$ no longer satisfies the condition σ_c, then we delete this row from the result. This exactly corresponds to the rules in Figure 29.

ii) If the original row $\bowtie \{p_i\}$ does not satisfies the condition σ_c, then the resulting row $\bowtie \{r_i\}$ will not satisfy the condition either, since the condition is null-intolerant. In this case, $\sigma_c(\bowtie \{p_i\}) \bowtie (\bowtie \{q_i\})$ evaluates to empty result and will not make any changes.

Hence we proved that this update propagation rule always holds under the delete case.

(2) Next, we prove the insert case. Given a key value k_1, we define a set of rows $\{r_i\}$ as $r_i = \pi_{K,B_1,...,B_n}(\sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)\ AND\ K=k_1}(V \uplus \bigtriangleup V))$. If there is no row that satisfies the condition for a particular i, we set $r_i = (k_1, \perp, ..., \perp)$. We also define two other sets of rows $\{p_i\}$ and $\{q_i\}$ as $p_i = \pi_{K,B_1,...,B_n}(\sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)\ AND\ K=k_1}(V))$ and $q_i = \pi_{K,B_1,...,B_n}(\sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)\ AND\ K=k_1}(\bigtriangleup V))$. Similarly, we also set p_i or q_i as $(k_1, \perp, ..., \perp)$ if no row satisfies.

Based on the above definition, the output of $GPIVOT(V \uplus \bigtriangleup V)$ for key k_1 is $\bowtie \{r_i\}$. The output of $GPIVOT(V)$ for key k_1 is $\bowtie \{p_i\}$. The output of $GPIVOT(\bigtriangleup V)$ for key k_1 is $\bowtie \{q_i\}$. Similar to the proof for the rules in in Figure 22, the following function f must hold: $\bowtie \{r_i\} = \bowtie \{f(p_i, q_i)\}$.

Then we consider the following two cases:

i) If the original row $\bowtie \{p_i\}$ satisfies the condition σ_c, then $\sigma_c(\bowtie \{p_i\}) = \bowtie \{p_i\}$, since $\bowtie \{p_i\}$ represents one row. Or in other words, the original output contains row $\bowtie \{p_i\}$. Hence we can compute the new row by $\sigma_c(\bowtie r_i) = \sigma_c(\bowtie f(p_i, q_i))$. Note that in this case the resulting row $\bowtie \{r_i\}$ must still satisfy the condition σ_c. The reason is that if $\bowtie \{r_i\}$ did not satisfy the condition, then turning some of its columns to null, i.e., $\bowtie \{p_i\}$, still should not satisfy the condition (if σ_c is null-intolerant). We thus only need to perform updates using function F, as the update term in

Figure 29.

ii) If the original row $\bowtie \{p_i\}$ does not satisfy the condition σ_c, the resulting $\bowtie \{r_i\}$ may now satisfy the condition. Hence, in this case, we have to *recompute* $\bowtie \{r_i\}$, since the original output does not contain the row $\bowtie \{p_i\}$. The recomputation can be evaluated as $GPIVOT(\pi_K(\triangle V) \bowtie (V \uplus \triangle V))$. We then insert these new rows that now satisfy the condition, $\sigma_c(GPIVOT(\pi_K(\triangle V) \bowtie (V \uplus \triangle V)))$.

Furthermore, we observe that only when those rows in $\triangle V$ that related to the columns referenced in σ_c may change the resulting row now satisfy σ_c. Hence, we can add $\sigma_{c'}$ to reduce the join size as in Figure 29, where $\sigma_{c'}$ is of the form $(A_1, ..., A_m) = (a_1^{i_1}, ..., a_m^{i_1}) \vee ... \vee (A_1, ..., A_m) = (a_1^{i_l}, ..., a_m^{i_l})$.

Hence we proved that this update propagation rule always holds under the insert case. ∎

7 Experimental Evaluation

In this section, we will experimentally evaluate the effectiveness of our proposed techniques for incremental view maintenance. In particular, we will evaluate the performance of the maintenance plans constructed by various propagation rules proposed in Section 6.

7.1 Setup

We choose a non-intrusive implementation [14, 2] of the GPIVOT operator on top of a commerical DBMS (Oracle 10g [1]). In particular, similar to [8], we implement GPIVOT using the following SQL GROUPBY subquery:

SELECT K, $\{max(case((A_1, ..., A_m) = (a_1^i, ..., a_m^i), B_1, \perp)),$
$max(case((A_1, ..., A_m) = (a_1^i, ..., a_m^i), B_2, \perp)),$
...
$max(case((A_1, ..., A_m) = (a_1^i, ..., a_m^i), B_n, \perp))\}$
FROM V
WHERE $(A_1, ..., A_m)$ *in* $\{(a_1^i, ..., a_m^i)\}$
GROUP BY K

Note that the maintenance plan constructed by our method contains the propagation phase and the apply phase (Section 3). The propagate phase computes the *final delta*. The apply phase first evaluates a join between the final delta and materialized view. Based on the join results, INSERT, DELETE or UPDATE will be applied to the view. It is important to apply these DML in one statement in order to avoid materializing any temporary results. For this, we will use the MERGE

operation [1] to achieve this. In particular, the MERGE operation inserts a row if there is no match between the final delta and the materialized view, updates a row in the materialized view if there is such a match, and deletes a row if the updated row now contains only $\perp$ entries or no longer satisfies the select condition.

We run our experiments on a TPC-H [20] database with scale factor 1.0, i.e., total around 1 Gigabyte data. All the experiments are conducted on a two 540MHZ-CPU machine with 1G memory, running Linux. We allocate 200M memory for buffer cache and 200M memory for Sort and Hashjoin.

7.2 Maintaining Non-aggregate Views

Our experiments focus on different types of views, different sizes and characteristics of the source changes and how different maintenance methods perform under these conditions. We first consider non-aggregate views.

7.2.1 Without SELECT on TOP of GPIVOT

Figure 32 gives the algebra definition of a non-aggregate materialized view. As can be seen, it first pivots the Lineitem table and then joins with the Orders and Customer tables. The size of this view is 1,500,000 rows.

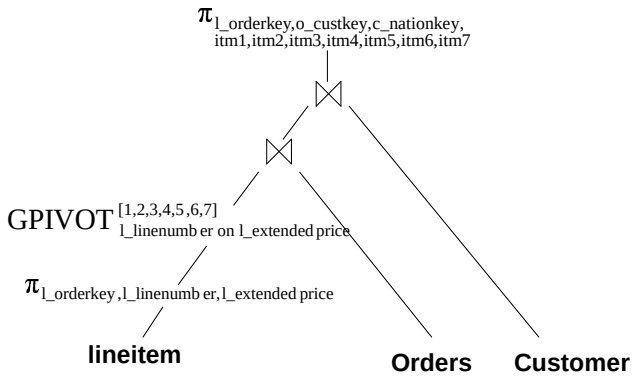


Figure 32: Materialized View Definition (1)

We first consider the delete case on Lineitem table. The following three methods can be used to refresh the view. The first method is to perform full recomputation. The second method is to perform incremental maintenance using the insert/delete propagation rules for GPIVOT in

Figure 22, while the third method is to first pullup GPIVOT to the top of the algebra tree and then apply the update propagation rules in Figure 23.

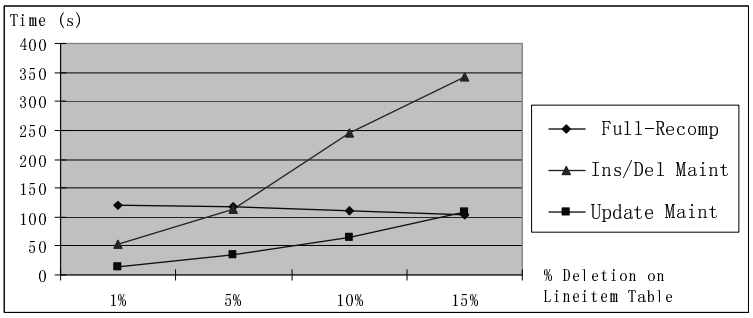


Figure 33: Maintenance of View (1) under Deletion

Figure 33 depicts the maintenance results. Here y-axis denotes the maintenance cost in seconds. x-axis denotes the percentages of deletion on Lineitem table. As can be seen from the figure, the maintenance method using the update rules considerably outperforms the method using the insert/delete rules by order of magnitude. This is due to the costly maintenance plan generated by the insert/delete rules, which can be easily seen from two examples in Figure 25 and 26.

Next, we consider the insert case on Lineitem table. In particular, we distinguish between two extreme cases. The first case is that the insert of the source data causes only updates to the view. Under this scenario, the update propagation rules likely will significantly outperform the insert/delete propagation rules as the former can avoid deleting the view tuples and re-inserting them again. The second case is that the insert of the source data causes only inserts to the view. Under this scenario, the insert/delete rules may perform better since such deletion and re-insertion would not ever occur. The goal of this experiment is to justify if the update propagation rules are always preferable choices.

Figure 34 depicts the maintenance results for the first case when the source changes result in only view updates. As can be seen, the maintenance using the update propagation rules significantly outperforms the other two alternatives as the costly deletion and then re-insertion is avoided. The results are quite similar to the delete case.

Figure 35 depicts the maintenance performance for the second case when the source changes result in only view insertions. Under this scenario, the maintenance using the insert/delete rules perform much better than the former case. The reason is that we need not delete any existing

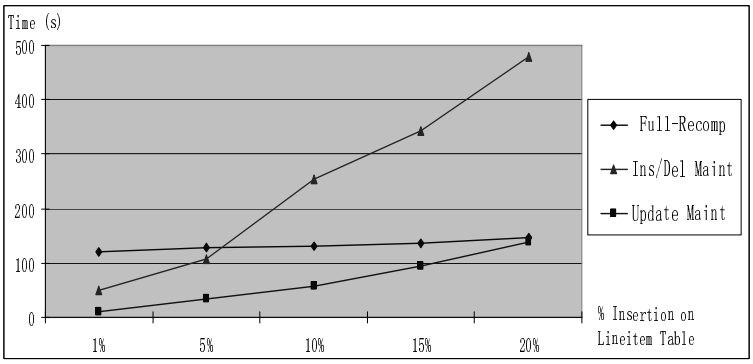


Figure 34: Maintenance of View (1) under Insertion (Only View Updates)

rows in the view and re-insert them again. However, the maintenance using the update rules still outperform the maintenance using the insert/delete rules.

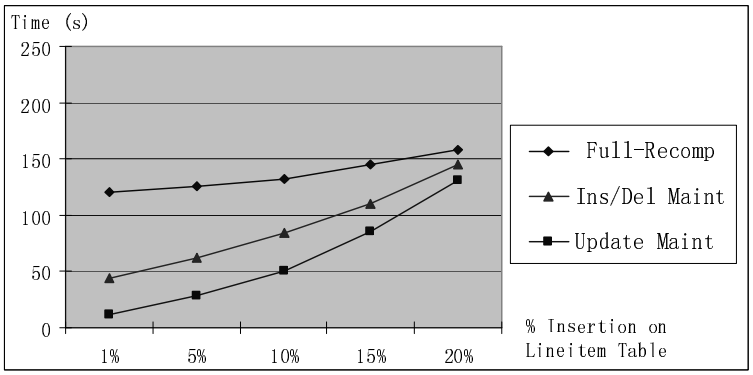


Figure 35: Maintenance of View (1) under Insertions (Only View Insertions)

The reason is that the insert/delete rules need to access the original pivot result, i.e., GPIVOT(lineitem), as in Figure 22. Even if we can push down the join predicate so as to partially compute GPIVOT(lineitem), still it can be a significant cost. In comparison, by first pulling up GPIVOT and then applying the update propagation rules, we can avoid computing any portion of GPIVOT(lineitem). Instead a join with the materialized view will be performed, which is less costly. Hence, in conclusion, the experiments in Figure 33, 34 and 35 confirm our basic heuristics that the update rules are always preferable choices than the insert/delete rules.

7.2.2 With SELECT on TOP of GPIVOT

Figure 36 gives the algebra definition of another type of view, namely, non-aggregate materialized view with a SELECT on top of GPIVOT. As can be seen, it first pivots the Lineitem table and

then chooses these rows whose first item price is greater than 30000. The results are then joined with the Orders and Customer tables. The size of this view is 890,000 rows.

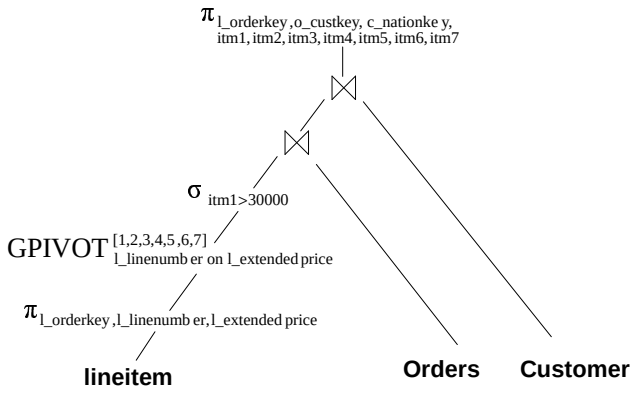


Figure 36: Materialized View Definition (2)

We first consider the delete case on Lineitem table. The following four methods can be used to refresh the view. The first method is to perform full recomputation. The second method is to perform incremental maintenance using the insert/delete propagation rules for GPIVOT in Figure 22. The third method is to pullup GPIVOT to the top of the algebra tree, i.e., pushing SELECT down GPIVOT, in order to apply the update rules in Figure 23. The fourth method is to pull both SELECT and GPIVOT up and apply the combined update propagation rules in Figure 29.

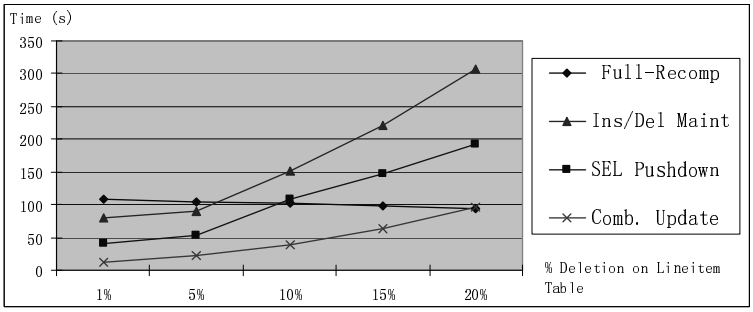


Figure 37: Maintenance of View (2) under Deletion

Figure 37 depicts the maintenance results. Here y-axis denotes the maintenance cost in seconds. x-axis denotes the percentages of deletion on Lineitem table. As can be seen from the figure, the maintenance method using our combined update rules (in Figure 29) considerably outperforms all three alternatives. While moving GPIVOT to the top of algebra tree by pushing down SELECT outperforms the method using insert/delete rules, it still generates more costly maintenance

plan compared to the method using our combined update rules. This can be easily seen by comparing the maintenance plans generated by these two methods. The maintenance plan by the SELECT/GPIVOT combined update propagation rules is

$(GPIVOT(\bigtriangledown L) \bowtie C \bowtie O) \bowtie MV$ [15].

In comparison, the maintenance by pushing down the SELECT operator is

$((GPIVOT(\pi_{l\_orderkey}(\sigma_{c'} \bigtriangledown L) \bowtie (L \dot{-} \bigtriangledown L) \uplus \pi_{l\_orderkey}(\sigma_{c'} L) \bowtie \bigtriangledown L) \bowtie O \bowtie C) \bowtie MV$ [16].

Obviously, propagating changes through multiple self-joins is non-trivial, as it generates multiple join terms [12]. Note that when the select condition involves more pivoted output columns, then more self-joins will be generated when pushing down the select operator. Hence, the select pushdown method will likely perform even worse in this case.

Next, we consider the insert case on Lineitem table. Figure 38 depicts the maintenance results.

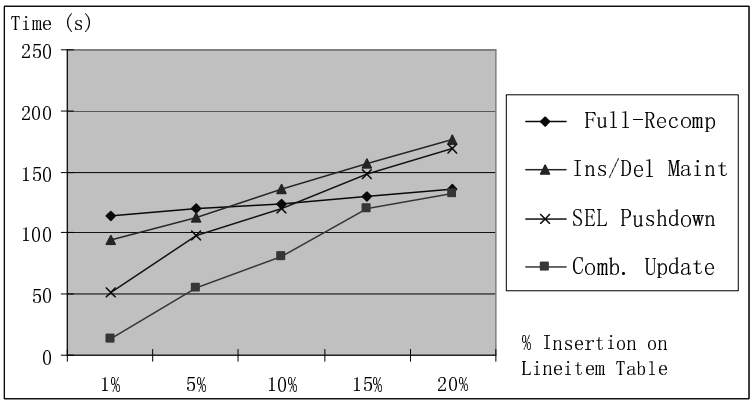


Figure 38: Maintenance of View (2) under Insertion

We find that our combined update propagation rules again outperform all other alternatives. Here the maintenance plan generated by combined update rules is

$((GPIVOT(\pi_{l\_orderkey}(\sigma_{c'}(\triangle L)) \bowtie L \uplus \triangle L) \bowtie O \bowtie C) \bowtie MV$.

In comparison, the maintenance plan by SELECT pushdown is

$((GPIVOT(\pi_{l\_orderkey}(\sigma_{c'}(\triangle L)) \bowtie (L \uplus \triangle L) \uplus \pi_{l\_orderkey}(\sigma_{c'} L) \bowtie \triangle L) \bowtie O \bowtie C) \bowtie MV$.

Clearly, the latter plan generates more join terms and will generate even more join terms when the select condition is more complex. In conclusion, our combined update rules are always preferable choices.

[15] Here L denotes Lineitem table, O denotes Orders table, C denotes Customer table, $\bigtriangledown L$ denotes deletion on Lineitem table and MV denotes materialized view.

[16] Here c' is $\sigma_{l\_linenumber=1 \wedge l\_extendedprice>30000}$

7.3 Maintaining Aggregate Views

Figure 39 gives the algebra definition of an aggregate materialized view. As can be seen, it first joins the Lineitem, Orders and Customer tables and then computes total price and count for each customer, nationality and year. After that, the summary data is pivoted by year on both sum and cnt in order to provide a crosstab view. The size of this view is 100,000 rows with 12 columns.

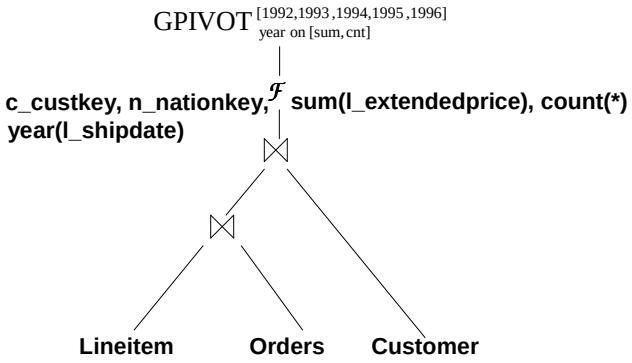


Figure 39: Aggregate Materialized View Definition (3)

We first consider the delete case on Lineitem table. The following three methods can be used to refresh the view. The first method is to perform full recomputation. The second method is to perform incremental maintenance using update rules for GPIVOT and using insert/delete rules for GROUPBY. The third method is to use combined update propagation rules for both GPIVOT and GROUPBY as in Figure 27.

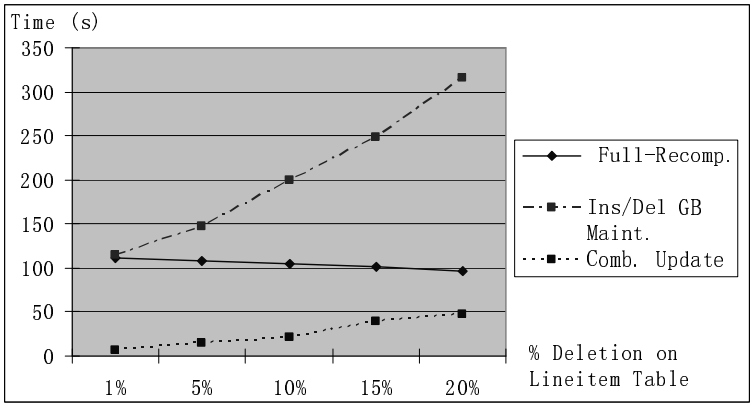


Figure 40: Maintenance of View (3) under Deletion

Figure 40 depicts the maintenance results. Here y-axis denotes the maintenance cost in seconds. x-axis denotes the percentages of deletion on Lineitem table. As can be seen from the figure, the

maintenance method using our combined update rules (in Figure 27) considerably outperforms the one using the insert/delete rules for GROUPBY [18] by order of magnitude. The reason is that the insert/delete rules for GROUPBY [18] are non-trivial, which involve costly identification and then recomputation of affected groups. Our combined update rules avoid using insert/delete rules for both GPIVOT and GROUPBY. Hence they perform much better.

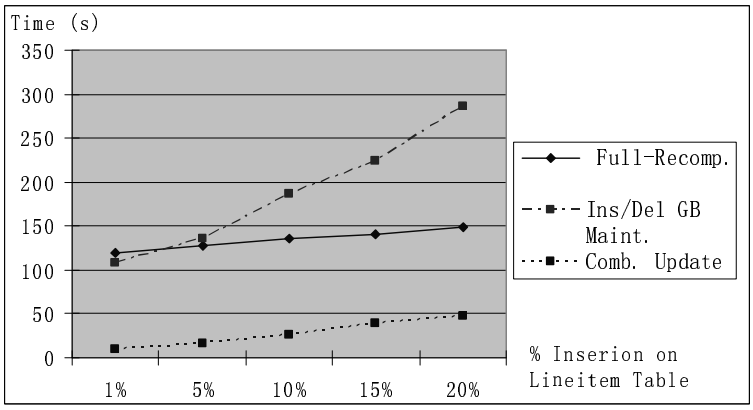


Figure 41: Maintenance of View (3) under Insertion

Next, Figure 41 depicts the maintenance results under insertion case. As can be seen, the results are similar to the deletion case. The maintenance using combined update rules performs significantly better. In conclusion, our combined update rules are always preferable choices for maintaining aggregate views.

8 Related Work

Incremental view maintenance has received considerable attentions from the database community for the last few years [12, 11, 17, 6, 19]. In [12], the authors propose algorithms for incremental view maintenance under bag semantics and also support recursive views in Datalog. In [11], the authors establish an algebraic framework for propagating deltas through each operator, which is more robust and extensible to new language constructs. In this work, we propose to extend this framework to also support pivot and unpivot operators.

PIVOT is similar to GROUPBY in many ways [8]. In [18], the authors propose the insert/delete and update propagation rules for GROUPBY operator. They also show that it is more preferable to use the latter rules. However, unlike the PIVOT operator, the GROUPBY operator loses the

detailed data. Hence the combination and pullup rules for GROUPBY are fairly restrictive. As a result, most commercial database systems only support SQJ+GROUPBY views. Fortunately, the pivot operator has a lot of interesting properties since it keeps the detailed data. As shown in this paper, they can be combined in many ways, resulting in a generalized pivot operator. They can also be pulled up in the query algebra tree, which is more flexbile than that for group by operator. As a result, it is possible to derive an efficient maintenance plan.

In [8], the authors propose the optimization and execution strategies for pivot and unpivot in Microsoft SQL Server. In fact, similar techniques can also be applied to include the GPIVOT and GUNPIVOT into the query engine as also briefly mentioned by the authors. In this paper, we address another important aspect of the pivot and unpivot operators, i.e., incremental view maintenance. We also show the necessity of GPIVOT definition for efficient view maintenance as well as for the optimization of queries with even just simple pivots.

The PIVOT operator defined in [8] has slight semantic difference than the definition here, namely, if we shall keep the pivot rows with all columns $\perp$ or not. One primary difference for the incremental maintenance of such a PIVOT operator are the propagation rules in the delete case. That is, when we have a maintained pivoted tuple as $(K, \perp, ..., \perp)$, we cannot simply delete it because there might still exist other K tuples in the underlying table. One solution is to create an auxiliary view which computes $count(*)$ for each K. We delete the view tuple $(K, \perp, ..., \perp)$ only when its count of K becomes 0. Clearly, the combination rules defined in this paper can help reduce the number of such auxiliary views.

The pivot and unpivot operators studied in this paper are basically first-order, since the input/output columns are predetermined in the query. In [14], the authors propose the SchemaSQL language with FOLD and UNFOLD operators which are very similar to pivot and unpivot operators. However, these two operators are high-order since the output columns are dynamically determined. The incremental maintenance SchemaSQL views was first studied in [13]. However, the technique is primarily tuple-based and not efficient for batch updates. The resulting maintenance plan is also not ready for a query optimizer. In this paper, though we study the first-order version of such operators, we are able to derive efficient maintenance plans. It is interesting future work to extend our proposed algorithms to support the maintenance of high-order pivot and unpivot operators.

9 Conclusions

In this paper, we propose a novel framework for view maintenance with pivot and unpivot operators. We find that a generalized pivot operator, GPIVOT, not only has more powerful semantics but is also crucial for incremental view maintenance. We propose the combination rules, pullup rules and various propagation rules for GPIVOT and GUNPIVOT in order to derive an efficient maintenance plan. Extensive performance evaluations confirm the effectiveness of various update propagation rules. There are a number of promising future directions, e.g., optimization and execution of GPIVOT/GUNPIVOT in RDBMS, maintenance of source updates in order to avoid always to decompose them into inserts and deletes, maintenance of pivot that includes all null tuples, maintenance of high-order pivot and unpivot operators and query matching for such views.

Acknowledgement We would like to thank Latha S. Colby at IBM Almaden Research Center for many valuable suggestions.

References

[1] Oracle 9i. http://www.oracle.com.

[2] R. Agrawal, A. Somani, and Y. Xu. Storage and Querying of E-Commerce Data. In *Proceedings of VLDB*, pages 149–158, 2001.

[3] R. G. Bello, K. Dias, A. Downing, J. J. F. Jr., J. L. Finnerty, W. D. Norcott, H. Sun, A. Witkowski, and M. Ziauddin. Materialized Views in Oracle. In *Proceedings of VLDB*, pages 659–664, 1998.

[4] S. Chaudhuri and U. Dayal. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Record*, 26(1):65–74, 1997.

[5] S. Chaudhuri and K. Shim. Query Optimization in the Presence of Foreign Functions. In *Proceedings of VLDB*, pages 529–542, 1993.

[6] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for Deferred View Maintenance. In *Proceedings of SIGMOD*, pages 469–480, 1996.

[7] N. Colossi, W. Malloy, and B. Reinwald. Relational Extensions for OLAP. *IBM System Journal, Vol 41, No 4*, 2002.

[8] C. Cunningham, G. Graefe, and C. A. Galindo-legaria. PIVOT and UNPIVOT: Optimization and Execution Strategies in an RDBMS. In *Proceedings of VLDB*, pages 998–1009, 2004.

[9] R. Elmasri and S. B. Navathe. *Fundamental of Database Systems*. Benjamin/Cummings, 1989.

[10] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total. In *Proceedings of ICDE*, pages 152–159, 1996.

[11] T. Griffin and L. Libkin. Incremental Maintenance of Views with Duplicates. In *Proceedings of SIGMOD*, pages 328–339, 1995.

[12] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. In *Proceedings of SIGMOD*, pages 157–166, 1993.

[13] A. Koeller and E. A. Rundensteiner. Incremental Maintenance of Schema-Restructuring Views. In *Proceedings of EDBT*, pages 354–371, 2002.

[14] L. V. S. Lakshmanan, F. Sadri, and S. N. Subramanian. On Efficiently Implementing SchemaSQL on an SQL Database System. In *Proceedings of VLDB*, pages 471–482, 1999.

[15] W. Lehner, R. Sidle, H. Pirahesh, and R. Cochrane. Maintenance of Automatic Summary Tables. In *Proceedings of SIGMOD*, pages 512–513, 2000.

[16] Microsoft SQL Server (Yukon). *http://www.microsoft.com/sql/yukon*.

[17] I. Mumick, D. Quass, and B. Mumick. Maintenance of Data Cubes and Summary Tables in a Warehouse. In *Proceedings of SIGMOD*, pages 100–111, May 1997.

[18] D. Quass. Maintenance Expressions for Views with Aggregation. In *Proceedings of the Workshop on Materialized Views: Techniques and Applications*, pages 110–118, 1996.

[19] K. Salem, K. S. Beyer, R. Cochrane, and B. G. Lindsay. How To Roll a Join: Asynchronous Incremental View Maintenance. In *Proceedings of SIGMOD*, pages 129–140, 2000.

[20] TPC. Benchmark standard specification. May 1995.