Efficient Incremental View Maintenance for Data Warehousing

by

Songting Chen

A Dissertation

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

in

Computer Science

by

December 18, 2005

APPROVED:

Prof. Elke A. Rundensteiner Advisor	Prof. Murali Mani Committee Member
Prof. Carolina Ruiz Committee Member	Dr. Latha S. Colby IBM Almaden Research Center External Committee Member

Abstract

Data warehousing and on-line analytical processing (OLAP) are essential elements for decision support applications. Since most OLAP queries are complex and are often executed over huge volumes of data, the solution in practice is to employ materialized views to improve query performance. One important issue for utilizing materialized views is to maintain the view consistency upon source changes. However, most prior work focused on simple SQL views with distributive aggregate functions, such as SUM and COUNT.

This dissertation proposes to consider broader types of views than previous work. First, we study views with complex aggregate functions such as variance and regression. Such statistical functions are of great importance in practice. We propose a workarea function model and design a generic framework to tackle incremental view maintenance and answering queries using views for such functions. We have implemented this approach in a prototype system of IBM DB2. An extensive performance study shows significant performance gains by our techniques.

Second, we consider materialized views with PIVOT and UNPIVOT

operators. Such operators are widely used for OLAP applications and for querying sparse datasets. We demonstrate that the efficient maintenance of views with PIVOT and UNPIVOT operators requires more generalized operators, called GPIVOT and GUNPIVOT. We formally define and prove the query rewriting 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 performance evaluations reveal the effectiveness of our techniques.

Third, materialized views are often integrated from multiple data sources. Due to source autonomicity and dynamicity, concurrency may occur during view maintenance. We propose a generic concurrency control framework to solve such maintenance anomalies. This solution extends previous work in that it solves the anomalies under both source data and schema changes and thus achieves full source autonomicity. We have implemented this technique in a data warehouse prototype developed at WPI. The extensive performance study shows that our techniques put little extra overhead on existing concurrent data update processing techniques while allowing for this new functionality.

Acknowledgments

I would like to express my gratitude to my advisor, Prof. Elke A. Rundensteiner, for her help, advice and patience throughout my graduate studies. Without her incredible responsiveness and the time she always had for me, this dissertation would not have been achieved.

My thanks also go to the members of my Ph.D. committee, Prof. Carolina Ruiz, Prof. Murali Mani and Dr. Latha S. Colby, who provided valuable suggestions and guidance to my research topics, drafts and talks that helped greatly to improve the presentation and contents of this dissertation.

I would also like to thank Richard Sidle, Latha S. Colby, Roberta Cochrane and Hamid Pirahesh at IBM Almaden Research Center for their help and advice throughout my 1.5-year internship at IBM. I learned quite a lot from this great industrial experience. Chapter 3 in this dissertation is based on the work done during my internship there.

The friendship of Bin Liu, Xin Zhang, Andreas Koeller, Hong Su, Ming Li and all the other previous and current DSRG members is much appreciated. They have contributed to many interesting and good-spirited discus-

sions related to this research.

Last, I would like to thank my wife Lin Qiao for her understanding and love during the past few years. Her support and encouragement was in the end what made this dissertation possible. My parents receive my deepest gratitude and love for their dedication and many years of support during my studies.

Contents

List of Figures

Chapter 1

Introduction

1.1 Data Warehouse and OLAP

The amount of information available to today's large-scale enterprises has been growing explosively. New data are being rapidly and continuously generated by various operational sources, such as auction databases and order processing systems. In order to make intelligent business decisions, complex analytical queries will be issued and answered across all data sources [CD97]. Since the modern data sources are becoming increasingly heterogeneous and are often distributed over a large network, it is often preferred that their data have to be extracted, transformed and loaded (ETL) [CD97] before complex analytical queries can be executed. Such ETL processes are very expensive.

Data warehouses are thus proposed for efficient support of such on-line analytical processing (OLAP) [CD97, Wid95]. A data warehouse extracts and integrates data from independent data sources and then stores the in-

tegrated data in a central database. Figure 1.1 depicts a typical data warehouse architecture. Here the warehouse data is extracted from multiple operational databases, external sources or legacy sources. The extracted data is often further aggregated under different granularities in order to provide a summary of the underlying data. Various front-end tools have been developed to analyze such summary data for decision making, such as query reporting, analysis and data mining [CD97].

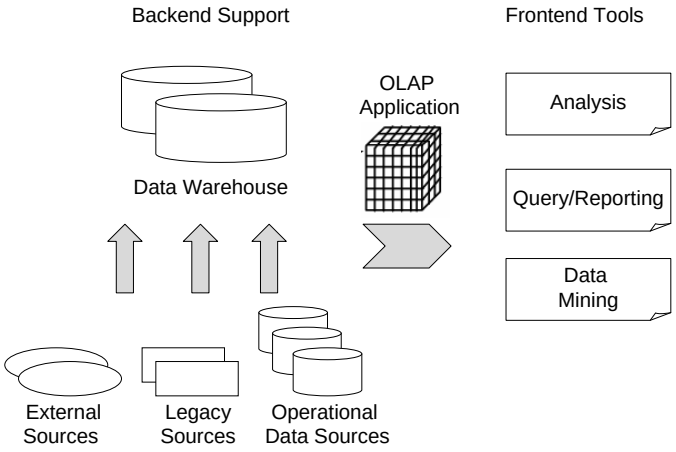


Figure 1.1: Data Warehouse Architecture

Most analytical queries over the warehouse data are fairly complex involving multiple joins and aggregations [ZCL+00]. Such queries are typically operated over huge volumes of data (many terabytes). Furthermore, most of these queries require interactive response, i.e., response time typically in a few seconds [ZCL+00]. Traditional query optimization techniques often fail to meet such new requirements. The solution in practice is to create *materialized views* to vastly improve the query performance. Materialized views pre-compute and store the query results physically in order

to avoid repetitive computation. Experience with TPC [TPC95] and decision support applications demonstrates that tremendous performance gain can be achieved by using materialized views to answer complex queries [ZCL+00].

1.2 Materialized Views

Extensive prior research [LMSS95, AESY97, CM77, CNS03, CGL+96, GL95, GRT99, GT00, LMS95, SY81, SBCL00, SDJL96, ZCL+00, ZGMHW95] has been conducted in the area of materialized views due to its importance to data integration and warehouse applications.

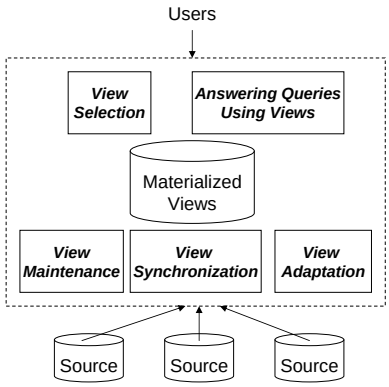


Figure 1.2: Overview of View Management Tasks

Figure 1.2 gives a big picture of the five most important view management tasks, namely, *view maintenance*, *answering queries using views*, *view selection*, *view synchronization* and *view adaptation*. View maintenance incrementally maintains the view consistency upon source data changes. View synchronization rewrites the view definition upon source schema changes

in order to keep the view definition consistent with the source schema. View adaptation incrementally adapts the view content after the view definition is rewritten. Answering queries using views is to automatically rewrite user queries using views. View selection is to pre-select a set of views to materialize (possibly under a size constraint) in order to maximize the query performance. In the next section, we will give an overview of these view management techniques.

1.2.1 State-of-the-art View Management Techniques

View Maintenance. Since materialized views correspond to pre-computed and stored query results, they may become out-of-date when the underlying sources are changed. Hence, one important issue is to maintain the materialized views' consistency upon any source changes. While re-computing views from scratch in response to any source updates may be acceptable for some relatively static databases, it is unaffordable when the source changes are frequent. Hence, *incremental view maintenance*, as an efficient alternative, [GMS93, LMSS95, AESY97, CGL$^+$96, GL95, SBCL00, ZGMHW95] has been proposed and extensively studied.

We can classify the existing view maintenance algorithms into two categories, namely, *algorithmic* [GMS93, BLT86, KR81, CW91] and *algebraic* [GL95, Qua96, GK98, KR02].

Given a view and source updates, *algorithmic* view maintenance algorithms derive *a program* (a program can be a collection of deductive rules or SQL statements) whose evaluation maintains the view. [KR81] presents the first proposal, the *finite differencing* algorithm, for incremental view main-

tenance under a functional data model. The output of the maintenance algorithm adds several lines of code into the source update transaction in order to also update the view. [CW91] proposes to maintain the view by automatically deriving a set of production rules (active rules) [WCL91]. It assumes set semantics of all base tables and a key is required to exist in the view. [GMS93] proposes a *counting algorithm* for maintaining views under bag semantics. It essentially keeps track of the multiplicity of each view tuple, or in other words, the number of derivations of each view tuple. The insert deltas have a positive count while the delete deltas have a negative count. A view tuple is deleted from the view if its count becomes 0.

The main issues with algorithmic view maintenance algorithms are that (1) the correctness of the algorithms is hard to prove, especially when the view language is extended, it is unclear and hard to prove if the existing algorithms will still work; (2) the output of maintenance algorithms (a program as mentioned above) is also hard to optimize.

Hence *algebraic* solutions have been proposed to address these limitations. More specifically, an algebraic approach [GL95] pre-defines a set of primitive change propagation rules for each operator. The maintenance plan can then be constructed by propagating changes through each algebra operator in the view query algebra tree and recursively applying those primitive rules. The output of such algorithms, namely, the maintenance plan, can be optimized by a cost-based query optimizer. Also since the algorithm is algebra-based, the result is not tied to any particular query language.

Due to these benefits mentioned above, algebraic view maintenance al-

gorithms have been extensively explored. Most existing work builds upon such an algebraic maintenance framework by considering more types of operators or considering different underlying data models. [QW91] studies the join view maintenance under set semantics. [GL95] extends that work to assume bag algebra. [Qua96, GK98] also consider group-by and outer-join operators. [KR02] studies the *higher-order operators* for maintaining SchemaSQL views [LSS99]. Algebra-based maintenance work has also been studied beyond the relational data model, e.g., [DESR03] maintains XQuery views based on an XML algebra [ZPR02]. All the work above clearly reflects the extensibility of such an algebraic maintenance framework. Such extensibility lies in the fact that for each algebra operator, its change propagation is independent of its application context. Hence we can reuse the existing change propagation rules for the same operator in more complex language constructs.

Finally, incremental view maintenance techniques are applicable to many other applications, such as trigger/constraint processing [CW91], cache/replica maintenance [GLRG04], to name a few. The view self-maintenance problem [QGMW96] can also be considered as an application of the view maintenance techniques. That is, given a view and source updates, after generating the maintenance plan, we can easily determine if we need to query the sources or not. Some recent emerging applications, such as *continuous query* processing [LPT99] over data streams, are also closely related to the incremental view maintenance techniques.

View Synchronization. Schema evolution is unavoidable for many modern applications [LNR02, VMP03, YP05], since the business requirements are typically changing rapidly, or new data types such as biological data or new constraints are emerging. Examples include the schema design evolution for data integration, physical data design evolution such as XML to relational mapping, etc. As the schema of today's data becomes increasingly complex and requires more rapid changes and growth, evolution is now a common problem. There is a growing interest to address the schema evolution issues in the research community [LNR02, VMP03, YP05].

When schema evolution occurs, the view definition must be maintained to stay consistent [LNR02, VMP03, YP05]. View Synchronization [NLR98, LNR02] aims at evolving the view definition when the schema of the base relation has been changed. In the EVE system [NLR98, LNR02], two primitive types of source schema changes that may affect the view defined upon them are considered: *RenameSC* that renames the source attributes or relations and *DropSC* that deletes attributes or relations. Note that the addition of relations or attributes does not change the views. To maintain *RenameSC*, we can simply modify the corresponding view definition by using the new names. To handle *DropSC*, the basic idea is to find some alternative source to replace the dropped data.

The schema mapping evolution techniques in [VMP03, YP05] consider more flexible semi-structured data. In that work, more types of primitive schema changes are considered. For example, beyond adding or removing elements, we can also add or remove constraints, or restructure the schema.

Finally, note that the view synchronization process does not require the

extent of the rewritten view to be always exactly equivalent to the original one. This is a reasonable assumption for information integration over a large scale and dynamic data sources [LNR02, VMP03, YP05]. Whenever this view definition rewrite is a non-equivalent one, then the next step, namely, view adaptation, becomes necessary.

View Adaptation. There are two reasons for rewriting the view definition, namely, either the user wants to explicitly update the view definition [GMR95], or the source schema changes force the evolution of the view definition [LNR02, VMP03, YP05] as mentioned above.

View adaptation has been studied under the view synchronization [LNR02] context. In that work, the view definition has to be evolved since the change of the source schema may invalidate the view definition. In [NR99], the authors propose a view adaptation technique using the same approach as in [GMR95]. That is, we pre-define a set of adaptation rules for primitive schema changes. Any complex schema change can be modeled as a sequence of primitive schema changes.

View adaptation, in principle, is a variant of the view maintenance problem [GMS93]. The difference is that in the view maintenance context, the source table changes, while in the view adaptation, the view definition itself changes. The methodology in [GMR95] is fairly close to build an algebraic framework for incremental view maintenance [QW91, GL95].

At the same time, view adaptation also has a close relationship to answering queries using views, which will be described shortly. That is, the derivation of each primitive adaptation rule is based on the equivalent

query rewriting. Intuitively, given an old view and a new view, view adaptation can be considered as a variation of the following problem: how to answer the new view from the old view.

Answering Queries using Views. Once the materialized views are created and maintained, it becomes important to utilize these views to answer user queries. *Answering queries using views* is to find efficient methods to rewrite the query using a set of previously defined materialized views instead of accessing the base relations.

The core of answering query using views consists of the following two techniques: *query containment* [CM77] and *query rewriting* [LMS95]. Query containment is to check if the view contains enough information to answer the query. Query rewriting is to rewrite the query to refer to the materialized views based on the information from query containment checking.

Both query containment and query rewriting are studied under various query languages, such as SPJ queries [CM77, SY81], queries with arithmetic comparison predicates [Klu88, LS93], queries with aggregations [GRT99, CNS03] and more recently XPATH queries [MS02]. Query rewriting has been studied for conjunctive queries [LMS95], multi-block queries [ZCL+00], aggregate queries [SDJL96, GT00], queries over semi-structured data [PV99], restructuring queries [Mil98] such as SchemaSQL [LSS96] and most recently XML queries, such as XQuery [CR05a, MS05].

Depending on the applications, there are two distinct categories of work on answering queries using views. The work in the first category has been done in the context of query optimization and maintenance of physical

data independence [LMS95, CKP95, SDJL96, GT00]. They consider only equivalent query rewriting in order to ensure the correctness of the plan. The work in the second category is for information integration in a more loosely-coupled environment [PL00, AD98, YL87]. Even partial answers are acceptable for these applications. Hence they consider not only equivalent query rewritings but also contained query rewritings.

View Selection. The view selection problem is to choose a set of views to materialize in order to achieve the best query performance for given a given query workload. Typically, view selection is under a space constraint [HRU96, Gup97], and/or a maintenance cost constraint [GM99]. In [CHS01, ZZL+04], the authors describe strategies to automatically select both views and indexes.

Unlike answering queries using views that need to handle ad-hoc queries, in view selection scenarios, the queries are known. Hence, most view selection algorithms start from identifying common sub-expressions [CHS01, Gup97] among queries. These common sub-expressions serve as the candidates of the materialized views.

One fundamental practical issue with view selection is that there are many possibly competing factors to be considered during the view selection phase, such as space, query performance, update performance, etc. This makes the view selection decision computationally challenging. Hence, most existing work uses heuristics-based solutions [HRU96, CHS01, ZZL+04]. The other difficulty is that view selection is typically a static process. Whenever the query workload changes, the previous selected views may no

longer be the best choices. Hence, incremental and/or adaptive view selection algorithms become necessary for such dynamic environments.

1.2.2 New Challenges for View Management

We note that the five view management techniques described above have been developed for specific types of views. When the view language is extended, the corresponding techniques have to be extended to support the new language constructs as well. Despite the abundance of prior work described in Section 1.2.1, we observe that the classes of views considered so far in the literature are still limited and thus not sufficient to support complex data warehousing and OLAP applications.

- First, many decision support queries typically involve complex aggregate functions for statistical modeling and trend analysis, such as variance and regression. The effective support of such queries using views is essential to the performance for such applications. However, to our knowledge, most existing work only considers simple distributive aggregate functions, such as SUM and COUNT [MQM97, Qua96, SDJL96, ZCL+00].

- Second, many decision support queries also include various ETL and OLAP operations, such as PIVOT and UNPIVOT. The effective support of these queries using views is also important to performance. However, to date, most views considered in the literature are simple SELECT-PROJECT-JOIN views with simple aggregation. Clearly, there is a need to support more powerful ETL and OLAP operations.

As we will describe in Section 1.3.1, this class of views has many applications in practice.

The other limitation is that most prior work [AESY97, ZGMHW95] assume a static environment in which the data sources' schema remains unchanged. This is however no longer a valid assumption since schema evolution is common for many modern data sources. Most online data sources nowadays are typically dynamic, autonomous and distributed over a large-scale of networks [LNR02]. How to manage the materialized views in such a dynamic environment becomes increasingly important.

In this dissertation, we will address the *view maintenance* part for these new challenges, since without effective view maintenance techniques, the refresh cost will become prohibitively expensive, making such views useless in practice.

1.3 Dissertation Objectives

1.3.1 Views with PIVOT and UNPIVOT Operators

Data in data warehouses is typically multidimensional. For example, in a sales data warehouse, the product, location and time of sales might be some of the dimensions of interest. These dimensions are often hierarchical. For example, the location dimension may be organized as a city-state-country hierarchy. Many complex OLAP operations are designed to perform online analysis on such multidimensional data, such as *drill down* or *roll up* (decrease or increase the level of aggregation) in one or more dimension

hierarchies, *slice and dice* (select one particular dimension value) and *pivot* (re-orient the multidimensional view of data) [CD97].

Relational database engines [LSPC00, Mic] 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 [GBLP96] to support multidimensional aggregation. Making such operators explicit to a relational database engine provides excellent optimization opportunities [EN89]. Another example is the inclusion of PIVOT and UNPIVOT operators into Microsoft SQL Server [CGGL04, Mic] for efficient execution and optimization.

PIVOT and UNPIVOT are frequently used OLAP operators [CD97]. Recently, these two operators have also been found to be useful for sparse data set processing. Agrawal et al. [ASX01] propose to use the *vertical* format to store sparse datasets in order to avoid a large number of columns with many NULL values. For example, in Figure 1.3, the table $ItemInfo$ stores the attributes of each auction. While there might be thousands of different item attributes, each individual item may just have a few of them. In this scenario, 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. In [ASX01], the authors suggest to store such sparse dataset in the vertical format, namely, the attribute names are treated explicitly as data values and are stored along with their corresponding attribute values in the same tuple. For example, as in Figure 1.3, the row (1,Type,TV) denotes auction 1's attribute 'Type' and the corresponding value 'TV'. The PIVOT operator transforms the vertical ta-

ble into a horizontal format. More precisely, as in Figure 1.3, 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 the data values.

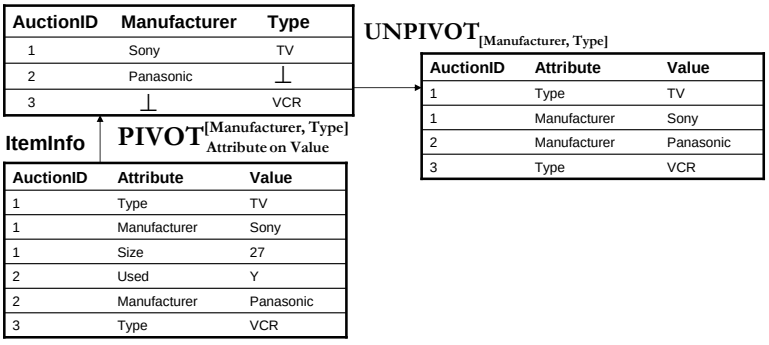


Figure 1.3: PIVOT and UNPIVOT Operators

Recent work [CGGL04, Mic] proposes to provide direct support of these two operators inside the query engine. The benefits are multi-fold [CGGL04, Mic]. For example, we can now optimize the query containing pivot/unpivot by moving these operators around the algebra tree. We can also develop various strategies for optimizing the execution of such operators. Despite these execution and optimization strategies, these operators are still potentially costly to evaluate especially when applied to huge datasets as common to data warehousing scenarios. Utilizing materialized views to answer such queries is a commonly accepted solution. However, one critical issue, the incremental maintenance of views with PIVOT and UNPIVOT operators remains unsolved.

In this dissertation, we propose a novel approach to address the above

challenges and show how to efficiently and incrementally maintain views with PIVOT and UNPIVOT operators [CR05b].

1.3.2 Views with Complex Aggregate Functions

Aggregate queries are frequently used in decision support applications. Typically huge fact tables are first joined with the dimension tables and then aggregated over various granularities with multiple statistical measurements, such as *variance* and *regression*. These aggregate functions are frequently used for statistical modeling and trend analysis. The effective support of such queries using views is essential to improve the performance for such applications. However, to our knowledge, most existing work only considers distributive aggregate functions, such as SUM and COUNT [MQM97, Qua96, SDJL96, ZCL+00].

Consider the following view definition (based on the simplified TPC-H [TPC95] schema for succinctness) which stores the regression model between price and quantity built for each customer [1]:

$$
\begin{array}{ll}
\textsf{CREATE VIEW} & SalesAnalysis \textsf{ AS} \\
\textsf{SELECT} & o\_custkey, \\
 & regr\_slope(l\_extendedprice, l\_quantity)\ as\ slp, \\
 & count(*)\ as\ cnt \\
\textsf{FROM} & lineitem,\ orders \\
\textsf{WHERE} & l\_orderkey\ =\ o\_orderkey \\
\textsf{GROUP BY} & o\_custkey
\end{array}
\tag{1.1}
$$

[1] Regr_slope is linear regression slope function, while Regr_intercept is linear regression intercept function.

The first issue is how to incrementally maintain $SalesAnalysis$ when the base tables change. The incremental maintainability of aggregate views depends on the properties of the aggregate functions. The behavior of an aggregate function with respect to incremental maintenance can be classified into one of the three categories [GBLP96]. A *distributive* function, such as *sum* and *count*, can be computed using only the existing value and the delta value under both inserts and deletes. An *algebraic* function, such as *variance* and *regression*, can be incrementally computed with the aid of some additional functions. A *holistic* function, such as *median*, cannot be incrementally computed using finite-sized storage.

Most existing work on aggregate view maintenance [MQM97, Qua96] and view matching [SDJL96, ZCL+00] only focus on distributive functions. However, most useful statistical functions fall into the category of *algebraic* functions, such as *variance* or *regression* in the above example. Views with such algebraic functions have not been sufficiently studied in the literature.

The second issue is how to answer queries using such views, e.g., how to answer the following query using *SalesAnalysis*.

$$
\begin{array}{ll}
\textsf{SELECT} & c\_nationkey, \\
 & regr\_slope(l\_extendedprice, l\_quantity), \\
 & regr\_intercept(l\_extendedprice, l\_quantity) \\
\textsf{FROM} & customer,\ lineitem,\ orders \\
\textsf{WHERE} & l\_orderkey = o\_orderkey\ AND \\
 & o\_custkey = c\_custkey \\
\textsf{GROUP BY} & c\_nationkey
\end{array}
\tag{1.2}
$$

In order to achieve this, we will need effective mechanisms for how to

compute the *regr_intercept* in the query using the *regr_slope* in the view.

The main challenge in addressing these issues is that we need a generic solution to support all these functions rather than a separate hard-coded solution for each one of them. The reason is that such functions are abundant in practice, such as *correlation, covariance, skewness, kurtosis* or even *user-defined aggregate functions* [WZ00]. A generic solution would greatly increase the system's extensibility to add the support for new functions while ensuring correctness.

In this dissertation, we propose a workarea function model and design a generic framework to address the above issues using our model [CSC+03, CCPS03].

1.3.3 Views over Dynamic and Autonomous Sources

In dynamic environments like the WWW, the data sources may change their schema, semantics as well as their query capabilities. In correspondence, the mapping or view definition must be maintained to be kept consistent [LNR02, VMP03]. Moreover, in a loosely coupled environment, such as the Data Grid [JR03], the data sources are typically owned by different providers and function independently from one another. Hence they may commit update transactions without any concern about how those changes may affect the mappings or views defined upon them. Such autonomous source schema restructuring poses new challenges for data integration.

As we will illustrate via examples in Example 1, when maintaining a source update, we may need to query the data sources for more information by issuing *maintenance queries* [ZGMHW95]. However, in these new

autonomous and dynamic environments, such queries may either return erroneous results due to concurrent data updates or may even fail completely due to concurrent schema changes.

Example 1 *Assume we want to integrate data from the book store and library category to provide the user the sales as well as the detailed book information (Figure 1.4). The book Retailer data, being in the XML format, is mapped into the relational table StoreItems as a relational wrapper table. The Library catalog of the detailed book information can be accessed by a general-purpose wrapper, which is used to execute a query and extract source changes to notify the view manager. Now the integrated view BookInfo from both data sources can be defined by the SQL query in Query (1.3).*

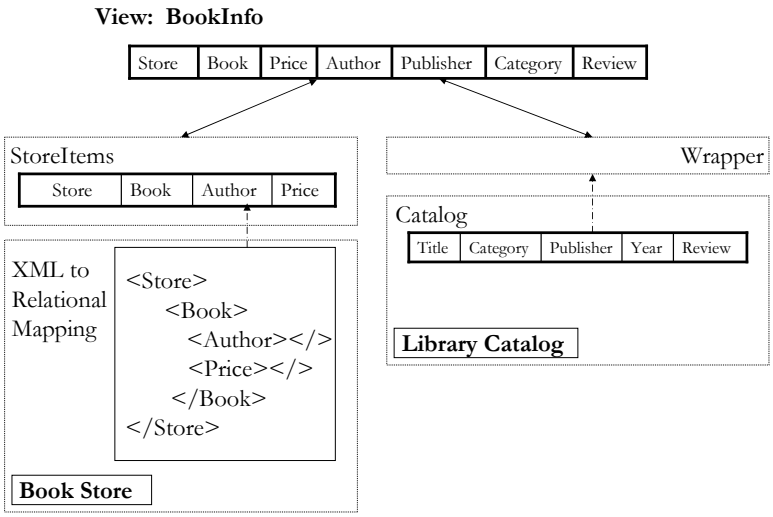


Figure 1.4: Description of View and Data Sources

Now assume a new book is inserted into the Library catalog. This new book is extracted by the wrapper as "ΔC =('Data Integration Guide', 'Adams', 'Engineering', 'Princeton' ...)". To determine its delta effect on the view, an incremental

maintenance query (Query (1.4)) [ZGMHW95] will be generated by decomposing the view query (1.3) into individual source queries. Two different anomalies can be distinguished:

```
CREATE VIEW  BookInfo AS
SELECT       Store, Book, Author, Price,
             Publisher, Category, Review
FROM         StoreItems S, Catalog C
WHERE        S.Book = C.Title
```

(1.3)

```
SELECT  Store, Book, Author, Price
FROM    StoreItems S
WHERE   Book = 'Data Integration Guide'
```

(1.4)

(a) **Data Update Anomaly***: Assume that before the execution of Query (1.4), the* StoreItems *table committed a data update ΔS = insert ('Amazon', 'Data Integration Guide', 'Adams', 35.99). This new tuple would be included in the query result of Query (1.4). Thus one final tuple ('Amazon', 'Data Integration Guide', 35.99, 'Adams', 'Princeton', 'Engineering', ...) will be inserted into the view. However, later when we maintain the view based on ΔS, the same tuple would be inserted into the view again. A duplication anomaly occurs due to concurrent data updates [ZGMHW95].*

(b) **Broken Query Anomaly***: Now assume an alternative XML-Relational mapping has been chosen in BookStore database as shown in Figure 1.5. That is, the StoreItems table is normalized into two tables, namely, Item and Store. Then Query (1.4) faces a schema conflict and cannot succeed since StoreItems table is no longer available.*

While recent work [AESY97, ZGMHW95] has proposed *compensation-based* solutions to remove the effect of concurrent data updates from query results, clearly these existing solutions fail under source schema changes.

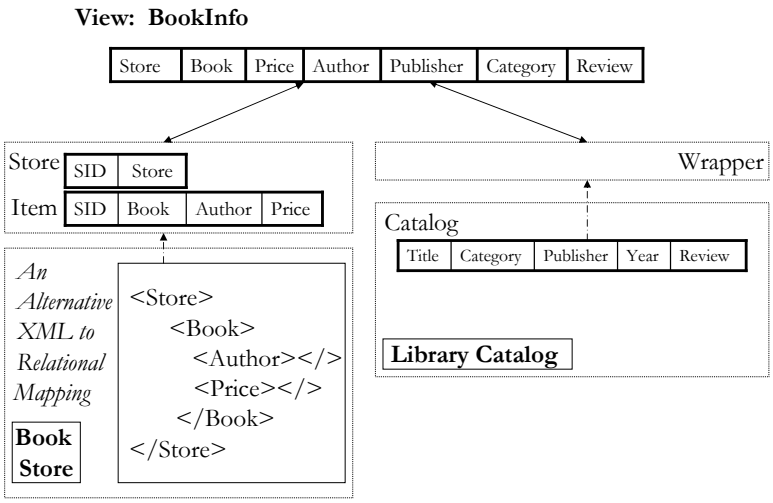


Figure 1.5: Change of Store Schema

The reason is that if the source schema has been concurrently changed, neither maintenance nor compensation queries would get any query response due to the discrepancy of the source schema with the schema required by the queries. Interleaving of concurrent source data and schema changes even complicates the maintenance further.

In this dissertation, we propose a general concurrency control strategy to solve all types of anomalies mentioned above even in a mixed fashion [CZC+01, CCZ+04, CLR04].

1.4 Outline

The rest of the dissertation is organized as follows. Chapter 2 describes the techniques for maintaining views with pivot and unpivot operators. Chapter 3 presents a generic framework for how to manage views with complex

aggregate functions. Chapter 4 introduces the maintenance of views in dynamic and autonomous environments. Chapter 5 summarizes this dissertation and discusses possible future work.

Chapter 2

Views with PIVOT and UNPIVOT Operators

2.1 Our Contributions

In this work, we propose a systematic approach for efficient incremental maintenance of views with PIVOT and UNPIVOT operators, as motivated in Section 1.3.1. Our solution is based on the algebraic view maintenance framework designed in [GL95]. The benefits of tackling the incremental view maintenance at the algebraic level are many-fold. First the techniques are not tied to any particular query language. Second, the correctness of the solution can easily be shown. Third, the result, namely, the maintenance plan, can be optimized by a cost-based optimizer.

In summary, the main contributions of this part of the dissertation are:

- We propose a novel framework for efficient incremental maintenance

of views with PIVOT and UNPIVOT operators. We demonstrate that the transformation of the view query is a necessary step in order to obtain an efficient maintenance plan. Such query transformation has not been considered as a prerequisite step for efficient view maintenance in prior work.

- We propose to transform the query into a *top-heavy* shape for efficient view maintenance. That is, there is only one PIVOT in the query and that PIVOT resides on top of the query algebra tree as its root. For this, we propose a *generalized pivot operator* GPIVOT that can combine multiple PIVOT operators in order to eventually keep only one GPIVOT in the query. Note that such GPIVOT operator also has more powerful semantics than prior operators in the literature [CGGL04, ASX01, LSS99]. We formally define these *combination rules* and establish the proof for the correctness as well as the completeness of these rules.

- We formally define the *swapping rules* for GPIVOT and its reverse operator GUNPIVOT in order to move these operators up and down in the query tree. We prove the correctness as well as the completeness of these swapping rules. The combination rules and the swapping rules, both classes of *query equivalence rules*, are useful not only for view maintenance and but also for query optimization.

- We propose the *propagation rules* for GPIVOT and GUNPIVOT. These rules are needed for the incremental maintenance of views with such operators. Our techniques preserve the closure, that is, they generate

a maintenance plan in the form of again an algebra tree (with the same types of operators). Hence such plan can be optimized by a cost-based optimizer using our proposed query equivalence rules.

- We demonstrate that these propagation rules may still generate an inefficient maintenance plan when other operators exist in the view query, such as SELECT and GROUPBY. We design special-purpose *combined propagation rules* for multiple operators as one atomic unit in order to derive a more efficient maintenance plan. We also formally prove the correctness of these propagation rules.

- An extensive performance evaluation is conducted on top of a commercial database. The experimental results confirm that 1) query transformation does help to generate a more efficient maintenance plan, and 2) combined propagation rules for multiple operators do generate a more efficient maintenance plan than considering these operators separately.

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. Overall, our solution fits nicely into the existing maintenance framework for aggregate views shared by current commercial database engines [LSPC00, MQM97]. 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 any future query engine.

The organization of the rest of the chapter is as follows. Section 2.2 studies the basic propagation rules for pivot. Section 2.3 presents the overview of our proposed solution for view maintenance. We define the GPIVOT and GUNPIVOT operators and the combination rules in Section 2.4. The swapping rules for GPIVOT and GUNPIVOT are described in Section 2.5. 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 2.6. Section 2.7 presents the results of our performance study, while Section 2.8 reviews the related work.

2.2 Basics on PIVOT and UNPIVOT

2.2.1 Notations

A domain D is a set of values. A value can be of primitive data types, such as a string, a number, etc. D_N is a special value domain that represents $names$. We assume that there is a bijective mapping between domain D and domain D_N. This means that all values can be converted to names and vice versa. In this work, we use words in capital letters, e.g., A and B, to denote values in domain D_N. We use words in small letters, e.g., a and b, to denote values in domain D. Furthermore, we use " " to denote the conversion of values between these two domains. For instance, $A \in D_N$ and $a \in D$, while "A" $\in D$ and "a" $\in D_N$.

A relation or a table is defined as $(M, S, \{(a_1^1, ..., a_n^1), ..., (a_1^m, ..., a_n^m)\})$, where $M \in D_N$ denotes its relation name, $S = (A_1, ..., A_n) \in (D_N)^n$ denotes its n attribute or column names and $\{(a_1^1, ..., a_n^1), ..., (a_1^m, ..., a_n^m)\}$

$\subseteq D^n$ denotes its relation extent. Each $(a_1^j, a_2^j, ..., a_n^j)$, $j = 1..m$, is called a tuple. Based on the above definition, a table has n columns. Each of them has name A_i and has values $\{a_i^1, ..., a_i^m\}$, $i = 1..n$.

In this work, we adopt the traditional definition for relational operators, such as SELECT (σ), PROJECT (π), SET PROJECT (δ, i.e., select distinct), RENAME (ρ), NATURAL JOIN ($\bowtie$), SEMI-JOIN ($\rhd\!\!<$), ANTI SEMI-JOIN ($\bar{\rhd}\!\!\!\!<$), LEFT OUTER-JOIN ($\sqsupset\!\!\bowtie$), FULL OUTER-JOIN ($\sqsupset\!\!\bowtie\!\!\sqsubset$) and GROUP-BY ($\mathcal{F}$) [Ull89, EN89]. Note that the definition of the relational operators typically contains both values and names [Ull89, EN89]. Take the PROJECT operator for example. Given a relation $(M, (A_1, ..., A_n), \{(a_1^1, ..., a_n^1), ..., (a_1^m, ..., a_n^m)\})$, we have $\pi_{A_i}(M) = \{(a_i^1), ..., (a_i^m)\}$. Since in this work, we do not modify the definitions of any of these well-known relational operators, the readers are referred to [Ull89, EN89] for their definitions. Finally, Table 2.1 summarizes the notations that will be used throughout this chapter.

2.2.2 PIVOT and UNPIVOT Operators

We now define the PIVOT and UNPIVOT operators [1]. Assume V is a table with the attributes (K, A, B), where K denotes possibly multiple columns and A, B are one column each. The PIVOT operator is defined in Equation (1). It takes columns A and B as input parameters and $[a_1, ..., a_n]$ as output parameters, where each a_i $(i = 1..n)$ is a value of column A. Here $\rho_{(\text{“}a_i\text{”})}(B)$ renames the output column name from B to “a_i”. $\sqsupset\!\!\bowtie\!\!\sqsubset_{i=1}^{n}$ is an abbreviation of full outer-joins over n input tables as in Table 2.1. Such

[1] Except for the NULL handling, the PIVOT and UNPIVOT operators defined in this work are similar to v2h/h2v [ASX01], FOLD/UNFOLD [LSS99] and pivot/unpivot in [CGGL04].

Notations	Meaning
V	Vertical table
H	Horizontal table
$A, B, K, A_1, ..., A_n$	Column names
(K, A, B)	Columns of one table
$a, b, k, a_1, ..., a_n$	Column values
$(a_1, a_2, ..., a_n)$	One tuple
$\{(a_1^i, a_2^i, ..., a_n^i)\}$	A set of tuples
$\bowtie_{i=1}^{p} \{(a_1^i, a_2^i, ..., a_n^i)\}$	Pairwise binary join from $(a_1^1, a_2^1, ..., a_n^1)$ to $(a_1^p, a_2^p, ..., a_n^p)$
$\sqsupset\!\bowtie\!\sqsubset_{i=1}^{p}\{(a_1^i, a_2^i, ..., a_n^i)\}$	Pairwise full outer-join from $(a_1^1, a_2^1, ..., a_n^1)$ to $(a_1^p, a_2^p, ..., a_n^p)$
$\perp$	Empty value, similar to NULL

Table 2.1: Notations

abbreviations will also be used for JOIN and UNION in the rest of this chapter.

$$\text{PIVOT}_{A \text{ on } B}^{[a_1,...,a_n]}(V) = [\sqsupset\!\bowtie\!\sqsubset_{i=1}^{n} \pi_{K,\ \rho_{(\text{“}a_i\text{”})}(B)}(\sigma_{A=a_i}(V))] \quad (1)$$

For a particular K value k, the full outer-join is used to find the rows with $K = k$ and A being any of $\{a_1, ..., a_n\}$. Note that there must exist at most one row that satisfies $K = k \wedge A = a_i$, since A and K together form the key. If there does not exist such a row with $K = k$ and $A = a_i$, then this missing value will be denoted as '$\perp$'.

An example of PIVOT is depicted in Figure 1.3. Note that in order for the results to be meaningful, the set of columns K and column A together must form the *key* of table V [CGGL04]. Then the key for the pivoted output

table is K. Otherwise, take Figure 1.3 for example. Assume there is another row (1,type,VCR) in the ItemInfo table, i.e., *(AuctionID, Attribute)* no longer forms a key, then we have confusion of auction 1's type information since two values in this case, TV and VCR, cannot be put into the same output column.

Now we assume that the table H has the attributes $(K, A_1, ..., A_n)$, where K denotes possibly multiple columns and each A_i is one column, respectively. UNPIVOT is defined in Equation (2) with columns $[A_1, ..., A_n]$ as the input parameters. Note that here K need not be the key of table H for the applicability of UNPIVOT [CGGL04].

$$\text{UNPIVOT}_{[A_1,...,A_n]}(H) = [\cup_{i=1}^{n} \pi_{K,\text{“}A_i\text{”},A_i}(\sigma_{A_i \neq \perp}(H))] \quad (2)$$

Here $\pi_{\text{“}A_i\text{”}}$ creates a column with constant values "A_i". One example of UNPIVOT is given in Figure 1.3, where the column names, namely, Manufacturer and Type, are converted into data values.

Figure 2.1 depicts an example view composed of the traditional relational algebra and pivot operators. In this example, the vertical table Payment stores the different types of payment information. It is first pivoted to get 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 $\mathcal{F}$ [EN89] to specify the group-by operator where the columns to group on are (Manu, Type) and the aggregation lists are (sum(Credit), sum(ByAir)). The aggregate results are pivoted again in order to provide a crosstab view

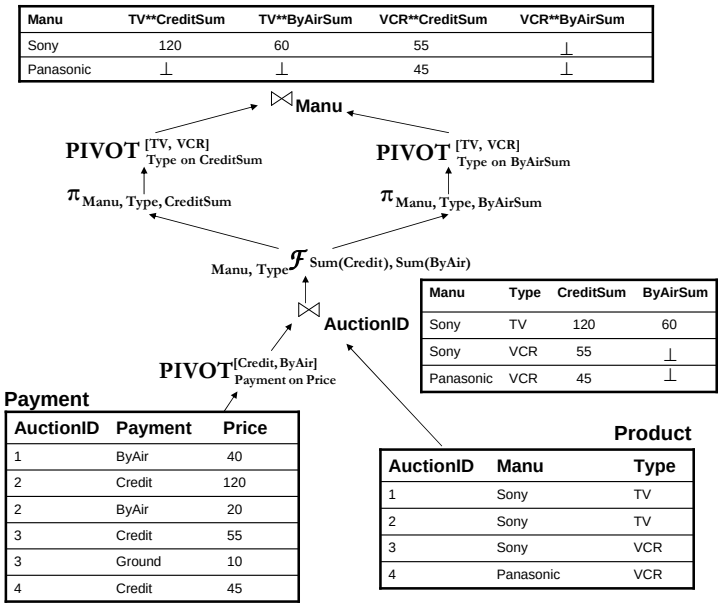


Figure 2.1: A Sample ROLAP View

of the summary data. That is, for each manufacturer, we report its TV and VCR's Credit and ByAir payments by the left and right PIVOT operators and then join their respective results. We will show in the rest of this chapter a strategy for generating an efficient maintenance plan for complex ROLAP views such as this one.

2.2.3 Basic Propagation Rules

As a first step to tackle the incremental maintenance of views with PIVOT and UNPIVOT operators, we study the propagation rules for these operators. Note that PIVOT is very similar to group-by. Actually it can be simulated using a group-by query [CGGL04] (as we will show in Section 2.7). Hence the propagation rules for PIVOT are very similar to the ones for

group-by [Qua96].

Figure 2.2 depicts an intuitive example to show how to propagate changes through the PIVOT operator [2]. Assume three tuples were inserted into the *ItemInfo* table. The first *insert/delete propagation rules*, intuitively, delete the old output tuples affected by the source inserts and insert the new output tuples introduced by the source inserts. In this example, we remove the old output tuples $(2, Panasonic, \perp)$ and $(3, \perp, VCR)$, and insert the new output tuples $(2, Panasonic, DVD)$, $(3, Panasonic, VCR)$ and $(4, \perp, DVD)$. Such new output tuples are generated by combining the old output tuple and the delta tuples, i.e., $PIVOT(I) \bowtie PIVOT(\Delta I)$.

The second *update propagation rules*, intuitively, 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 delta tuples will be *inserted* and the matched view tuples will be *updated* (by SQL Update statement). In this example, we update the old output tuples from $(2, Panasonic, \perp)$ and $(3, \perp, VCR)$ to $(2, Panasonic, DVD)$, $(3, Panasonic, VCR)$ and insert a new output tuple $(4, \perp, DVD)$. Note that there are also two types of rules for group-by that are very similar in flavor [Qua96].

2.2.4 Discussion of Propagation Rules

We note that both propagation rules in Figure 2.2 access the original pivoted result, PIVOT(I). If the PIVOT operator is an intermediate operator in the query plan, then re-evaluating this intermediate result PIVOT(I) or

[2] The propagation rules for unpivot are relatively simple and will be discussed in Section 2.6, together with the detailed formalism.

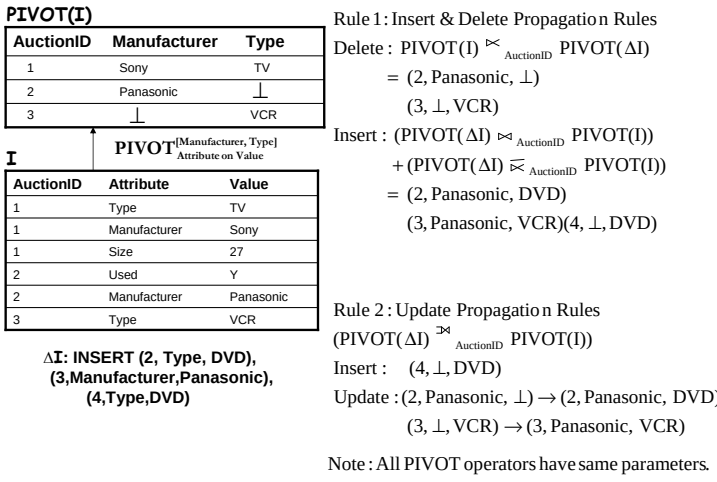


Figure 2.2: Example for Propagating Changes through PIVOT

even just partially re-evaluating it by predicate pushdown could still be fairly 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 safely assume that PIVOT(I) is available and accessible. Instead we could perform a join between the delta and the materialized view itself. Hence, these propagation rules clearly would become more efficient when the PIVOT operator is the *last* operator in the query plan.

Furthermore, even when the PIVOT operator 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 2.2, (2, Panasonic, ⊥) and (3, ⊥, VCR) are deleted from the view and then re-inserted into the view as (2, Panasonic, DVD) and (3, Panasonic, VCR). In comparison, the update rules make in-place changes of these rows by an

SQL update statement. Such deletion and then re-insertion generally introduces more CPU and I/O costs than the update approach.

Based on the observations above, we conclude that in order to derive an efficient maintenance plan: (1) the PIVOT operator should be preferably 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 [Qua96]. For example, the propagation rules for GROUPBY can also use either insert/delete or update operations. The update propagation rules are preferable and in fact are the ones incorporated into many commercial systems [BDD+98, LSPC00]. 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. However, existing work simply restricts the view definition to be SELECT-PROJECT-JOIN with GROUPBY as the last operator [BDD+98, LSPC00, MQM97]. In this work, we lift such restrictions and focus on how to efficiently maintain general views with PIVOT and UNPIVOT operators.

2.3 Solution Overview

As we consider general views with PIVOT and UNPIVOT operators in this work, there might be multiple PIVOT operators in a given query algebra tree (see the example query in Figure 2.1). As discussed earlier, except for the top PIVOT in the tree, other intermediate PIVOT operators may not propagate changes efficiently. Hence, as shown in Figure 2.3, 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. Note that if there are multiple PIVOT operators in the query, then at least one of them cannot be on the top of the query tree. Hence we propose to combine them into a new **single extended** PIVOT operator with more powerful semantics. We call this the *Generalized PIVOT (GPIVOT)*.

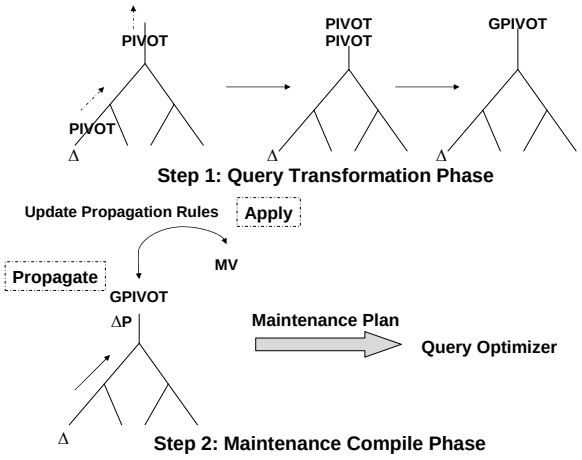


Figure 2.3: Solution Overview

The second step is to construct the maintenance plan based on the transformed query algebra tree. The resulting maintenance plan is also in the form of a query algebra tree and logically consists of two phases, namely, *propagate phase* and *apply phase* as described below.

Assume a materialized view MV is defined as $GPIVOT(P)$ as shown in Figure 2.3. The *propagate phase* propagates the source deltas up the query tree using existing propagation rules for relational operators [GL95]. Assume ΔP is the resulting maintenance plan for the sub-query P. The final

maintenance plan for the *propagate phase* is $GPIVOT(\Delta P)$, which is commonly referred to as the *final delta* [MQM97]. The *apply phase* applies the update propagation rules (Figure 2.2) for the top GPIVOT operator by integrating the *final delta* from *propagate phase*, namely, $GPIVOT(\Delta P)$ into the view MV (i.e., $GPIVOT(P)$). Finally, the resulting maintenance plan consists of the sub-plans for both *propagate phase* and *apply phase*, and can be optimized by a cost-based optimizer before execution.

Together, this two-phase processing of *propagate* and *apply* phases fits nicely into the traditional aggregate view maintenance framework [LSPC00, MQM97]. This makes our solution easily integrable into existing systems.

2.4 GPIVOT and GUNPIVOT: Generalized PIVOT and UNPIVOT

2.4.1 Definition of GPIVOT and GUNPIVOT

In this section, we will first describe how to combine multiple PIVOT operators. We will show that the resulting 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 given in Equation (3). Here we assume that the input table V has attributes $(K, A_1, A_2, ..., A_m, B_1, B_2, ..., B_n)$, where K denotes possibly multiple columns, A_i and B_j ($i = 1..m$, $j = 1..n$) denote one column each. Note that this input table schema V will be used in the rest of the chapter for GPIVOT. $(K, A_1, ..., A_m)$ must form a key for pivot applicability. Compared to the simple PIVOT op-

erator, the input parameters for GPIVOT involve multiple columns, namely, $[A_1, ..., A_m]$ and $[B_1, ..., B_n]$. The output parameters for GPIVOT are $[(a_1^1, ..., a_m^1), ..., (a_1^p, ..., a_m^p)]$, where each $(a_1^i, ..., a_m^i)$ is a tuple for columns $(A_1, ..., A_m)$.

$$\begin{aligned}&\text{GPIVOT}^{[(a_1^1,...,a_m^1),...,(a_1^p,...,a_m^p)]}_{[A_1,...,A_m]\ on\ [B_1,...,B_n]}(V)\\ &= [\exists\!\!\!\!\subset^{p}_{i=1}\pi_{K,\ \rho_{(\text{“}a_1^i**...**a_m^i**B_1\text{”})}B_1,...,\ \rho_{(\text{“}a_1^i**...**a_m^i**B_n\text{”})}B_n}(\\ &\qquad\qquad \sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)}(V))]\ ^{3}\end{aligned} \tag{3}$$

An example of GPIVOT is shown in Figure 2.4. Here input parameters are $[Manufacturer, Type]$ and $[Price, Quantity]$. The output parameters are $[\{Sony, Panasonic\} \times \{TV, VCR\}]$, or in other words, any combination of these values $(Sony, TV), (Sony, VCR), (Panasonic, TV), (Panasonic, VCR)$. 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 "$a_1^i **a_2^i **...**a_m^i **B_j$" (alternatively, we can use a separate table to store such column name information). Note that the GPIVOT operator is able to pivot *multiple* measurements based on *multiple* dimensions, a rather common and highly useful operation [CMR02] for multi-dimensional databases.

The GUNPIVOT operator is designed to decode the column names in the reverse way (alternatively we may extract names from a separate table mentioned before). Its formal definition is in 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

[3]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.

Country	Sony **TV **Price	Sony **TV **Quantity	Sony **VCR **Price	Sony **VCR **Quantity	Panasonic **TV **Price	Panasonic **TV **Quantity	Panasonic **VCR **Price	Panasonic **VCR **Quantity
USA	250	50	50	40	220	20	45	30
Japan	260	10	60	60	240	70	55	90

GPIVOT [{Sony, Panasonic}×{TV, VCR}] [Manufacturer, Type] on [Price, Quantity]

GUNPIVOT [(Sony**TV**Price, Sony**TV**Quantity), (Sony**VCR**Price, Sony**VCR**Quantity), (Panasonic**TV**Price, Panasonic**TV**Quantity), (Panasonic**VCR**Price, Panasonic**VCR**Quantity)]

Country	Manufacturer	Type	Price	Quantity
USA	Sony	TV	250	50
USA	Sony	VCR	50	40
USA	Panasonic	TV	220	20
USA	Panasonic	VCR	45	30
Japan	Sony	TV	260	10
Japan	Sony	VCR	60	60
Japan	Panasonic	TV	240	70
Japan	Panasonic	VCR	55	90

Figure 2.4: Example for GPIVOT and GUNPIVOT

each "$a_1^i **...a_m^i **B_j$" is one column. Note that this input table schema H will be used in the rest of the chapter for GUNPIVOT. Similar to the simple UNPIVOT, here K need not to be the key of table H in order for the applicability of GUNPIVOT. Nonetheless, its column names must all conform to the same structure. One example is in Figure 2.4.

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

Since PIVOT is a special case of GPIVOT and UNPIVOT a special case

of GUNPIVOT, in the rest of this chapter we will only consider GPIVOT and GUNPIVOT. All our results obviously apply to PIVOT and UNPIVOT as well.

2.4.2 Combination Rules for GPIVOT

Multicolumn PIVOT. The first combination rule for GPIVOT is called *multicolumn pivot*. This rule is applicable when the two GPIVOT operators have the same parameters for the pivoting columns.

Take the view in Figure 2.1 for example. Here both the total Credit and the total ByAir payments are pivoted by first pivoting each of them individually and then joining their respective results. We propose to combine these two pivot operators into one that simply pivots both 'CreditSum' and 'ByAirSum' by 'Type' 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 in Equation (3).

$$\begin{aligned}
&\text{GPIVOT}^{[(a_1^1,\ldots,a_m^1),\ldots,(a_1^p,\ldots,a_m^p)]}_{[A_1,\ldots,A_m]\ on\ [B_1,\ldots,B_n]}(V) = \\
&\text{GPIVOT}^{[(a_1^1,\ldots,a_m^1),\ldots,(a_1^p,\ldots,a_m^p)]}_{[A_1,\ldots,A_m]\ on\ [B_1,\ldots,B_j]}(\pi_{K,A_1,\ldots,A_m,B_1,\ldots,B_j}(V)) \bowtie_K \\
&\text{GPIVOT}^{[(a_1^1,\ldots,a_m^1),\ldots,(a_1^p,\ldots,a_m^p)]}_{[A_1,\ldots,A_m]\ on\ [B_{j+1},\ldots,B_n]}(\pi_{K,A_1,\ldots,A_m,B_{j+1},\ldots,B_n}(V)) \qquad (5)
\end{aligned}$$

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

$$\begin{aligned}
&GPIVOT^{[(a_1^1,\ldots,a_m^1),\ldots,(a_1^p,\ldots,a_m^p)]}_{[A_1,\ldots,A_m]\ on\ [B_1,\ldots,B_j]}(\pi_{K,A_1,\ldots,A_m,B_1,\ldots,B_j}(V)) = \\
&\qquad \exists\!\!\!\bowtie\!\!\!\subset^p_{i=1} \pi_{K,B_1,\ldots,B_j}(\sigma_{(A_1,\ldots,A_m)=(a_1^i,\ldots,a_m^i)}(\pi_{K,A_1,\ldots,A_m,B_1,\ldots,B_j}(V))) \\
&GPIVOT^{[(a_1^1,\ldots,a_m^1),\ldots,(a_1^p,\ldots,a_m^p)]}_{[A_1,\ldots,A_m]\ on\ [B_{j+1},\ldots,B_n]}(\pi_{K,A_1,\ldots,A_m,B_{j+1},\ldots,B_n}(V)) = \\
&\qquad \exists\!\!\!\bowtie\!\!\!\subset^p_{i=1} \pi_{K,B_{j+1},\ldots,B_n}(\sigma_{(A_1,\ldots,A_m)=(a_1^i,\ldots,a_m^i)}(\pi_{K,A_1,\ldots,A_m,B_{j+1},\ldots,B_n}(V)))
\end{aligned}$$

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

In other words, we need to prove the following:

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

Our proof is based on the observation that both sides of Equation (5.1) contain a key K in its output, since K is a key to each of the join input tables and the equi-join condition is on K. Hence, in order to prove Equation (5.1), we first show that both sides output the same set of key values. Then we show that for each key value, they generate the same row.

1) Let us first consider Equation (3). It is easy to derive that the output of the GPIVOT operator contains a key value k iff there exists at least one row with $K = k$ and $(A_1, ..., A_m) = (a_1^i, ..., a_m^i)$ for any i in $[1, p]$. Based on this observation, the left side of Equation (5.1) outputs the 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) outputs the following 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 any K value $k \in \delta_K(\sigma_{(A_1,...,A_m)=(a_1^1,...,a_m^1)\vee...\vee(A_1,...,A_m)=(a_1^p,...,a_m^p)}(V))$, both sides of Equation (5.1) yield the same output tuple. In this

work, we will translate the outer-joins in Equation (5.1) to inner-joins for ease of proof, since outer-join in general is not associative [RPZ04]. For this, we define a set of rows $\{r_1, ..., r_p\}$ *as:* $r_i = \pi_{K,B_1,...,B_n}(\sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)\ AND\ K=k}(V))$. *Note that there must be* at most *one such tuple* r_i *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 set* r_i *to* $(k, \perp, ..., \perp)$. *Based on this definition, for a given* k, *the output of the right side of Equation (5.1) is* $\ltimes_{i=1}^{p} \{r_i\}$ *(an abbreviation of* $r_1 \ltimes ... \ltimes r_p$*). While the output of the left side of Equation (5.1) is* $[\ltimes_{i=1}^{p} \{\pi_{K,B_1,...,B_j}(r_i)\}] \ltimes [\ltimes_{i=1}^{p} \{\pi_{K,B_{j+1},...,B_n}(r_i)\}] = \ltimes_{i=1}^{p} \{r_i\}$ *since* $\pi_{K,B_1,...,B_j}(r_i) \ltimes \pi_{K,B_{j+1},...,B_n}(r_i) = r_i$. *Hence for any* k, *both sides of Equation (5.1) yield the same output tuple.*

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

∎

PIVOT Composition. The second rule is to combine two adjacent GPIVOT operators in the query tree. Here adjacent means that the output of one GPIVOT operator feeds into another GPIVOT operator. In this case, when *all* the pivoted output columns (i.e., all "$a_1^i * *... * *a_m^i * *B_j$" columns in Equation (3)) of the first GPIVOT are the input parameters of the second GPIVOT, we can combine these two operators into one. This may occur when the user wants to pivot the measurements by more than one dimension. One simple example is shown in Figure 2.5. 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.

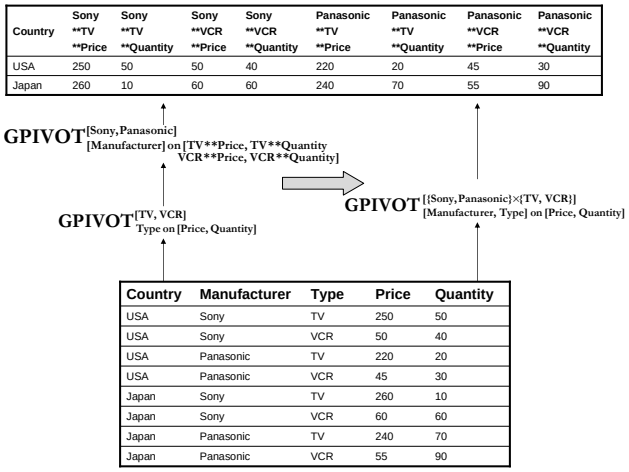


Figure 2.5: Composition of GPIVOT

Equation (6) formally defines this rule. We assume $\{(a_1^1, ..., a_l^1), ..., (a_1^p, ..., a_l^p)\}$ as the output values of $(A_1, ..., A_l)$ and $\{(a_{l+1}^1, ..., a_m^1), ..., (a_{l+1}^q, ..., a_m^q)\}$ 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., $[\{\text{“}a_{l+1}^i **..** a_m^i **B_j\text{”}\}]$, i=1,...,q and j=1,...,n, as the input parameters.

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

Proof for Equation (6): *It is easy to show that both sides of Equation (6) have a key K in its output based on the definition of GPIVOT. Next we follow a similar approach as in the proof for Equation (5), namely, we first prove that both sides*

generate the same set of key values and then prove that for any key value, both sides generate the same tuple.

1) The left side of Equation (6) outputs the key set: $\delta_K(\sigma_{[(A_1,\ ...,A_l)=(a_1^1,...,a_l^1)\ \vee...\vee(A_1,...,A_l)=(a_1^p,...,a_l^p)]\wedge\ [(A_{l+1},...,A_m)=(a_{l+1}^1,...,a_m^1)\vee...\vee(A_{l+1},...,A_m)=(a_{l+1}^q,...,a_m^q)]}(V))$.

The right side of Equation (6) outputs the key set: $\delta_K(\sigma_{(A_1,...,A_l)=\ (a_1^1,...,a_l^1)\vee...\vee\ (A_1,...,A_l)=(a_1^p,...,a_l^p)}\ (\delta_{K,A_1,...,A_l}(\sigma_{(A_{l+1},...,A_m)=\ (a_{l+1}^1,...,a_m^1)\vee...\vee(A_{l+1},...,A_m)=(a_{l+1}^q,\ ...,a_m^q)}(V))$. *By pushing down the selection, we get* $\delta_K(\sigma_{[(A_1,...,A_l)=(a_1^1,...,a_l^1)\ \vee...\vee(A_1,...,A_l)=(a_1^p,...,a_l^p)]\wedge\ [(A_{l+1},...,A_m)=(a_{l+1}^1,...,a_m^1)\ \vee...\vee(A_{l+1},...,A_m)\ =(a_{l+1}^q,...,a_m^q)]}(V))$. *Hence, both sides generate the same set of key values.*

2) Next for any K value k, we define a set of rows $\{r_{11}, r_{12}, ..., r_{pq}\}$ *as:* $r_{ij} = \pi_{K,B_1,...,B_n}(\sigma_{(A_1,...,A_l)=(a_1^i,...,a_l^i)\ AND\ (A_{l+1},...,A_n)\ =(a_{l+1}^j,...,a_n^j)\ AND\ K=k}(V))$, *where i=1,...,p and j=1,...,q. If table V does not contain any tuple that satisfies the above condition, then we set* r_{ij} *to* $(k_1, \perp, ..., \perp)$. *Based on this definition, the output tuple for value k of the left side of Equation (6) is* $\bowtie\{r_{ij}\}$, *i.e.,* $r_{11} \bowtie r_{12} \bowtie ... \bowtie r_{p_1p_2}$.

For the right side of Equation (6), we have:

$$GPIVOT^{[\{(a_1^1,...,a_l^1),...,(a_1^p,...,a_l^p)\}]}_{[A_1,...,A_l]\ on\ [\{\text{“}a_{l+1}^i**..**a_m^i**B_j\text{”}\}]}($$
$$GPIVOT^{[\{(a_{l+1}^1,...,a_m^1),...,(a_{l+1}^q,...,a_m^q)\}]}_{[A_{l+1},...,A_m]\ on\ [B_1,...,B_n]}(V)) =$$
$$\rtimes^p_{g=1}\pi_{K,\{\text{“}a_{l+1}^i**..**a_m^i**B_j\text{”}\}}(\sigma_{(A_1,...,A_l)=(a_1^g,...,a_l^g)}$$
$$[\rtimes^q_{h=1}\pi_{K,A_1,...,A_l,B_1,...,B_n}(\sigma_{(A_{l+1},...,A_m)=(a_{l+1}^h,...,a_m^h)}(V))]).$$

By pushing down the first selection, we get:

$$\rtimes^p_{g=1}\pi_{K,\{\text{“}a_{l+1}^i**..**a_m^i**B_j\text{”}\}}([\rtimes^q_{h=1}\pi_{K,A_1,...,A_l,B_1,...,B_n}($$
$$\sigma_{(A_1,...,A_l)=(a_1^g,...,a_l^g)\ AND\ (A_{l+1},...,A_n)=(a_{l+1}^h,...,a_n^h)}(V))]).$$

Hence, for a given K value k, the output tuple is $\bowtie^p_{g=1}\ [\bowtie^q_{h=1}\ r_{gh}] = r_{11} \bowtie r_{12} \bowtie$

... $\bowtie r_{pq}$. 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.

∎

Note that we can apply the combination rules developed in this section in the query graph in order to reduce the number of pivots. The first *multi-column pivot* combination rule increases the value columns (measurements), while the second *pivot composition* rule increases the pivoting columns (dimensions). 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, since after combination we can reduce the amount of computation.

Completeness of Combination Rules: Besides the two combination rules described above, we now show that Equation (6) is the only possible rule for combining any two adjacent GPIVOT operators. Here 'adjacent' means the output of the first GPIVOT is the input of the second GPIVOT.

Proof: *We note that if the two adjacent GPIVOT operators can be combined into one, then the following two observations must hold:*

(1) By the GPIVOT definition in Equation (3), one column may participate in either the parameters $\{A_i\}$ or $\{B_i\}$, but not both.

(2) Based on the GPIVOT definition in Equation (3), the pivoted output column names *must have the same structure, i.e., "$a_1^i * *.. * *a_m^i * *B_j$", where each a_k^i is a value of column A_k. Furthermore, for any column "$a_1^i * *.. * *a_m^i * *B_j$",*

its values *must also have the following property. That is, its values must equal to the values of column B_j in the original table, whose $(A_1, ..., A_m)$'s value equals to any $\{(a_1^i, ..., a_m^i)\}$. Hence even if a different naming method is used, all pivoted output column values must still have the same properties. To make our reasoning easier, in this work, we assume such naming is explicitly available to us.*

We now assume that the first GPIVOT outputs pivoted output columns $\{$"$a_1^i * *.. * *a_m^i * *B_j$"$\}$. We also assume that the second GPIVOT pivots columns $(X_1, ..., X_p)$ by columns $(Y_1, ..., Y_q)$. We will show that the two GPIVOT operators are not combinable under the following three cases: 1) at least one "$a_1^i * *.. * *a_m^i * *B_j$" $\notin \{X_1, ..., X_p\}$ and "$a_1^i * *.. * *a_m^i * *B_j$" $\notin \{Y_1, ..., Y_q\}$; 2) at least one $a_1^{i_1} * *..* *a_m^{i_1} * *B_j \in (X_1, ..., X_p)$; 3) $\{$"$a_1^{i_1} * *..* *a_m^{i_1} * *B_j$"$\} \subset \{Y_1, ..., Y_q\}$. Thus the only left case is $\{$"$a_1^{i_1} * *..* *a_m^{i_1} * *B_j$"$\} = \{Y_1, ..., Y_q\}$, which is shown in Equation (6). Figure 2.6 depicts four representative examples for two adjacent GPIVOTs in order for easy understanding of our proof.

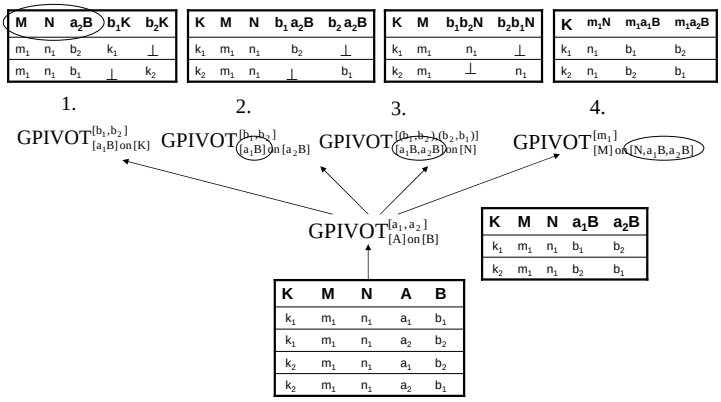


Figure 2.6: Example for Combining Two Adjacent GPIVOTs

First, we consider the case that the second GPIVOT does not use at least one

"$a_1^i * *.. * *a_m^i * *B_j$" in its input parameters, namely, "$a_1^i * *.. * *a_m^i * *B_j$" $\notin$ $(X_1, ..., X_p)$ and "$a_1^i * *.. * *a_m^i * *B_j$" $\notin (Y_1, ..., Y_q)$. In this case, in the final output, the column "$a_1^i * *.. * *a_m^i * *B_j$" cannot have the same name structure to the pivoted output columns from the second GPIVOT. This violates the observation (2). For example, in the first case of Figure 2.6, the pivoted output column a_2B is not used for the second GPIVOT and is thus not consistent with its output columns, such as b_1K and b_2K, in terms of their name structure, which also indicates that their column values cannot have the same relationship to the values in the original table.

Second, based on the result above, we know that any *"$a_1^i * *.. * *a_m^i * *B_j$" must be in either $(X_1, ..., X_p)$ or $(Y_1, ..., Y_q)$. Now assume that there is at least one "$a_1^{i_1} * *.. * *a_m^{i_1} * *B_j$" $\in (X_1, ..., X_p)$. For any "$a_1^{i_2} * *.. * *a_m^{i_2} * *B_j$", $i_1 \neq i_2$, two cases can be distinguished. (i) If "$a_1^{i_2} * *.. * *a_m^{i_2} * *B_j$" $\in (Y_1, ..., Y_q)$, then the original column B_j will have part of its values remain as values in the output due to "$a_1^{i_2} * *.. * *a_m^{i_2} * *B_j$", and have the other part of its values become column names in the output due to "$a_1^{i_1} * *.. * *a_m^{i_1} * *B_j$". This violates observation (1) that one column cannot participate in both input parameters. The second case in Figure 2.6 depicts such an example. As can be seen, b_1 and b_2 become both data values and column names. ii) If "$a_1^{i_2} * *.. * *a_m^{i_2} * *B_j$" $\in (X_1, ..., X_p)$, then the output parameters for the second GPIVOT will need to specify data values for both "$a_1^{i_1} * *.. * *a_m^{i_1} * *B_j$" and "$a_1^{i_2} * *.. * *a_m^{i_2} * *B_j$". In this case, the final output column name will have two values from the same column B_j. This however violates observation (1), i.e., the same column is allowed to appear in the input parameters only once. The third case in Figure 2.6 depicts such a case. That is, the result column names have both b_1 and b_2 that are from the same column B.*

By (i) and (ii), we reach the conclusion that $(Y_1, ..., Y_q)$ *must contain all pivoted output columns of the first GPIVOT.*

Third, when $(Y_1, ..., Y_q)$ *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* $(Y_1, ..., Y_q)$ *contains extra columns besides all the previous pivoted output columns, as the fourth case in Figure 2.6. In this case, the two GPIVOTs cannot be combined either. The reason is that the previous pivoted output column names contain "*$a_1^i * *.. * *a_m^i$*", such as* a_1 *and* a_2 *in the above example, while the extra columns do not have such information, such as* N*. Hence the resulting pivoted output columns cannot have the same name structure, which violates observation (2).*

Thus for any two adjacent GPIVOT operations, Equation (6) is the only possible combination rule. ∎

2.4.3 Split Rules for GPIVOT

The split rules for GPIVOT can easily be derived based on the combination rules. For example, the Equations (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. For example, given $GPIVOT^{[\{a_1,a_2\}\times\{b_1,b_2\}]}_{[A,B]\ on\ [C,D]}(T)$, we may split it as $GPIVOT^{[\{a_1,a_2\}]}_{[A]\ on\ [\text{“}b_1**C\text{”},\text{“}b_1**D\text{”},\text{“}b_2**C\text{”},\text{“}b_2**D\text{”}]}(GPIVOT^{[\{b_1,b_2\}]}_{[B]\ on\ [C,D]}(V))$ or as $GPIVOT^{[\{a_1,a_2\}\times\{b_1,b_2\}]}_{[A,B]\ on\ [C]}(V) \bowtie GPIVOT^{[\{a_1,a_2\}\times\{b_1,b_2\}]}_{[A,B]\ on\ [D]}(V)$. The conditions for the applicability of such split rules are the same for those in Equation (5) and (6). However, it is not clear if such split rules will be beneficial for query optimization or for some other tasks.

Nonetheless, there is one useful split rule for parallel processing of

GPIVOT (similar to the parallel processing of the simple PIVOT in [CGGL04]). That is, we compute the GPIVOT sub-results for each processor and then combine them together to generate the final output. This is very similar to the standard local/global aggregation for parallel aggregate processing.

More specifically, assume a dataset S is partitioned to S_1,...,S_n, where each S_i is be processed by one processor and $S = S_1 \cup ... \cup S_n$. Hence at each processor, we compute $GPIVOT(S_1)$, ..., $GPIVOT(S_n)$. In order to compute $GPIVOT(S)$, the GPIVOT sub-results at each processor is combined iteratively as follows:

$GPIVOT(S_1 \cup S_2) = f(GPIVOT(S_1), GPIVOT(S_2))$,

$GPIVOT(S_1 \cup S_2 \cup S_3) = f(GPIVOT(S_1 \cup S_2), GPIVOT(S_3))$,

...

$GPIVOT(S_1 \cup ... \cup S_n) = f(GPIVOT(S_1 \cup ... \cup S_{n-1}), GPIVOT(S_n))$.

At each step, the combination of the sub-result $GPIVOT(S_i)$ is semantically equivalent to inserting the dataset S_i. Hence we can use the propagation rules for the *insert* case in Section 2.6.2, denoted as f above. We will describe these propagation rules in details in later sections.

2.5 Swapping Rules for GPIVOT and GUNPIVOT

As motivated in Section 2.3, 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 swapping rules for both GPIVOT and GUNPIVOT in order to move them up or move them down the query tree. In particular, we will consider the swapping rules be-

tween GPIVOT/GUNPIVOT and four most common relational operators in data warehousing environments, namely, SELECT, PROJECT, JOIN and GROUPBY (including aggregate functions) [GBLP96, ZCL+00]. Table 2.2 gives the roadmap of this section. The swapping rules between GPIVOT and GUNPIVOT will given in Section 2.5.5. We will put the correctness proofs of these swapping rules in Appendix in order to improve readability. In each of these sections, we will also show the completeness of these rules in the sense that there is no other possible swapping rule.

	SELECT	PROJECT	JOIN	GROUPBY
GPIVOT	Section 2.5.1	Section 2.5.2	Section 2.5.3	Section 2.5.4
GUNPIVOT	Section 2.5.6	Section 2.5.7	Section 2.5.8	Section 2.5.9

Table 2.2: Roadmap of Swapping Rules

2.5.1 Swapping Rules for GPIVOT/SELECT

Both the rules for moving GPIVOT up SELECT or the rules for moving GPIVOT down SELECT will be studied. To show completeness, we will first show *all* possible scenarios. Then we will describe if there is any applicable swapping rule under that scenario. Note that Sections 2.5.1 to 2.5.9 all follow the same structure.

Pullup GPIVOT through SELECT

There are two possible scenarios, namely, if the select predicate is on the pivoted output columns or not.

1) If the select 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 2.7.

2) If the select 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 2.7, in order to push down the condition $\sigma_{Sony**TV**Price>200}$, we first need to find its corresponding tuple in the original table by $\sigma_{Manu=Sony\ \wedge\ Type=TV\ \wedge\ Price>200}(V)$. Note that the results of this selection however only contain the Sony TV price information. In order to get the pivot results, we need to perform a self-join with the original table to find other information about these countries. In combination, the GPIVOT pullup is rewritten as:

$GPIVOT(\pi_{Country}(\sigma_{Manu=Sony\ \wedge\ Type=TV\ \wedge\ Price>200}(V)) \bowtie V)$.

More self-joins are required if more pivoted output columns are involved. Intuitively, each pivoted output column (and its selection) corresponds to one tuple (and its selection) in the original table. Hence a select condition on multiple pivoted output columns corresponds to multiple selections on different tuples. Furthermore, in order to make sure all the selections are satisfied, we need to perform more self-joins. Formally, assume a selection predicate over two pivoted output columns as: $\sigma_{\text{“}a_1^{i_1}**\ldots a_m^{i_1}**B_{l_1}\text{”}\ cp\ \text{“}a_1^{i_2}**\ldots a_m^{i_2}**B_{l_2}\text{”}}(GPIVOT^{[\{(a_1^i,\ldots,a_m^i)\}]}_{[A_1,\ldots,A_m]\ on\ [B_1,\ldots,B_n]}(V))$. Here 'cp' is a comparison operation, such as $=$ or $\leq$. This select predicate can be pushed down based on the rule in Equation (7) (proof in Appendix A). Here K^1 and $B^1_{l_1}$ mean the columns K and B_{l_1} in the left operand, while K^2 and $B^2_{l_2}$ mean the columns K and $B^2_{l_2}$ in the right operand.

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

Note that when $i_1 = i_2$, i.e., the column names have the same prefix, then the first join can be avoided: $GPIVOT(\pi_{K^1}[\sigma_{(A_1\ldots A_m)=(a_1^{i_1}\ldots a_m^{i_1})\ \wedge\ B_{l_1}\ op\ B_{l_2}}(V)] \bowtie V)$. The reason is that for any $k \in K$, $\sigma_{(A_1\ldots A_m)=(a_1^{i_1}\ldots a_m^{i_1})\wedge K=k}(V)$ and $\sigma_{(A_1\ldots A_m)=(a_1^{i_2}\ldots a_m^{i_2})\wedge K=k}(V)$ correspond to the same tuple, when $i_1 = i_2$.

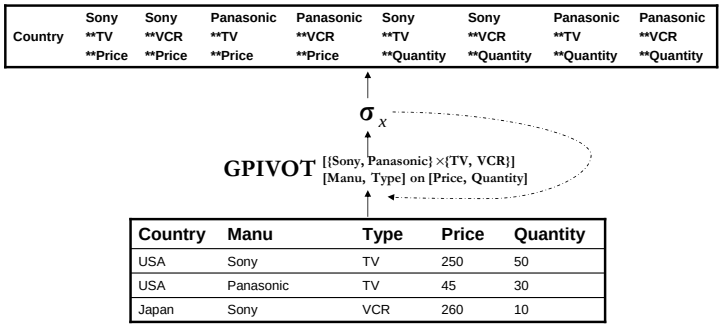


Figure 2.7: Pullup GPIVOT through SELECT

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 as mentioned before. 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 mul-

tiple self-joins since propagating changes through multiple self-join expressions is non-trivial. That is, such propagation would generate multiple join terms [GMS93], which is rather expensive. 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* SELECT and GPIVOT up the query tree and design special update propagation rules. We will describe this technique in Section 2.6.3.

Push GPIVOT Down SELECT

There are three possible scenarios. Assume the input table is $V(K, A_1, ..., A_m, B_1, .., B_n)$ as before and GPIVOT pivots $[A_1, ..., A_m]$ on $[B_1, ..., B_n]$. The select condition can be either on the columns of K, or on the pivoting columns in $\{A_i\}$, or on the value columns in $\{B_j\}$.

1) If the select condition is on K, such as '$\sigma_{country=USA}$' in Figure 2.8, then we can push GPIVOT down the SELECT operator without change.

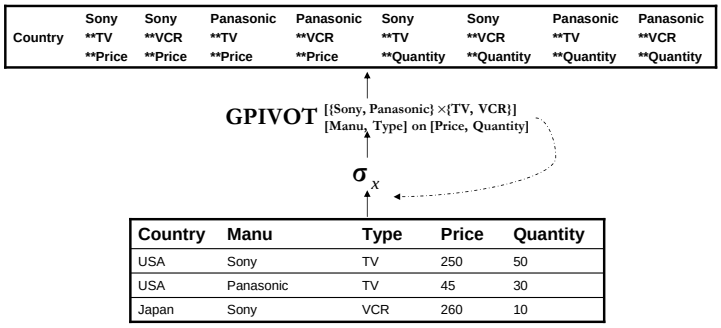


Figure 2.8: Pushdown GPIVOT through SELECT

2) If the select condition is only on the pivoting columns in $\{A_i\}$, such as '$\sigma_{Type=TV}$', then the pushdown results in a PROJECT, which turns all non-

'TV' columns, in this case, 'VCR' related columns, into $\perp$. The reason is that all other non-'TV' information will be filtered by the select. After that, we will need a SELECT to remove the rows that contain only '$\perp$' columns, since by GPIVOT definition in Equation (3), GPIVOT does not output rows with all $\perp$ values. More precisely, it becomes: '$\sigma_{not\ all \perp}(\pi_{country, Sony**TV**Price,}$ $_{Sony**TV**Quantity,\ Panasonic**TV**Price,\ Panasonic**TV**Quantity,\perp,\perp,\perp,\perp})$'.

3) If the select condition is on the value columns in $\{B_j\}$, 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. This is followed by a SELECT, which also removes the rows that contain only $\perp$ columns. More precisely, it becomes '$\sigma_{not\ all \perp}(\pi_{country,\ case(Sony**TV**Price,\ Sony**TV**Quantity),case(Sony**VCR**Price,}$ $_{Sony**VCR**Quantity),\ case(Panasonic**TV**Price,\ Panasonic**TV**Quantity),\ case(Sony**}$ $_{VCR**Price,Sony**VCR**Quantity))}$'. 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 combination of scenario 2), we now describe the pushdown rule in Equation (8).

$$GPIVOT^{[\{(a_1^i,...,a_m^i)\}]}_{[A_1,...,A_m]\ on\ [B_1,...,B_n]}(\sigma_{A_u=x \wedge B_v=y}(V)) =$$
$$\sigma_{not\ all\ \perp}(\pi_{K,\{case(\text{“}a_1^{i_1}**...a_u^{i_1}...**a_m^{i_1}**B_1\text{”},...,\text{“}a_1^{i_1}**...a_u^{i_1}...**a_m^{i_1}**B_n\text{”})\}})($$
$$GPIVOT^{[\{(a_1^i,...,a_m^i)\}]}_{[A_1,...,A_m]\ on\ [B_1,...,B_n]}(V)) \qquad (8)$$

Here the case expression, $case(\text{“}a_1^{i_1} * *...a_u^{i_1}... * *a_m^{i_1} * *B_1\text{”}, ..., \text{“}a_1^{i_1} * *...a_u^{i_1}... * *a_m^{i_1} * *B_n\text{”})$, outputs $(\text{“}a_1^{i_1} * *...a_u^{i_1}... * *a_m^{i_1} * *B_1\text{”}, ..., \text{“}a_1^{i_1} * *...a_u^{i_1}... * *a_m^{i_1} * *B_n\text{”})$ only

when $a_u^{i_1} = x \wedge$ "$a_1^{i_1} * *...a_u^{i_1}...* *a_m^{i_1} * *B_v$" $= y$. Otherwise, it outputs $(\perp, ..., \perp)$. Note that this rule can also easily be extended to handle more complex conditions, such as disjunctive conditions. For example, if the condition on the left side of Equation (8) is $\sigma_{A_u=x \vee B_v=y}(V)$, then the condition in the case expression on the right side becomes $a_u^{i_1} = x \vee$ "$a_1^{i_1} * *...a_u^{i_1}...* *a_m^{i_1} * *B_v$" $= y$.

2.5.2 Swapping Rules for GPIVOT/PROJECT

Pullup GPIVOT through PROJECT

In this work, we consider the negative project, i.e., the removal of columns. There are two possible scenarios. That is, we remove either the non-pivoted output columns or the pivoted output columns.

1) The project operator that drops the non-pivoted output columns can be pushed down unless this project results in the loss of the key, since GPIVOT requires the existence of a key in the input for applicability. For example, the drop of the 'Country' column above the GPIVOT in Figure 2.7 cannot be pushed down since the output no longer contains a key.

2) The project operator that drops the pivoted output columns needs 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 the equation will output TV with $\perp$ while the right part will not.

Push GPIVOT Down PROJECT

Similarly, we also consider the negative project, i.e., the removal of columns. One example is shown in Figure 2.9. As can be seen, if the GPIVOT is

pushed down, then there are two rows regarding 'USA'. 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 $\rightarrow$ Year' holds, then we can pushdown GPIVOT.

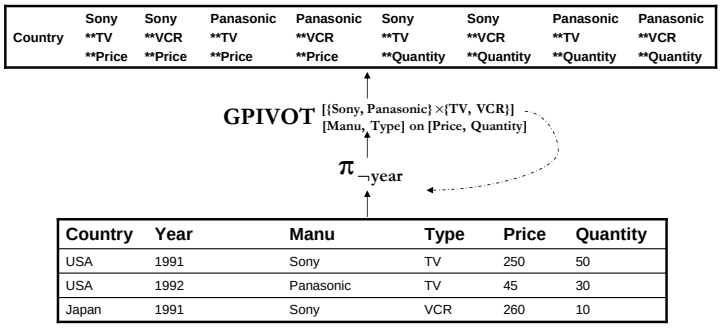


Figure 2.9: Pushdown through PROJECT

2.5.3 Swapping Rules for GPIVOT/JOIN

Pullup GPIVOT through JOIN

Note that the join result should preserve a key in order to pull up the GPIVOT, since GPIVOT requires the existence of a key for applicability. In this case, both operands having a key must hold. This requirement may seem restrictive at first. We note that in the data warehousing scenarios 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.

There are two possible scenarios, namely, if the join predicate is on the pivoted output columns or not. Actually, the rules of pulling up GPIVOT

through the JOIN operator are similar to those for the SELECT operator.

1) If the join condition is not on the pivoted output columns, then we can pull GPIVOT up without change. An example is shown in Figure 2.10. That is, since *AuctionID* is a non-pivoted output column, the GPIVOT operator can be pulled up (the GPIVOT pullup through GROUPBY in the figure will be explained later).

2) 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.

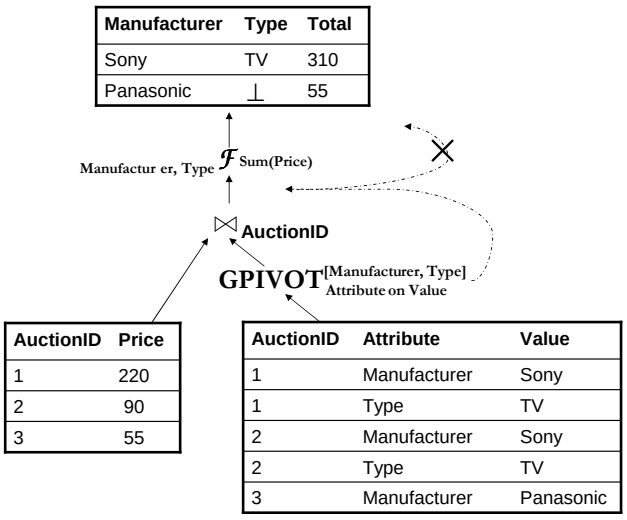


Figure 2.10: Pullup GPIVOT through Join and GROUPBY

As mentioned earlier, we should avoid introducing such multiple self-joins since they would result in inefficient multiple maintenance join terms. Assume a join is $GPIVOT(A) \bowtie_{c_1 \wedge c_2} B$, where c_1 is the join condition involving only non-pivoted output columns of $GPIVOT(A)$ and c_2 is the

join condition involving the pivoted output columns. We can pull up the GPIVOT as $c_2(GPIVOT(A \bowtie c_1 B))$. Then we can pull both c_2 and the GPIVOT together as a pair up the algebra tree (same as that for SELECT). Note that if the join condition c_1 is empty, then the GPIVOT pullup results in a Cartesian product of the underlying tables. If the join condition is $c_1 \vee c_2$, then we cannot split these two conditions. For those two cases, we instead need to choose the insert/delete propagation rules for this GPIVOT.

Push GPIVOT Down JOIN

There are two possible scenarios here. That is, either GPIVOT takes parameters from both join tables or it takes parameters only from one table.

1) Clearly, if GPIVOT takes parameter columns from both of the join tables, then we have to perform GPIVOT after the join.

2) Now assume the expression is GPIVOT(V$\bowtie_{c_1}$T), where c_1 is the join condition. We further assume the same table schema $(K, A_1, ..., A_m, B_1, ..., B_n)$ for V and table schema (X, Y) for T. GPIVOT pivots $\{A_i\}$ on $\{B_j\}$. In this case, the pushdown rules are quite similar to those for push GPIVOT down SELECT.

Three sub-cases are considered. First, if the join condition c_1 is on the column K of table V, e.g., $K = X$, then we can push GPIVOT down the join without change. Second, if the join condition c_1 is on the pivoting column of table V, e.g., $B_2 = X$, then the pushdown result is $\sigma_{not\ all\ \perp}($ $\pi_{K,\{case(\text{“}a_1^i**..a_n^i**B_1\text{”},...,\text{“}a_1^i**..a_n^i**B_m\text{”})\},X,Y}(GPIVOT(V) \bowtie_{(\text{“}a_1^1**..a_n^1**B_2\text{”}=X \vee ... \vee \text{“}a_1^p**..a_n^p**B_2\text{”}=X)} T))$. 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). This can be easily derived from Equation (8). Finally, if the join condition σ_1 is on the value columns 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 [LSS99].

2.5.4 Swapping Rules for GPIVOT/GROUPBY

Pullup GPIVOT through GROUPBY

There are two possible scenarios. That is, either the pivoted output columns are used as group-by columns, or they are used for computing aggregate functions.

1) If the pivoted output columns are group-by columns, then we cannot pull GPIVOT up. Figure 2.10 depicts an example when the GPIVOT operator cannot be pulled up. 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 in the relational model.

2) If the pivoted output columns are used to compute the aggregate, then we can pull up the GPIVOT. The lower PIVOT in Figure 2.1 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 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 2.1 can thus be rewritten as in Figure 2.11.

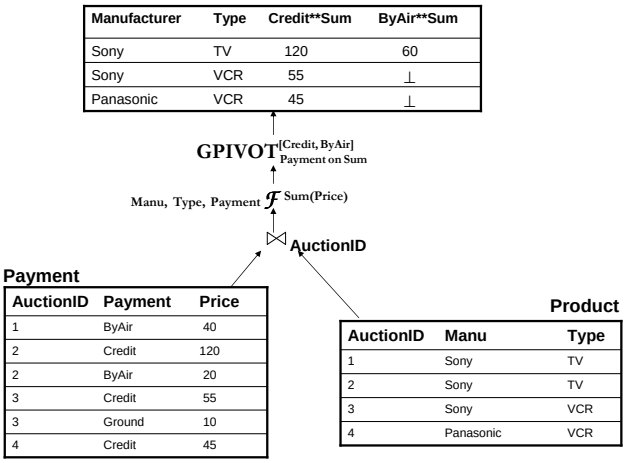


Figure 2.11: Pullup GPIVOT through GROUPBY

Formally, assume the same table V with schema $(K, A_1, ..., A_m, B_1, ..., B_n)$ and GPIVOT pivots $\{A_i\}$ on $\{B_j\}$. The GROUPBY operator $\mathcal{F}$ takes $K' \subseteq K$ as group-by columns and computes aggregate function f over the pivoted output columns $\{\text{“}a_1^i * *... * *a_m^i * *B_j\text{”}\}$. This pull up rule is given in Equation (9). Note that in order to have Equation (9) work, the aggregate function f is restricted to those that do not take $\perp$ into account (as NULL value). Furthermore, it outputs $\perp$ when all the group data are $\perp$. This would require COUNT to output $\perp$ instead of 0 when encountering a

group with all $\perp$ values.

$$_{K'}\mathcal{F}_{f(\{\text{“}a_1^i**\ldots**a_m^i**B_j\text{”}\})}(GPIVOT^{[\{(a_1^i,\ldots,a_m^i)\}]}_{[A_1\ldots A_m]\ on\ [B_1\ldots B_n]}(V)) = GPIVOT^{[\{(a_1^i,\ldots,a_m^i)\}]}_{[A_1,\ldots,A_m]\ on\ [f(B_1),\ldots,f(B_n)]}({}_{K',A_1,\ldots,A_m}\mathcal{F}_{f(B_1),\ldots,f(B_n)}(V)) \quad (9)$$

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. There are two possible scenarios. That is, either GPIVOT pivots some of the $\{B_i\}$ columns or pivots all the $\{B_i\}$ columns.

1) If GPIVOT pivots some of the $\{B_i\}$ columns, then we cannot push GPIVOT down GROUPBY. Assume one B_j column is not chosen for pivot. This B_j now becomes part of the key for pivoting based on GPIVOT definition in Equation (3). Note that since B_j is an aggregate column, we cannot get B_j without first performing GROUPBY. Hence we cannot push GPIVOT down.

2) If GPIVOT pivots all $\{B_i\}$, then Equation (9) can be applied in the reverse manner in order to push GPIVOT down. However, the input to GROUPBY must contain a key in order for the applicability of GPIVOT.

2.5.5 Swapping Rules for GPIVOT/GUNPIVOT

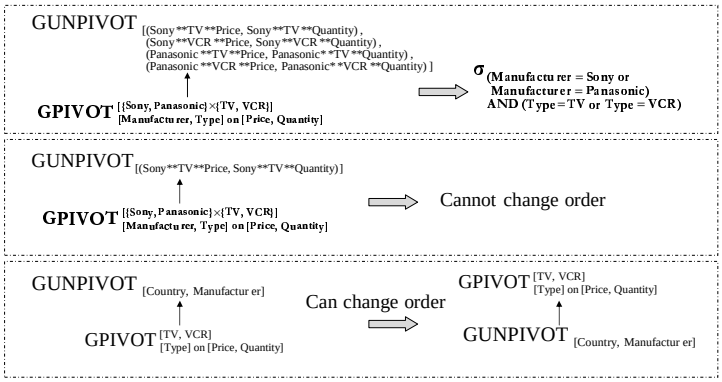
Pullup GPIVOT through GUNPIVOT

There are three possible scenarios. That is, either GUNPIVOT takes *all* the pivoted output columns as its input parameters, or GUNPIVOT takes *part* of the pivoted output columns as its input parameters, or GUNPIVOT takes *none* of the pivoted output columns as its input parameters.

1) GUNPIVOT takes *all* the pivoted output columns as input parameters, then these two operators cancel each other. For example, in the first case of Figure 2.12, these two operators cancel each other. That is, they can be replaced by a simple select condition. Equation (10) formally defines this rule. Here σ_s is a disjunctive predicate on $(A_1...A_m)$, i.e., they equal to any $(a_1^i, ..., a_m^i)$.

$$GUNPIVOT_{[\{\text{“}a_1^i**...a_m^i**B_j\text{”}\}]}(\\ GPIVOT^{[\{a_1^i,...,a_m^i\}]}_{[A_1...A_m] \ on \ [B_1...B_n]}(V)) = (\sigma_s(V)) \qquad (10)$$

2) When GUNPIVOT takes part of the pivoted output columns as its input parameters, their order cannot be changed. Assume two pivoted output columns, “$a_1^{i_1} * *...a_m^{i_1} * *B_i$” and “$a_1^{i_2} * *...a_m^{i_2} * *B_j$”. The first is used for GUNPIVOT as input parameter while the second is not. The final result contains column “$a_1^{i_2} * *...a_m^{i_2} * *B_j$” and columns $A_1, ..., A_m, B_i$. If we were to exchange the order between GPIVOT and GUNPIVOT, then GPIVOT cannot pivot columns $A_1, ..., A_m$ and still retain these columns. In the second case of Figure 2.12, we can see that GUNPIVOT now only



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 2.12: Pullup GPIVOT through GUNPIVOT

partially uses the pivoted output columns. Not only their order cannot be changed, 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.

3) Finally, if the parameters between GPIVOT and GUNPIVOT have no overlap, as the third case in Figure 2.12, then their order can be reversed. Formally, we assume table V has schema $(K, G_1, ..., G_l, A_1, ..., A_m, B_1, ...B_n)$. $(K, G_1, ..., G_l, A_1, ..., A_m)$ together form the key of table V.

$$GUNPIVOT_{[\{G_1,...,G_l\}]}(GPIVOT^{[\{(a_1^i,...,a_m^i)\}]}_{[A_1...A_m]\ on\ [B_1...B_n]}(V)) = GPIVOT^{[\{(a_1^i,...,a_m^i)\}]}_{[A_1...A_m]\ on\ [B_1...B_n]}(GUNPIVOT_{[\{G_1,...,G_l\}]}(V)) \quad (11)$$

Push GPIVOT Down GUNPIVOT

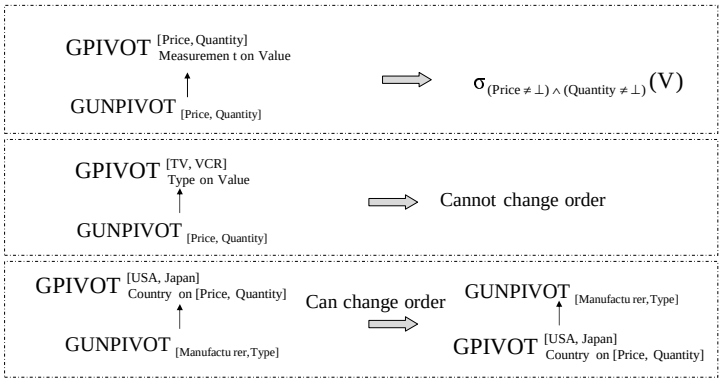
Similarly, there are also three possible scenarios. That is, either GPIVOT takes *all* the unpivoted output columns as its input parameters, or GPIVOT takes *part* of the unpivoted output columns as its input parameters, or GPIVOT takes *none* of the unpivoted output columns as its input parameters.

1) When the GPIVOT takes the GUNPIVOT output columns as parameters. As can be seen in the first case in Figure 2.13, these two operators cancel each other, resulting in a simple selection. Formally, assume table H has schema $(K, \{\text{“}a_1^i * *..a_m^i * *B_j\text{”}\})$, i=1..p and j=1..n. Here K denotes possibly multiple columns and is the key. Equation (12) depicts this rule. Here σ_s is a disjunctive predicate on columns $\{\text{“}a_1^i * *... * *a_m^i * *B_j\text{”}\}$. That is, they do not all equal to $\perp$.

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

2) When GPIVOT partially uses the output columns of GUNPIVOT, as in the second case of Figure 2.13, their order cannot be changed. The reason is that GPIVOT has to use the output columns of GUNPIVOT.

3) Finally, if the parameters between GPIVOT and GUNPIVOT have no overlap, as the third case in Figure 2.13, then their order can be reversed. This essentially is the reverse application of Equation (11).



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 2.13: Push GPIVOT down GUNPIVOT

2.5.6 Swapping Rules for GUNPIVOT/SELECT

Starting from this section, we will present the swapping rules for GUNPIVOT. We now refer to the unpivoted output columns that originated from column values as *value columns* and refer to the unpivoted output columns that originated from column names as *name columns*. For example, in Figure 2.14, 'Type' is a name column while 'Price' is a value column.

Pull GUNPIVOT through SELECT

There are three possible scenarios for pulling GUNPIVOT through SELECT, namely, either the select predicate is on the non-unpivoted output columns, or on value columns, or on name columns.

1) If the select condition is on the non-unpivoted output columns, then we can push it down without any changes such as the condition $\sigma_{Country=USA}$

in Figure 2.14.

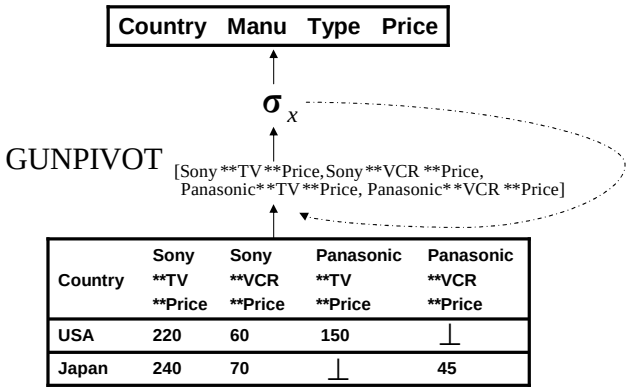


Figure 2.14: Pullup GUNPIVOT through SELECT

2) If the select condition is on a value column, e.g., $\sigma_{Price=150}$, then pushing this select down results in a project that changes the column content. 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 $\perp$.

3) If the select condition is on a 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 $\pi_{\neg(Sony**VCR**Price,\ Panasonic**VCR**Price)}$.

Formally, we assume table H with schema $(K, \{\text{“}a_1^i * *...a_m^i * *B_j\text{”}\})$ (i=1..p, j=1..n), where K can be multiple columns and each “$a_1^i * *...a_m^i * *B_j$” is one column. We further assume a select condition as: $\sigma_{A_u=x \wedge B_v=y}($ $GUNPIVOT_{[\{\text{“}a_1^i**...a_m^i**B_j\text{”}\}]}(H))$. Equation (13) depicts this rule.

$$\sigma_{A_u=x \wedge B_v=y}(GUNPIVOT_{[\{\text{“}a_1^i**...a_m^i**B_j\text{”}\}]}(H)) =$$

$$GUNPIVOT_{[\{\text{“}a_1^i**..**x..a_m^i**B_j\text{”}\}]}(\pi_{K,\{case(\text{“}a_1^i**..**x..a_m^i**B_1\text{”},...,\text{“}a_1^i**..**x..a_m^i**B_n\text{”})\}}(H)) \quad (13)$$

Here $case(\text{“}a_1^i * *.. * *x..a_m^i * *B_1\text{”}, ..., \text{“}a_1^i * *.. * *x..a_m^i * *B_n\text{”})$ is a case expression that if $\text{“}a_1^i * *.. * *x..a_m^i * *B_v = y\text{”}$, then output $(\text{“}a_1^i * *.. * *x..a_m^i * *B_1\text{”}, ..., \text{“}a_1^i * *.. * *x..a_m^i * *B_n\text{”})$, otherwise output (⊥,...,⊥). 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 the individual GUNPIVOT using the above rules.

Push GUNPIVOT down SELECT

There are two possible scenarios for pushing GUNPIVOT down SELECT. That is, the select condition can be either on the non-unpivoted columns or on the unpivoted columns.

1) If the select condition is on the non-unpivoted columns, then we can push the GUNPIVOT operator down without changes, such as $\sigma_{Country=USA}$ in Figure 2.15.

2) If the select condition is on the columns to be unpivoted, e.g., $\sigma_{Sony**TV**Price=220}$, then pushing GUNPIVOT down results in self-joins, i.e., $\pi_{Country}(\sigma_{Sony**TV**Price=220}(H)) \bowtie GUNPIVOT(H)$. Formally, assume a selection predicate over two output columns to be unpivoted as:

$$\begin{aligned} &GUNPIVOT_{[\{\text{“}a_1^i**...a_m^i**B_j\text{”}\}]}(\sigma_{\text{“}a_1^{i_1}**...a_m^{i_1}**B_{l_1}\text{”} \ cp \ \text{“}a_1^{i_2}**...a_m^{i_2}**B_{l_2}\text{”}}(H)) \\ &= \pi_K(\sigma_{\text{“}a_1^{i_1}**...a_m^{i_1}**B_{l_1}\text{”} \ cp \ \text{“}a_1^{i_2}**...a_m^{i_2}**B_{l_2}\text{”}}(H)) \bowtie \\ &\quad GUNPIVOT_{[\{\text{“}a_1^i**...a_m^i**B_j\text{”}\}]}(H) \end{aligned} \quad (14)$$

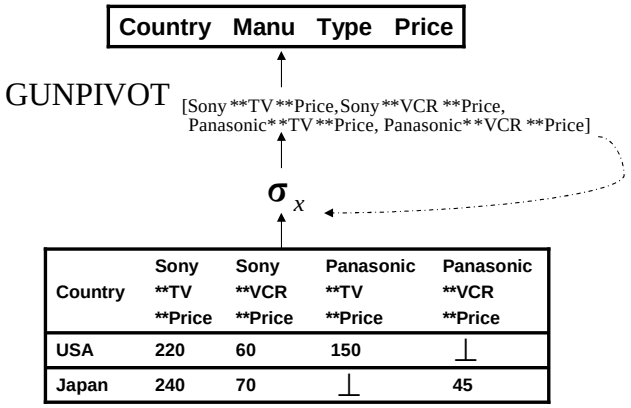


Figure 2.15: Push GUNPIVOT Down SELECT

2.5.7 Swapping Rules for GUNPIVOT/PROJECT

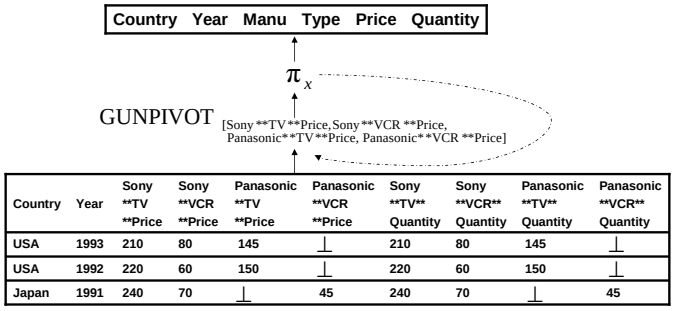
Pull GUNPIVOT through PROJECT

Similarly, we also consider the negative project, i.e., the removal of columns. There are also three scenarios for pulling GUNPIVOT through PROJECT. That is, either we remove either the non-unpivoted output columns, or the value columns, or the name columns.

1) If the project is to remove the non-unpivoted output columns, such as $\pi_{\neg Country}$ in Figure 2.16, we can push the project down without changes.

2) If the project is to remove the value column from the GUNPIVOT output, e.g., $\pi_{\neg Price}$ in Figure 2.16, then pulling up GUNPIVOT results in a project that removes all price related columns, i.e., $\pi_{\neg(Sony**TV**Price, Sony**VCR**Price,\ Panasonic**TV**Price, Panasonic**VCR**Price)}$.

3) If the project is to remove the *name column* from the GUNPIVOT output, e.g., $\pi_{\neg Manu}$ in Figure 2.16, then pulling up GUNPIVOT requires to modify the column names, i.e., removing 'Sony' and 'Panasonic' from the



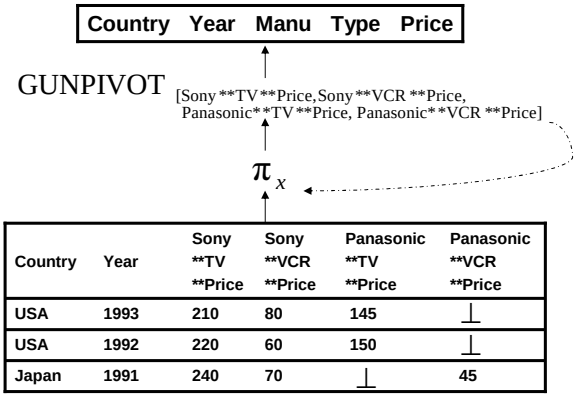
Country	Year	Sony **TV **Price	Sony **VCR **Price	Panasonic **TV **Price	Panasonic **VCR **Price	Sony **TV** Quantity	Sony **VCR** Quantity	Panasonic **TV** Quantity	Panasonic **VCR** Quantity
USA	1993	210	80	145	⊥	210	80	145	⊥
USA	1992	220	60	150	⊥	220	60	150	⊥
Japan	1991	240	70	⊥	45	240	70	⊥	45

Figure 2.16: Pull GUNPIVOT through PROJECT

column names.

Push GUNPIVOT down PROJECT

Note that since the GUNPIVOT operator will not take the removed columns as parameters, it is always possible to push the GUNPIVOT down. For example, we can pull $\pi_{\neg Country}$ up in Figure 2.17. 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.



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

Figure 2.17: Push GUNPIVOT Down PROJECT

2.5.8 Swapping Rules for GUNPIVOT/JOIN

Pull GUNPIVOT through JOIN

The rules for pulling GUNPIVOT above the JOIN are quite similar to those for SELECT in Section 2.5.6. There are also three possible scenarios. That is, the join predicate is either on the non-unpivoted output columns, or on the value columns, or on the name columns.

1) If the join predicate is on the non-unpivoted columns, then we can pull GUNPIVOT above the JOIN without changes.

2) If the join predicate is on a *value column* from the output of GUNPIVOT, then the GUNPIVOT pullup results in a JOIN followed by a PROJECT. Formally, we assume table H with schema $(K, \{\text{“}a_1^i**...a_m^i**B_j\text{”}\})$ (i=1..p, j=1..n), where K can be multiple columns and each “$a_1^i**...a_m^i**B_j$” is one column. Table T has schema (X, Y). 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: $B_l = X$. Equation (15) formally depicts this rule.

$$GUNPIVOT_{[\{\text{“}a_1^i**...a_m^i**B_j\text{”}\}]}(H) \bowtie_{B_l=X} (T) =$$
$$GUNPIVOT_{[\{\text{“}a_1^i**...a_m^i**B_j\text{”}\}]}(\pi_{K,\{case(\text{“}a_1^i**...a_m^i**B_1\text{”},...,}$$
$$\text{“}a_1^i**...a_m^i**B_n\text{”})\},X,Y(\ H \bowtie_{\text{“}a_1^1**...a_m^1**B_l\text{”}=X \vee ... \vee \text{“}a_1^p**...a_m^p**B_l\text{”}=X}\ T)) \quad (15)$$

Here case(“$a_1^i**...a_m^i**B_1$”, ..., “$a_1^i**...a_m^i**B_n$”) is a case expression that if “$a_1^i**...a_m^i**B_l$” $= X$, then output (“$a_1^i**...a_m^i**B_1$”, ..., “$a_1^i**...a_m^i**B_n$”), otherwise output $(\perp, ..., \perp)$.

3) If the join predicate is on a *name column* from the output of GUNPIVOT, e.g., $A_l = Y$ in the above example. Then the pullup of GUNPIVOT requires a join between the column value Y and the column name "$a_1^i * *... * *a_m^i * *B_j$". This requires a higher order feature of the query language [LSS99].

Push GUNPIVOT down JOIN

There are two possible scenarios for pushing GUNPIVOT down JOIN. That is, the join condition can be either on the columns to be unpivoted or not.

1) If the join condition is on the non-unpivoted columns, then we can push GUNPIVOT down the join.

2) If the join condition is on the columns to be unpivoted, then the push-down results in self-joins. Formally, assume the same table schema H and table T with schema (X, Y).

$$GUNPIVOT_{[\{"a_1^i**...a_m^i**B_j"\}]}(H \bowtie_{"a_1^{i_1}**...a_m^{i_1}**B_l"=X} T) =$$
$$\pi_K(H \bowtie_{"a_1^{i_1}**...a_m^{i_1}**B_l"=X} T) \bowtie$$
$$GUNPIVOT_{[\{"a_1^i**...a_m^i**B_j"\}]}(H) \qquad (16)$$

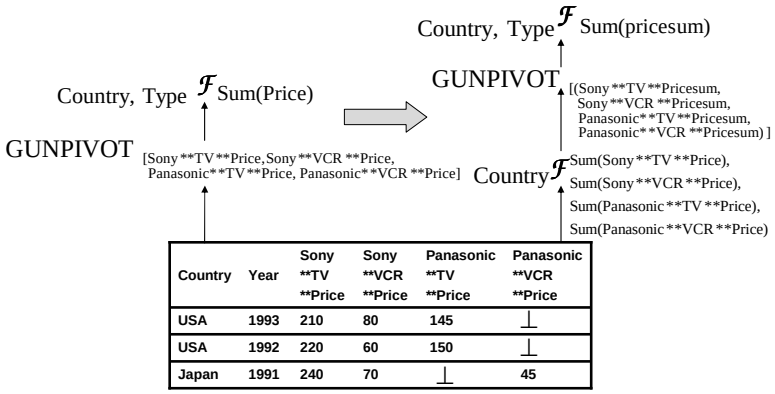
2.5.9 Swapping Rules for GUNPIVOT/GROUPBY

Pull GUNPIVOT through GROUPBY

There are three possible scenarios here. That is, the GROUPBY operator may either aggregate over the value columns from the unpivoted results,

or aggregate over the name columns from the unpivoted results, or do not use any of such columns.

1) When the GROUPBY operator aggregates over the value columns, we may switch their order. Actually, by first unpivoting a table and then performing aggregation, we are able to do *horizontal aggregation* [LSS99]. As can be seen from the example in Figure 2.18, all the prices regarding 'USA' have been summed up even they appear as values in several columns of the same row.



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

Figure 2.18: Pull GUNPIVOT through GROUPBY

In this case, pulling up GUNPIVOT results in a two-level aggregation as shown in Figure 2.18. 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 (17). Here again we assume the table schema H as $(K, \{a_1^i * *...a_m^i * *B_j"\})$ (i=1..p, j=1..n). The output schema of GUNPIVOT(H) is $(K, A_1, ..., A_m, B_1, ..., B_n)$. For simplicity, we also assume here the aggregate function f is sum or count. We can easily extend f to *distributive* or *algebraic* functions (see

Chapter 3) [GBLP96]. Here K' is a subset of columns $(K, A_1, ..., A_m)$ and $K'' = K \cap K'$.

$$
\begin{aligned}
&{}_{K'}\mathcal{F}_{f(B_j)}(GUNPIVOT_{[\{\text{“}a_1^i**...a_m^i**B_j\text{”}\}]}(H)) = \\
&\quad {}_{K'}\mathcal{F}_{f(FB_j)}(GUNPIVOT_{[\{\text{“}a_1^i**...a_m^i**FB_j\text{”}\}]}(\\
&\qquad {}_{K''}\mathcal{F}_{\{\rho_{\text{“}a_1^i**...a_m^i**FB_j\text{”}}(f(\text{“}a_1^i**...a_m^i**B_j\text{”}))\}}(H)) \qquad (17)
\end{aligned}
$$

2) 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.

3) Finally, note that even if the GROUPBY operator does not use any unpivoted output columns of GUNPIVOT, we still cannot remove the GUNPIVOT operator. The reason is that the GUNPIVOT operator may affect the cardinality of the input table, which may subsequently affect the GROUPBY results.

Push GUNPIVOT down GROUPBY

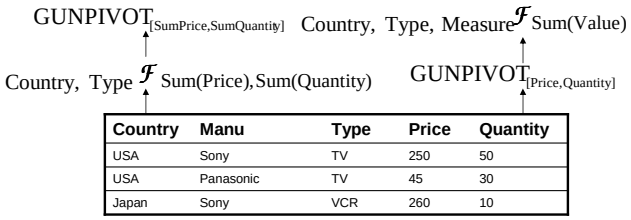
There are two possible scenarios. That is, the GUNPIVOT operator either unpivots the aggregate columns, or unpivots the group-by columns.

1) If GUNPIVOT unpivots the aggregate columns as shown in Figure 2.19, then we can push GUNPIVOT down the group-by. Formally, assume the group-by 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 [4] over column B_i. The GUNPIVOT unpivots $(f(B_1), f(B_2), ..., f(B_n))$ and outputs *name columns*

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

C_N and *values columns* C_V.

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



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

Figure 2.19: Push GUNPIVOT Down GROUPBY

2) If GUNPIVOT unpivots any group-by columns, e.g., [Country,Type], then we cannot push it down. The reason is that after pushing GUNPIVOT down, we cannot perform group-by on *multi-values* from the same column.

2.6 Incremental View Maintenance

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

2.6.1 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 that is a prerequisite for enabling efficient main-

tenance. 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. We note that most existing view maintenance work [GMS93, MQM97] also has this same assumption. If there is no key in the view, i.e., it contains duplicates, then the *count algorithm* [GMS93] would need to maintain the multiplicity of each tuple. This is equivalent to having 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 an SQL update/delete statement to apply the changes efficiently. Most commercial DBMSs [BDD$^+$98, LSPC00] 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 [CD97]. In comparison, pivoting product type based on total sales is often problematic. The reason is that the functional dependency, measurements $\rightarrow$ dimensions, often does not hold. This makes the pivot not applicable.

Note that for the views that do not have the above properties, e.g., having duplicates, we have to apply the insert/delete propagation rules. The resulting maintenance plan may still outperform the full re-computation approach when the source change is small. This makes our solution complete in the sense that it is capable of maintaining any general views with

GPIVOT and GUNPIVOT operators.

2.6.2 Propagation Rules for GPIVOT and GUNPIVOT

The insert/delete propagation rules for GPIVOT and GUNPIVOT are depicted in Figure 2.20. Here '1.' means the first join operand and '2.' means the second join operand. The update propagation rules for GPIVOT are depicted in Figure 2.21. Note that we assume set semantics in this paper since GPIVOT requires a key to exist in its input.

$$\mathrm{GUNPIVOT}(H+\Delta H) = \mathrm{GUNPIVOT}(H) - \mathrm{GUNPIVOT}(\Delta H)$$
$$\mathrm{GUNPIVOT}(H-\nabla H) = \mathrm{GUNPIVOT}(H) - \mathrm{GUNPIVOT}(\nabla H)$$

$$\mathrm{GPIVOT}(V+\Delta V) = \mathrm{GPIVOT}(V)$$
$$(1) \quad - \ \mathrm{GPIVOT}(V) \ltimes_K \mathrm{GPIVOT}(\Delta V)$$
$$(2) \quad + \ \pi_{K, f(1.X, 2.X)}(\mathrm{GPIVOT}(\Delta V) \bowtie_K \mathrm{GPIVOT}(V))$$
$$(3) \quad + \ \mathrm{GPIVOT}(\Delta V) \ \overline{\ltimes}_K \mathrm{GPIVOT}(V)$$
$$\mathrm{GPIVOT}(V-\nabla V) = \mathrm{GPIVOT}(V)$$
$$(4) \quad - \ \mathrm{GPIVOT}(V) \ltimes_K \mathrm{GPIVOT}(\nabla V)$$
$$(5) \quad + \ \sigma_c(\pi_{K, g(1.X, 2.X)}(\mathrm{GPIVOT}(\nabla V) \bowtie_K \mathrm{GPIVOT}(V)))$$

$f(1.X, 2.X)$: 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_c: $\sigma_{\text{any pivoted output column } X \neq '\perp'}$

Figure 2.20: Insert/Delete Propagation Rules for GPIVOT and GUNPIVOT

As can be seen in Figure 2.20, the propagation rules for GUNPIVOT are quite straightforward. The propagation rules for GPIVOT in Figure 2.20 and 2.21 formalize the ideas in Figure 2.2. We now formally prove these rules.

GPIVOT($V + \Delta V$):
T = GPIVOT(ΔV) $\overrightarrow{\bowtie}_K$ GPIVOT(V)
insert: $\pi_{1.*}(\sigma_{2.K \text{ ISNULL}}(T))$
update: $2.X = f(1.X, 2.X)$ where $2.X$ IS NOT NULL

GPIVOT($V - \nabla V$):
T = GPIVOT(∇V) $\overleftarrow{\bowtie}_K$ GPIVOT(V)
update: $2.X = g(1.X, 2.X)$
delete: all $2.X = '\perp'$

Figure 2.21: Update Propagation Rules for GPIVOT

Proofs for Propagation Rules in Figures 2.20 and 2.21: *We first prove the rules in Figure 2.20. The correctness of the GUNPIVOT rules can be shown as follows:*

(1) $GUNPIVOT_{[\{\text{“}a_1^i**...a_m^i**B_j\text{”}\}]}(H + \triangle H)$ *(i=1..p, j=1..n)*

$= [\cup_{i=1}^{p} \pi_{K,a_1^i,...,a_m^i,\text{“}a_1^i**...a_m^i**B_1\text{”},...,\text{“}a_1^i**...a_m^i**B_n\text{”}}($

$\sigma_{any\ \text{“}a_1^i**...a_m^i**B_j\text{”} \neq \perp}(H + \triangle H))$

$= [\cup_{i=1}^{p} \pi_{K,a_1^i,...,a_m^i,\text{“}a_1^i**...a_m^i**B_1\text{”},...,\text{“}a_1^i**...a_m^i**B_n\text{”}}(\sigma_{any\ \text{“}a_1^i**...a_m^i**B_j\text{”} \neq \perp}(H)] +$

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

$= GUNPIVOT_{[\{\text{“}a_1^i**...a_m^i**B_j\text{”}\}]}(H) + GUNPIVOT_{[\{\text{“}a_1^i**...a_m^i**B_j\text{”}\}]}(\triangle H)$

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

(2.1) Next, we show the correctness of the rules for GPIVOT. We first prove the insert case. Given any K value k, we define a set of rows $\{r_i\}$ as:

$r_i = \pi_{K,B_1,...,B_n}(\sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)\ AND\ K=k}(V + \triangle V))$. *If there is no row*

that satisfies the condition for a particular i, we set $r_i = (k, \perp, ..., \perp)$. We also define other two sets of rows $\{s_i\}$ and $\{t_i\}$ as $s_i = \pi_{K,B_1,...,B_n}(\sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)\ AND\ K=k}(V))$ and $t_i = \pi_{K,B_1,...,B_n}(\sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)\ AND\ K=k}(\triangle V))$. Similarly, we also set s_i or t_i as $(k, \perp, ..., \perp)$ if no row satisfies. Based on the above definition, the following function f must hold for any i: $r_i = f(s_i, t_i)$, where $f(s_i, t_i)$ either equals s_i if $t_i = (k, \perp, ..., \perp)$ or equals t_i if $t_i \neq (k, \perp, ..., \perp)$. 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 in both.

Based on the above definition, the output of $GPIVOT(V + \triangle V)$ for key k is $r_1 \bowtie ... \bowtie r_p$, denoted as $\bowtie \{r_i\}$. The output of $GPIVOT(V)$ for key k_1 is $s_1 \bowtie ... \bowtie s_p$, denoted as $\bowtie \{s_i\}$. The output of $GPIVOT(\triangle V)$ for key k is $t_1 \bowtie ... \bowtie t_p$, denote as $\bowtie \{t_i\}$. Then we consider the following two cases:

If $\{s_i\}$ contains only $(k, \perp, ..., \perp)$ tuples, then the original output does not contain such a row with key k based on the GPIVOT definition. In this case, if $\{t_i\}$ contains any non-empty tuples, then $\bowtie \{r_i\} = \bowtie \{f(s_i, t_i)\} = \bowtie \{t_i\}$. This explains to the anti semi-join (line 3) in Figure 2.20.

If both $\{s_i\}$ and $\{t_i\}$ contain any non-empty tuples, then $\bowtie \{r_i\} = \bowtie \{f(s_i, t_i)\}$. This explains to the inner-join term (line 2) for generating a new row in Figure 2.20. Obviously, the original row with key k has to be deleted in this case. This explains the delete term (line 1) in Figure 2.20.

Hence the propagation rules hold under the insert case.

(2.2) We now prove the propagation rules for GPIVOT under the delete case. Given any K value k, we define a set of rows $\{r_i\}$ as $r_i = \pi_{K,B_1,...,B_n}(\ \sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)\ AND\ K=k}(V - \bigtriangledown V))$. If there is no row that satisfies the condition for a particular i, we set $r_i = (k, \perp, ..., \perp)$. We also define two other sets of rows

$\{s_i\}$ and $\{t_i\}$ as $s_i = \pi_{K,B_1,...,B_n}(\sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)\ AND\ \ K=k}(V))$ and $t_i = \pi_{K,B_1,...,B_n}(\sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)\ AND\ K=k}(\bigtriangledown V))$. Similarly, we also set s_i or t_i as $(k, \perp, ..., \perp)$ if no row satisfies them, respectively. Based on the above definition, the following function g must hold for any i: $r_i = g(s_i, t_i)$, where $g(s_i, t_i)$ either equals s_i if $t_i = (k, \perp, ..., \perp)$ or equals $(k, \perp, ..., \perp)$ if $t_i \neq (k, \perp, ..., \perp)$. 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 $V - \bigtriangledown V$.

If both $\{s_i\}$ and $\{t_i\}$ contain any non-empty tuples, then the resulting row becomes $\bowtie \{r_i\} = \bowtie \{g(s_i, t_i)\}$. The original row with $K = k$ has to be deleted in this case. This explains to the delete term (line 4) in Figure 2.20. We insert the new row only when not all $\{r_i\}$ equal $(k, \perp ..., \perp)$. This explains the insert term (line 5) in Figure 2.20.

Hence the propagation rules also hold under the delete case.

(3) It is easy to show that the update propagation rules in Figure 2.21 are equivalent to the insert/delete propagation rules in Figure 2.20. Intuitively, for those rows that we first delete them from the table and then re-insert them again, we can instead perform a single update. ∎

Figure 2.22 describes a simple example to show how to use our propagation rules to maintain a view. 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, namely, (1,Type,TV) and (2,Manufacturer,Panasonic), are inserted into the 'Items' table. Figure 2.23 depicts the

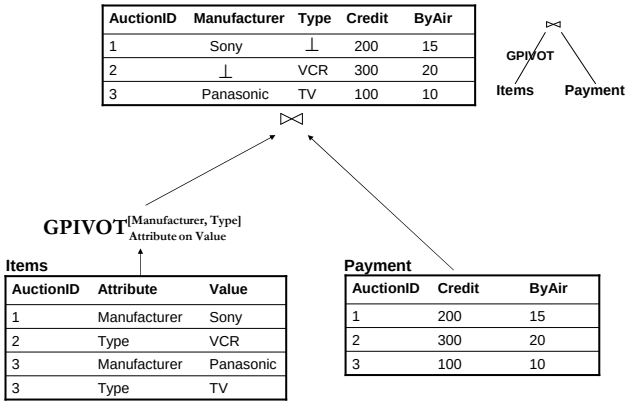


Figure 2.22: A Simple View with GPIVOT

change propagation by naively applying the insert/delete rules for GPIVOT. We can see that the change propagation through the GPIVOT operator generates one insert delta and one delete delta using the rules in Figure 2.20. Each of these two deltas will be further propagated through the JOIN operator. The final maintenance plan is shown at the bottom of the figure, which involves several GPIVOT and JOINs. The results show that we have to delete existing view tuples, namely, (1,Sony,⊥,200,15) and (2,⊥,VCR,300,15), and re-insert them with a few column changes.

Figure 2.24 depicts an alternative maintenance plan achieved by our GPIVOT pullup techniques in order to apply the more efficient update propagation rules. First, the GPIVOT operator is pulled up above the join and becomes the top of the query tree. The *propagation phase* propagates the source deltas through JOIN as $\Delta I \bowtie P$. It then computes the *final delta* as $GPIVOT(\Delta I \bowtie P)$. The *apply phase* uses the update propagation rules for GPIVOT in Figure 2.21 by evaluating a left outer-join between the fi-

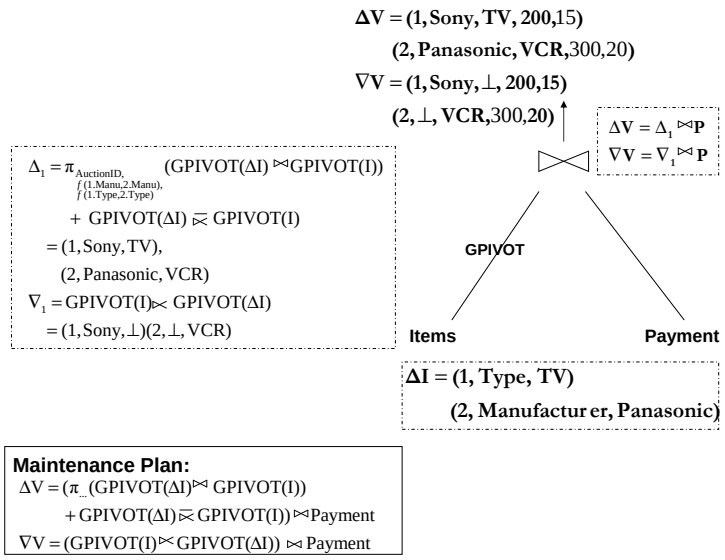


Figure 2.23: Maintenance without GPIVOT Pullup

nal 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 2.23.

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 GPIVOT.

2.6.3 Update Propagation Rules for Multiple Operators

Note that the propagation rules in Section 2.6.2 can be used to maintain any general views with GPIVOT and GUNPIVOT operators. However, the update propagation rules for GPIVOT in Figure 2.21 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 either be expensive to

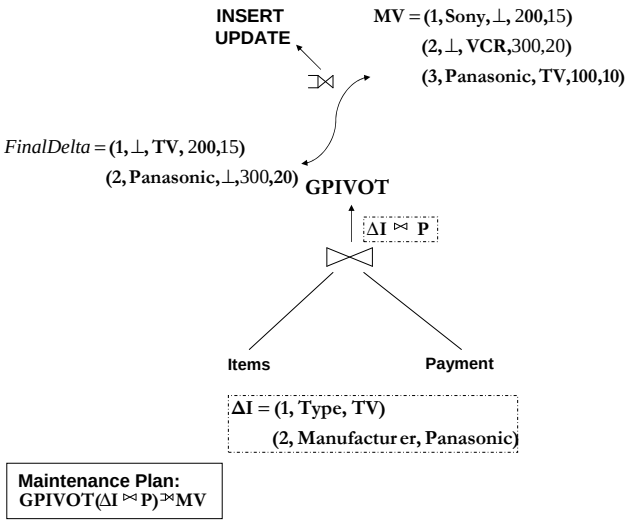


Figure 2.24: Maintenance with GPIVOT Pullup

move GPIVOT to top of the algebra tree or too expensive to compute the delta changes to the input. In the sections below, we propose to solve this problem by developing alternative update propagation rules.

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 [Qua96]. This is inefficient since we may need to re-compute some groups.

We propose to combine the GPIVOT update propagation rules and GROUPBY update propagation rules as depicted in Figure 2.25. Here we assume the output of aggregation, $\mathcal{F}(\mathcal{V})$, 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 $count(*)$. The GPIVOT operator is assumed to pivot $B_1, ..., B_n$ by $A_1, ..., A_m$. For simplicity, we only consider SUM and COUNT in this paper. It is not hard to extend to support AVG or other algebraic functions [GBLP96] using the techniques that will be described in Chapter 3 in this dissertation.

GPIVOT($\mathcal{F}(V + \Delta V)$):

T = GPIVOT($\mathcal{F}(\Delta V)$) $\rtimes_K$ GPIVOT($\mathcal{F}(V)$)

(1) insert: $\pi_{1.*}(\sigma_{2.K \text{ ISNULL}}(T))$;

(2) update: 2."a_1^i**...a_m^i**B_j" = 1."a_1^i**...a_m^i**B_j"+2."a_1^i**...a_m^i**B_j"

where 2.K IS NOT NULL

GPIVOT($\mathcal{F}(V - \nabla V)$):

T = GPIVOT($\mathcal{F}(\nabla V)$) $\bowtie_K$ GPIVOT($\mathcal{F}(V)$)

(3) update: case when 2."a_1^i**...a_m^i**cnt"−1."a_1^i**...a_m^i**cnt" $\neq 0$

then 2."a_1^i**...a_m^i**B_j" = 2."a_1^i**...a_m^i**B_j"−1."a_1^i**...a_m^i**B_j"

else 2."a_1^i**...a_m^i**B_j" = $\perp$ end;

(4) delete: all 2."a_1^i**...a_m^i**B_j" = $\perp$

Figure 2.25: Update Propagation Rules for GPIVOT over GROUPBY

As can be seen from Figure 2.25, under the insert case, we simply update the pivoted aggregate columns in the view by adding the corresponding columns in the delta (line 2 in Figure 2.25). If the delta row does not exist in the view before, then we insert it into the view (line 1). Under the delete case, we update the pivoted aggregate columns in the view by subtracting the corresponding columns in the delta. Note that when the count(*) column, namely, "$a_1^i * *...a_m^i * *cnt$" $= 0$, then all the output columns with the same prefix "$a_1^i * *...a_m^i$" become empty. That is, "$a_1^i * *...a_m^i * *B_j$" $= \perp$ for any j. The reason is that no more items exist

for this subgroup (line 3). When all the output columns become $\perp$, i.e., all subgroups are deleted, this row should be deleted from the view by the GPIVOT definition (line 4). We now formally prove these rules.

Proofs for Combined Update Propagation Rules in Figure 2.25: *We assume the output of the GROUPBY, namely, $\mathcal{F}(\mathcal{V})$, 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 $count(*)$. The GPIVOT operator is assumed to pivot $B_1, ..., B_n$ by $A_1, ..., A_m$. We first prove the insert case.*

(1) Given any K value k, we define a set of rows $\{r_i\}$ as $r_i = \pi_{K,B_1,...,B_n}\mathcal{F}(\sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)\ AND\ K=k}(V + \triangle V))$. If there is no row that satisfies the condition for a particular i, we set $r_i = (k, \perp, ..., \perp)$. We also define two other sets of rows $\{s_i\}$ and $\{t_i\}$ as $s_i =_{K,B_1,...,B_n} \mathcal{F}(\sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)\ AND\ K=k}(V))$ and $t_i =_{K,B_1,...,B_n} \mathcal{F}(\sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)\ AND\ K=k}(\triangle V))$, respectively. Similarly, we also set s_i or t_i respectively as $(k, \perp, ..., \perp)$ if no row satisfies.

Based on the above definition, the output of $GPIVOT(\mathcal{F}(V + \triangle V))$ for key k is $\bowtie \{r_i\}$. The output of $GPIVOT(\mathcal{F}(V))$ for key k is $\bowtie \{s_i\}$. The output of $GPIVOT(\mathcal{F}(\triangle V))$ for key k is $\bowtie \{t_i\}$.

By applying the propagation rules for GROUPBY in [Qua96], we have: if $s_i = (k, \perp, ..., \perp)$, then $r_i = t_i$; if $s_i \neq (k_1, \perp, ..., \perp)$, then $r_i = (k, s_i.B_1 + t_i.B_1, ..., s_i.B_n + t_i.B_n)$, assuming each aggregate function B_j is either a sum or count column.

Then we consider the following two cases:

If $\{s_i\}$ contains only $(k, \perp, ... \perp)$ tuples, then the original output does not

contain such a row with key k. In this case, if $\{t_i\}$ contains any non-empty tuples, then $\bowtie \{r_i\} = \bowtie \{t_i\}$. This explains the insert term (line 1) in Figure 2.25.

If both $\{s_i\}$ and $\{t_i\}$ contain any non-empty tuples, then $\bowtie \{r_i\} = \bowtie \{(k, s_i.B_1 + t_i.B_1, ..., s_i.B_n + t_i.B_n)\}$. This explains the update term (line 2) for generating the new row in Figure 2.25.

Hence the propagation rules hold under the insert case.

(2) Next, we prove the delete case. Given any K value k, we define a set of rows $\{r_i\}$ as $r_i = \pi_{K,B_1,...,B_n}\mathcal{F}(\sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)\ AND\ K=k}(V - \bigtriangledown V))$. If there is no row that satisfies the condition for a particular i, we set $r_i = (k, \perp, ..., \perp)$. We also define other two sets of rows $\{s_i\}$ and $\{t_i\}$ as $s_i = \pi_{K,B_1,...,B_n}\mathcal{F}(\sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)\ AND\ K=k}(V))$ and $t_i = \pi_{K,B_1,...,B_n}\mathcal{F}(\sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)\ AND\ K=k}(\bigtriangledown V))$. Similarly, we also set s_i or t_i as $(k, \perp, ..., \perp)$ if no row satisfies.

Based on the above defintion, the output of $GPIVOT(\mathcal{F}(V - \bigtriangledown V))$ for key k is $\bowtie \{r_i\}$. The output of $GPIVOT(\mathcal{F}(V))$ for key k is $\bowtie \{s_i\}$. The output of $GPIVOT(\mathcal{F}(\bigtriangledown V))$ for key k is $\bowtie \{t_i\}$.

By applying the propagation rules for GROUPBY in [Qua96], we have $r_i = (k, s_i.B_1 - t_i.B_1, ..., s_i.B_n - t_i.B_n)$, assuming each aggregate function B_j is either sum or count. Furthermore, if the count(*) column, namely, $s_i.cnt - t_i.cnt = 0$, then $r_i = (k, \perp, ..., \perp)$.

Thus, $\bowtie \{r_i\} = \bowtie \{(k, s_i.B_1 - t_i.B_1, ..., s_i.B_n - t_i.B_n)\}$. If the count(*) column $r_i.B_j = s_i.B_j - t_i.B_j = 0$, then we need to set $r_i = (k, \perp, ..., \perp)$. This explains the update term (line 3) in Figure 2.25. If the resulting $\{r_i\}$ contains only $(k, \perp ..., \perp)$ tuples, then we should delete this row from the pivot output based on the GPIVOT definition. This explains the delete term (line 4) in Figure 2.25.

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

We now use the motivating example (Figure 2.1) in Section 2.2.2 to describe how to use these update propagation rules to efficiently maintain this view. We first pull up 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 a COUNT(*) column in the view definition in order to make the view incrementally maintainable (Figure 2.26).

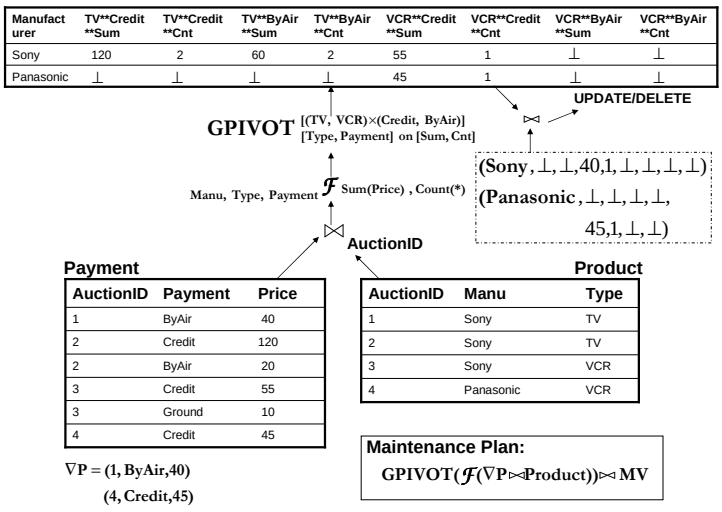


Figure 2.26: Maintenance of View in Figure 2.1

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 2.25 (apply phase) in order to maintain the view. Note that in Figure 2.26 since the resulting VCR**Credit**Cnt for Panasonic equals 0, both VCR**Credit**Cnt and VCR**Credit**Sum will be set to $\perp$. 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 2.21 require significant re-computation of the group-by operator.

Update Propagation Rules for SELECT over GPIVOT

As mentioned before, the other problem with the update propagation rules in Figure 2.21 is that it might be expensive to move GPIVOT to the top of the algebra tree. For example, the rewriting rules show that the pullup of GPIVOT through SELECT may result in multiple self-joins. The propagation through multiple self-joins generates multiple join terms [GMS93]. This can be quite inefficient. In this section, we propose alternative update propagation rules for SELECT (σ_c) on top of a GPIVOT depicted in Figure 2.27.

We first explain the delete case using an example. Figure 2.28 depicts a simple view query with a SELECT above the GPIVOT. To maintain the deletes on the Items table as shown in Figure 2.29, the pullup of GPIVOT above this selection generates multiple self-joins. Alternatively, 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 2.27. It first performs a join between

$\sigma_c(\text{GPIVOT}(V + \Delta V))$:

$T = \text{GPIVOT}(\pi_K(\sigma_{c'}(\Delta V)) \ltimes_K V + \Delta V) \ltimes_K \sigma_c(\text{GPIVOT}(V))$

insert: $\sigma_c(\pi_{1.*}(\sigma_{2.K \text{ ISNULL}} T))$;

update: $2.X = f(1.X, 2.X)$ where $2.K$ IS NOT NULL

$\sigma_c(\text{GPIVOT}(V - \nabla V))$:

$T = \text{GPIVOT}(\Delta V) \ltimes_K \text{GPIVOT}(V)$

update: $2.X = g(1.X, 2.X)$;

delete: $\sigma_{\neg c}(\pi_{2.*}(T))$

Figure 2.27: Update Propagation Rules for SELECT over GPIVOT

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 entries empty as in Figure 2.21. 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.

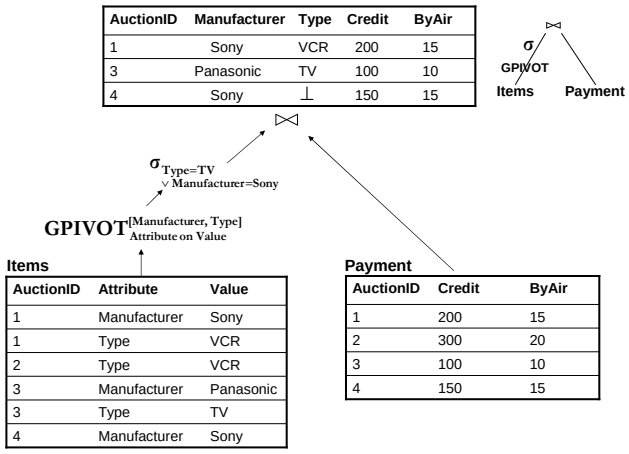


Figure 2.28: Views with SELECT over GPIVOT

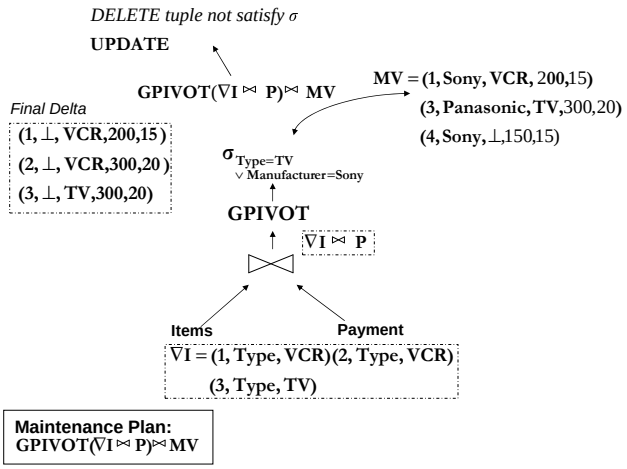


Figure 2.29: Maintenance with SELECT over GPIVOT

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. Auction 3 in Figure 2.28 is an example of this. 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.

In comparison, the source inserts may cause an originally unsatisfied tuple to now satisfy the 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. For instance, 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 2.27 is shown below.

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

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

Note that here we push σ'_c down the join 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 propagation rules below.

Proofs for Combined Update Propagation Rules in Figure 2.27: *We first prove the delete case.*

(1) Given any K value k, we define a set of rows $\{r_i\}$ *as* $r_i = \pi_{K,B_1,...,B_n}(\sigma_{(A_1,...,A_m)=(a^i_1,...,a^i_m)\ AND\ K=k}(V - \bigtriangledown V))$. *If there is no row that satisfies the condition for a particular* i, *we set* $r_i = (k, \perp, ..., \perp)$. *We also define two other sets of rows* $\{s_i\}$ *and* $\{t_i\}$ *as* $s_i = \pi_{K,B_1,...,B_n}(\sigma_{(A_1,...,A_m)=(a^i_1,...,a^i_m)\ AND\ K=k}(V))$ *and* $t_i = \pi_{K,B_1,...,B_n}(\sigma_{(A_1,...,A_m)=(a^i_1,...,a^i_m)\ AND\ K=k}(\bigtriangledown V))$. *Similarly, we also set* s_i *or* t_i *as* $(k_1, \perp, ..., \perp)$ *if no row satisfies their respective predicates.*

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

$GPIVOT(\bigtriangledown V)$ for key k is $\bowtie \{t_i\}$. Similar to the proof for the rules in Figure 2.20, the following function g must hold: $\bowtie \{r_i\} = \bowtie \{g(s_i, t_i)\}$.

Then we consider the following two cases:

i) If the original row $\bowtie \{s_i\}$ satisfies the condition σ_c , then $\sigma_c(\bowtie \{s_i\}) = \bowtie \{s_i\}$, since $\bowtie \{s_i\}$ results in only one row. Or in other words, the original output contains the row $\bowtie \{s_i\}$. Hence, we can compute the new row by $\sigma_c(\bowtie \{r_i\}) = \sigma_c(\bowtie \{g(s_i, t_i)\})$. If the resulting row $\bowtie \{r_i\}$ statisfies the condition σ_c, then we perform updates using function g (defined in the proof (2.2) for the rules in Figure 2.20). This explains the update term in Figure 2.27. If the resulting row $\bowtie \{r_i\}$ no longer satisfies the condition σ_c, then we delete this row from the result. This explains the delete term in Figure 2.27.

ii) If the original row $\bowtie \{p_i\}$ does not satisfy 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, $\sigma_c(\bowtie \{p_i\}) \bowtie (\bowtie \{q_i\})$ must evaluate to an empty result and will not make any changes to the view.

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

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

Based on the above definition, the output of $GPIVOT(V + \triangle V)$ for key k

is $\bowtie \{r_i\}$. *The output of* $GPIVOT(V)$ *for key* k *is* $\bowtie \{s_i\}$. *The output of* $GPIVOT(\triangle V)$ *for key* k *is* $\bowtie \{t_i\}$. *Similar to the proofs for the rules in Figure 2.20, the following function* f *must hold:* $\bowtie \{r_i\} = \bowtie \{f(s_i, t_i)\}$.

Then we consider the following two cases:

i) If the original row $\bowtie \{s_i\}$ *satisfies the condition* σ_c *, then* $\sigma_c(\bowtie \{s_i\}) = \bowtie \{s_i\}$, *since* $\bowtie \{s_i\}$ *results in only one row. Or in other words, the original output contains row* $\bowtie \{s_i\}$. *Hence we can compute the new row by* $\sigma_c(\bowtie \{r_i\}) = \sigma_c(\bowtie \{f(s_i, t_i)\})$. *Note that in this case the resulting row* $\bowtie \{r_i\}$ *must still satisfy the condition* σ_c. *The reason is that if the resulting row* $\bowtie \{r_i\}$ *did not satisfy the condition, then turning some of its columns to null, i.e., the original row* $\bowtie \{s_i\}$, *still should not satisfy the condition when* σ_c *is null-intolerant. We thus only need to perform updates using function* f. *This explains the update term in Figure 2.27.*

ii) If the original row $\bowtie \{s_i\}$ *does not satisfy the condition* σ_c *, the resulting* $\bowtie \{r_i\}$ *may now satisfy the condition. Hence, in this case, we have to re-compute* $\bowtie \{r_i\}$, *since the original output does not contain the row* $\bowtie \{s_i\}$. *The re-computation can be evaluated as* $GPIVOT(\pi_K(\triangle V) \bowtie (V + \triangle V))$, *i.e., we re-compute those rows that are affected by the deltas,* $\pi_K(\triangle V)$. *We insert these new rows that now satisfy the condition into the view. That is,* $\sigma_c(GPIVOT(\pi_K(\triangle V) \bowtie (V + \triangle V)))$.

Furthermore, we observe that only when those rows in $\triangle V$ *that related to the columns referenced in* σ_c *may generate new rows to the view. Hence, we can add* $\sigma_{c'}$ *to reduce the join size as in Figure 2.27.* $\sigma_{c'}$ *is of the form* $(A_1, ..., A_m) = (a_1^{i_1}, ..., a_m^{i_1}) \vee ... \vee (A_1, ..., A_m) = (a_1^{i_l}, ..., a_m^{i_l})$, *where* σ_c *refers to the columns "*$a_1^{i_j} * *... * * a_m^{i_j}$*" . This explains the insert term in Figure 2.27.*

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

2.7 Experimental Evaluation

In this section, we will experimentally evaluate the effectiveness of our maintenance framework by measuring the performance of the maintenance plans. In particular, we will show that the maintenance using update propagation rules by first transforming the query into a particular "top-heavy" shape is more efficient than the direct application of the insert/delete propagation rules.

2.7.1 Setup

Implementation

In this work, we manually generate the maintenance queries and measure the execution cost of these maintenance queries on top of a commercial DBMS (Oracle 10g [ora]). Compared to an actual implementation inside the database engine, our current maintenance measurement will not include the cost of generating a maintenance plan. However, we argue that such cost is generally much smaller than the actual maintenance. The reasons are that, first, although our maintenance plan generation involves some query transformation, such query transformation is much cheaper than that for query optimization. Since the goal of our query transformation is simply to move GPIVOT up, a single traversal of the query suffices. In comparison, the general query optimization needs to consider a much larger search space for query transformation. Second, in data warehousing scenarios, the size of data often easily exceeds giga-bytes or even tera-bytes. In this case, the cost of generating a maintenance plan becomes almost negligible com-

pared to the actual execution. Hence, our measurement is a good indication for the maintenance performance.

Recall that our proposed maintenance technique has two phases, namely, the propagation phase and the apply phase, as described in Section 2.3. During the propagation phase, we may need to compute GPIVOT. This currently has not yet been supported by any commercial database. To solve this, similar to [CGGL04], we implement GPIVOT using the following SQL GROUPBY subquery (in a concise form):

SELECT K,

$max(case((A_1, ..., A_m) = (a_1^1, ..., a_m^1), B_1, \perp))$ as "$a_1^1 ** ... ** a_m^1 ** B_1$",

$max(case((A_1, ..., A_m) = (a_1^1, ..., a_m^1), B_2, \perp))$ as "$a_1^1 ** ... ** a_m^1 ** B_2$",

...

$max(case((A_1, ..., A_m) = (a_1^1, ..., a_m^1), B_n, \perp))$ as "$a_1^1 ** ... ** a_m^1 ** B_n$",

...

$max(case((A_1, ..., A_m) = (a_1^i, ..., a_m^i), B_j, \perp))$ as "$a_1^i ** ... ** a_m^i ** B_j$",

...

$max(case((A_1, ..., A_m) = (a_1^p, ..., a_m^p), B_1, \perp))$ as "$a_1^p ** ... ** a_m^p ** B_1$",

$max(case((A_1, ..., A_m) = (a_1^p, ..., a_m^p), B_2, \perp))$ as "$a_1^p ** ... ** a_m^p ** B_2$",

...

$max(case((A_1, ..., A_m) = (a_1^p, ..., a_m^p), B_n, \perp))$ as "$a_1^p ** ... ** a_m^p ** B_n$"

FROM V

WHERE $(A_1, ..., A_m)$ in $\{(a_1^i, ..., a_m^i)\}$

GROUP BY K

Here $case((A_1, ..., A_m) = (a_1^i, ..., a_m^i), B_j, \perp)$ is a case expression that for a given row r in V, if its columns $(A_1, ..., A_m) = (a_1^i, ..., a_m^i)$, then we output its B_j column, otherwise, we output $\perp$. The *max* function is used here simply to conform to the SQL standard, since a non-group-by output column must be an aggregate column. $(A_1, ..., A_m)\ in\ \{(a_1^i, ..., a_m^i)\}$ is an abbreviation of a disjunctive predicate, i.e., $(A_1 = a_1^1 \wedge ... \wedge A_m = a_m^1) \vee ... \vee (A_1 = a_1^p \wedge ... \wedge A_m = a_m^p)$.

The apply phase will first evaluate 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 such INSERT, DELETE or UPDATE in one statement in order to avoid materializing any temporary results. For this, we will use the MERGE operation [ora] 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.

The syntax of MERGE for integrating the *final_delta* (the sub-query of propagate phase in Section 2.3) into the view in the case of insertion is as follows. Here we assume the view has key K and two pivoted output columns, namely, c_1 and c_2.

MERGE INTO view
USING final_delta
ON view.K = final_delta.K
WHEN MATCHED

UPDATE SET view.c_1 = case((final_delta.c_1 $=\perp$),view.c_1, final_delta.c_1),
view.c_2 = case((final_delta.c_2 $=\perp$),view.c_2, final_delta.c_2)
WHEN NOT MATCHED
INSERT VALUES (final_delta.K, final_delta.c_1, final_delta.c_2)

As can be seen, it first performs a join between the final_delta and the view on the key K. If there is such a matching tuple, then we update the view tuple using the case expression. In particular, this case expression outputs view.c_1 (or c_2) if final_delta.c_1 (or c_2) is $\perp$, otherwise it outputs final_delta.c_1 (or c_2). If there is no such match, then we know that it is a new row and insert it into the view.

The syntax of MERGE for integrating *final_delta* (the sub-query of propagate phase in Section 2.3) into the view in the case of deletion is as follows:

MERGE INTO view
USING final_delta
ON view.K = final_delta.K
WHEN MATCHED
UPDATE SET view.c_1 = case((final_delta.c_1 $=\perp$),view.c_1,$\perp$),
view.c_2 = case((final_delta.c_2 $=\perp$),view.c_2,$\perp$)
DELETE WHEN view.c_1 = $\perp$ AND view.c_2 = $\perp$

Compared to the insertion case, the difference is that when there is a match between final_delta and view, the case expression outputs $\perp$ if final_delta.c_1 (or c_2) does not equal $\perp$, otherwise it outputs view.c_1 (or c_2).

After generating the new view tuple, we shall delete it if all its pivoted output columns have become $\perp$.

Testbed

We run our experiments on a TPC-H [TPC95] database with scale factor 1.0, i.e., the total size of tables is around 1 gigabytes. The schema of the three TPC-H tables used in the experiments, namely, Lineitem, Orders and Customer, are described below.

Lineitem(l_orderkey, l_partkey, l_supplykey, l_linenumber, l_quantity, l_extendedprice, l_discount, l_tax, l_returnflag, l_linestatus, l_commitdate, l_receiptdate, l_shipinstruct, l_shopmode, l_comment). Total 6,000,000 rows.

Order(o_orderkey, o_custkey, o_orderstatus, o_totalprice, o_orderdate, o_orderpriority, o_clerk, o_shippriority, o_comment). Total 1,500,000 rows.

Customer(c_custkey, c_name, c_address, c_nationkey, c_phone, c_acctbal, c_mktsegment, c_comment). Total 150,000 rows.

Our experiments vary the types of views, as well as the sizes of the source changes and their impact on the views. We then study how the different maintenance methods perform under these conditions. All the experiments are conducted on a dual-CPU 800MHZ machine with 1G memory, running Linux.

2.7.2 Maintaining Non-aggregate Views

Without SELECT on TOP of GPIVOT

Figure 2.30 gives the algebra definition of a non-aggregate materialized view. As can be seen, it first pivots the Lineitem table and then joins the results with the Orders and Customer tables. The size of this view is 1,500,000 rows. Here the rename operator ρ is to rename the column names from 1,2,...,7 to itm1,itm2,...,itm7. The corresponding view definition in SQL used in our experiments is depicted in Figure 2.31.

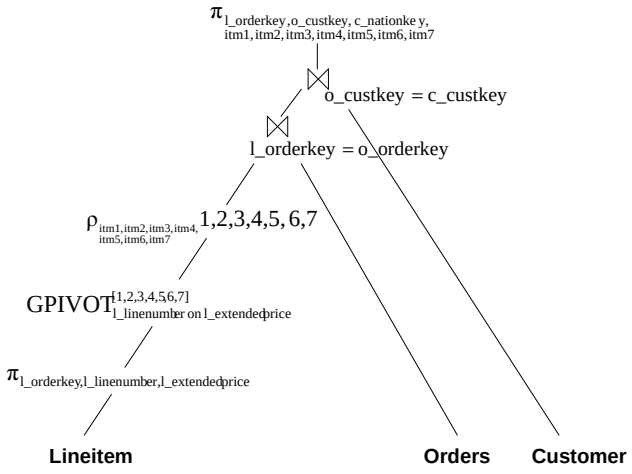


Figure 2.30: Materialized View Definition for View1

Before we describe the experimental results, we provide a brief cost analysis of the maintenance methods we use. In particular, we compare the maintenance using the insert/delete propagation rules to the maintenance using the update propagation rules in Figures 2.20 and 2.21.

```
CREATE  VIEW  view1
SELECT  l_orderkey, o_custkey,c_nationkey,
        max(case((l_linenumber = 1),l_extendedprice,⊥)) as itm1,
        max(case((l_linenumber = 2),l_extendedprice,⊥)) as itm2,
        ....
        max(case((l_linenumber = 7),l_extendedprice,⊥)) as itm7
FROM  Lineitem, Orders, Customer
WHERE  l_orderkey= o_orderkey AND o_custkey = c_custkey
        AND (l_linenumber = 1 OR ... OR l_linenumber = 7)
GROUPBY  l_orderkey, o_custkey,c_nationkey
```

Figure 2.31: Materialized View Definition for View1 in SQL

Cost analysis. *Assume one view is defined as* $V = GPIVOT(T)$ [5]. *Here* T *can be any arbitrary sub-query. The maintenance expressions generated using the insert/delete propagation rules are:*

(1) $GPIVOT(\Delta T) \bowtie GPIVOT(T)$ *for deleting existing view rows;*

(2) $GPIVOT(\Delta T) \rightarrowtail GPIVOT(T)$ *for generating the new view row.*

The maintenance expression generated using the update propagation rules is simply:

(3) $GPIVOT(\Delta T) \rightarrowtail V$.

Clearly, the cost of evaluating expression (3) is always cheaper than that of evaluating expressions (1) and (2). The reason is that, first the evaluation of expression (2) is always more expensive than that of expression (3). The reason is that $V = GPIVOT(T)$. *Expression (3) uses pre-computed result* V *while expression (2) instead re-computes it from scratch. Second, the computation of expression (1) is an extra cost, which the update propagation rules do not incur.*

[5] Here the parameters for GPIVOT are not important for cost analysis and are thus omitted for brevity.

Maintenance under Source Insertions: We now discuss the experimental results and show that they are consistent with our cost analysis. First, we consider the insert case on the 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 2.20. The third method is to first pull up GPIVOT to the top of the algebra tree and then apply the update propagation rules in Figure 2.21.

In particular, we distinguish between two extreme cases. The first case is that the insert of the source data causes only *view updates*. Under this scenario, the cost of expression (1) in the above cost analysis is maximal. Hence, update propagation rules likely will significantly outperform the insert/delete propagation rules. The second case is that the insert of the source data causes only *view inserts*. Under this scenario, the insert/delete rules may perform better since the execution cost of expression (1) is minimal. The goal of this experiment is to justify if the update propagation rules are always preferable choices.

Figure 2.32 depicts the maintenance results for the first case when the source changes result in only view updates. Here y-axis denotes the execution cost of the maintenance expression in seconds. x-axis denotes the percentages of insertion on the Lineitem table, namely, the number of tuples inserted over the number of tuples in the original Lineitem table.

As can be seen, the maintenance using the update propagation rules performs much better than the maintenance using the insert/delete propagation rules. This is consistent with our previous cost analysis, since the

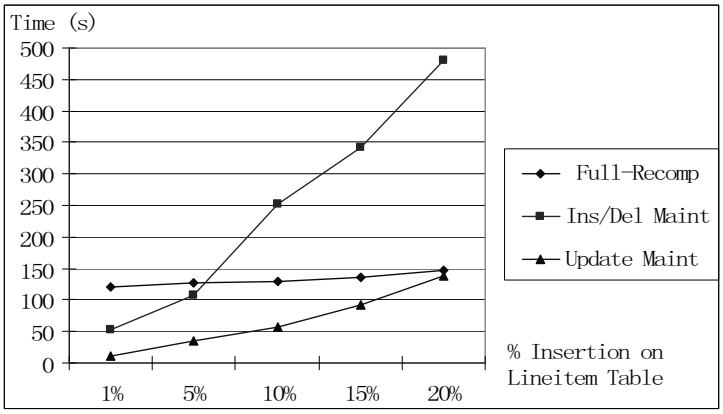


Figure 2.32: Maintenance of View1 under Source Insertion (Resulting in Only View Updates)

insert/delete rules cause many existing view tuples to be deleted and re-inserted again.

Also the maintenance using the update propagation rules outperforms the full re-computation when the size of insertion is small. At 1% insertion, the maintenance using the update propagation rules already performs one order of magnitude better than full re-computation (120s .vs. 13s). The maintenance cost increases faster than the full re-computation cost as the size of insertion increases. The reason is that the cost of integrating the deltas to the view increases. Such cost does not occur for full re-computation. The crossover point is around 20% in this case [6]. This makes our solution fairly attractive in data warehousing scenarios, since most data

[6]This crossover point primarily will be affected by the size of view, as will be described in the Workarea Chapter 3. The maintenance of aggregate (or pivoting) views contains two portions: propagate phase and apply phase. The propagate phase often accesses less data than that for re-computation, while the apply phase is an extra cost compared to re-computation. Hence the larger the size of the view, the lower the crossover point between incremental maintenance and full re-computation. The experimental results in Figure 3.7 describe such behavior.

warehouse periodical load is small compared to the base table size.

Figure 2.33 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 performs much better than in the former case, as no existing view tuples will be deleted and re-inserted again. However, the maintenance using the update rules still outperforms the maintenance using the insert/delete rules by a factor of two. This is also consistent with our previous cost analysis: because the maintenance expression (3) uses the pre-computed result (view), while the maintenance expression (2) has to re-compute the view from scratch.

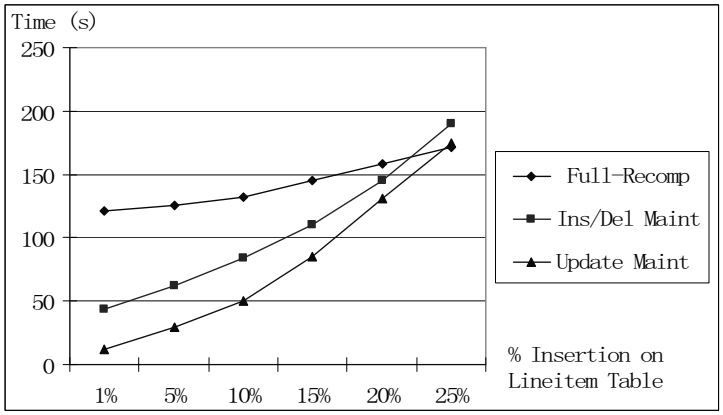


Figure 2.33: Maintenance of View1 under Source Insertions (Resulting in Only View Insertions)

Maintenance under Source Deletions: Next, we consider the maintenance of View1 (Figure 2.30) under the deletion on the Lineitem table. Similarly, three methods can be used to refresh the view. The first method is to perform full re-computation. The second method is to perform incremental maintenance using the insert/delete propagation rules for GPIVOT in Fig-

ure 2.20, while the third method is to first pull up GPIVOT to the top of the algebra tree and then apply the update propagation rules in Figure 2.21.

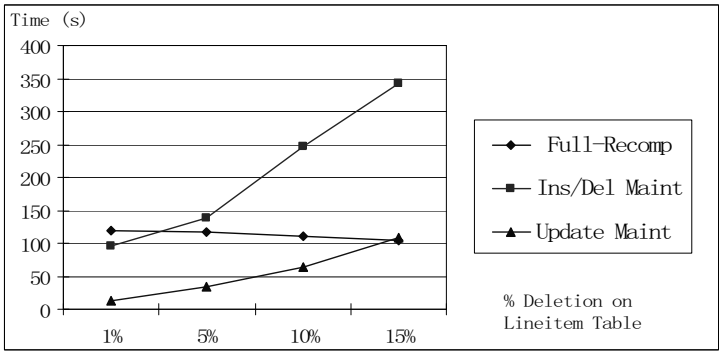


Figure 2.34: Maintenance of View1 under Deletion

Figure 2.34 depicts the maintenance results. Here y-axis denotes the execution cost of the maintenance expression in seconds. x-axis denotes the percentages of deletion on the Lineitem table. As can be seen from the figure, the maintenance method using the update rules performs three to four times better than the method using the insert/delete rules. This is consistent with our cost analysis and the results in the insertion case.

The maintenance cost increases as the size of deletion increases, while at the same time, the full re-computation cost decreases since the size of the base table decreases. The crossover point is now around 15% in this case. Note that such 15% deletion of the historical data warehouse data can be still fairly large in practice. This means that our incremental maintenance solution would be useful under most common data warehouse workloads.

Hence, from both our cost analysis and the experiments in Figures 2.32, 2.33 and 2.34, we conclude that the update propagation rules are always

preferable choices compared to the insert/delete propagation rules.

With SELECT on TOP of GPIVOT

Figure 2.35 gives the algebra definition of another type of view, namely, a 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. The corresponding view definition in SQL used in our experiments is depicted in Figure 2.36.

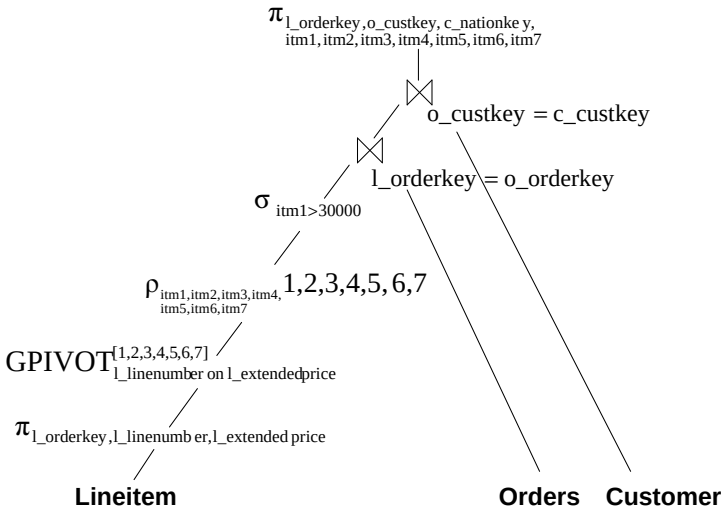


Figure 2.35: Materialized View Definition for View2

Before we describe the experimental results, we will provide a brief cost analysis of the maintenance methods we use. In particular, we will compare the maintenance using update propagation rules in Figure 2.21, which requires us to push down the SELECT resulting in multiple self-joins, to the

```
CREATE VIEW view 2
SELECT l_orderkey, o_custkey, c_nationkey,
       itm1,itm2,itm3,itm4,itm5,itm6,itm7
FROM  (SELECT  l_orderkey,o_custkey, c_nationkey,
          max(case((l _linenumber = 1),l_extended price,⊥)) as itm1,
          max(case((l _linenumber = 2),l_extended price,⊥)) as itm2,
          ...
          max(case((l _linenumber = 7),l_extended price,⊥)) as itm7
        FROM  Lineitem
        WHER E  l_orderkey = o_orderkey AND o_custkey = c_custkey
          AND (l_linenumber = 1 OR ... OR l_linenumber = 7)
        GROUPBY  l_orderkey) as TEMP, Orders, Customer
WHERE  l_orderkey = o_orderkey  AND  o_custkey = c_custkey
        AND itm1 > 30000
```

Figure 2.36: Materialized View Definition for View2 in SQL

maintenance using the combined update propagation rules in Figure 2.27.

Cost analysis. *We first consider the simplest case that the select predicate involves only one pivoted output column. Assume one view is defined as* $V = \sigma_c(GPIVOT(T))$. *Here* T *can be any arbitrary sub-query. After pushing* σ_c *down as in Section 2.5.1, we have* $V = GPIVOT(\sigma_{c'}(T) \bowtie T)$.

In the deletion case, the maintenance expression generated by SELECT push-down and the update propagation rules in Figure 2.21 is:

(1) $GPIVOT(\sigma_{c'}(\bigtriangledown T) \bowtie (T - \bigtriangledown T) + \sigma_{c'}(T) \bowtie \bigtriangledown T) \bowtie V$.

The maintenance expression generated by the combined update propagation rules explained in Figure 2.27 is simply:

(2) $GPIVOT(\bigtriangledown T) \bowtie V$.

Clearly, the cost of evaluating expression (2) is always cheaper than that of evaluating expression (1).

In the insertion case, the maintenance expression generated by SELECT pushdown and the update propagation rules in Figure 2.21 is:

(3) $GPIVOT(\sigma_{c'}(\triangle T) \bowtie (T + \triangle T) + \sigma_{c'}(T) \bowtie \triangle T) \rtimes V$.

The maintenance expression generated by the combined update propagation rules as in Figure 2.27 is:

(4) $GPIVOT(\triangle T + \sigma_{c'}(\triangle T) \bowtie T) \rtimes V$.

The maintenance expression (4) avoids the join term $\sigma_{c'}(T) \bowtie \triangle T$ *compared to expression (3). The semantics* $\sigma_{c'}(T) \bowtie \triangle T$ *is to find the delta rows with the keys that had satisfied the select condition in the original base tables. Actually these keys must be in the original view since* $V = GPIVOT(\sigma_{c'}(T))$*. The join between view* V *will provide us such information.*

When the select predicate involves more pivoted output columns, the maintenance by select pushdown will generate even more self-join terms, while the maintenance using combined update propagation rules remains the same.

Maintenance under Source Deletions: Based on the cost analysis, we now discuss the experimental results. We first consider the delete case on the 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 2.20. 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 2.21. The fourth method is to pull both SELECT and GPIVOT up and apply the combined update propagation rules

in Figure 2.27.

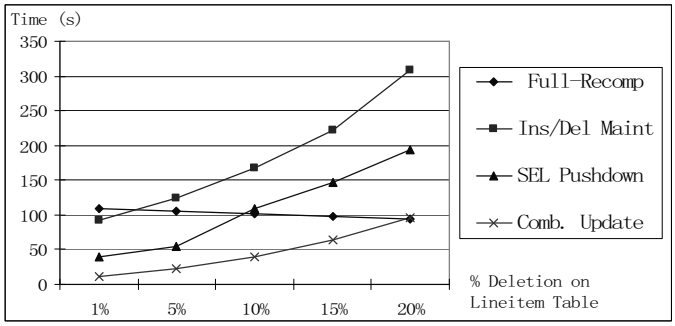


Figure 2.37: Maintenance of View2 under Deletion

Figure 2.37 depicts the maintenance results. Here the y-axis denotes the execution cost of the maintenance expression in seconds. x-axis denotes the percentages of deletions on the Lineitem table. As can be seen from the figure, the maintenance method using our combined update rules (in Figure 2.27) considerably outperforms both the maintenance method using the update propagation rules by SELECT pushdown and using the insert/delete propagation rules.

More precisely, the maintenance plan by the SELECT/GPIVOT combined update propagation rules is:

$(GPIVOT(\bigtriangledown L) \bowtie C \bowtie O) \bowtie MV$ [7].

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

$((GPIVOT(\sigma_{c'}(\bigtriangledown L) \bowtie (L - \bigtriangledown L) + \sigma_{c'}(L) \bowtie \bigtriangledown L) \bowtie O \bowtie C) \bowtie MV$ [8].

Obviously, propagating changes through multiple self-joins is non-trivial, as it generates multiple join terms [GMS93]. Note that when the select con-

[7] 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.

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

dition involves more pivoted output columns, then more self-joins will be generated when pushing down the select operator. Hence, the select push-down method will likely perform even worse in this case. This confirms the result of our cost analysis that the combined update propagation rules generate a cheaper maintenance plan.

Maintenance under Source Insertions: Next, we consider the insert case on the Lineitem table. Figure 2.38 depicts the maintenance results.

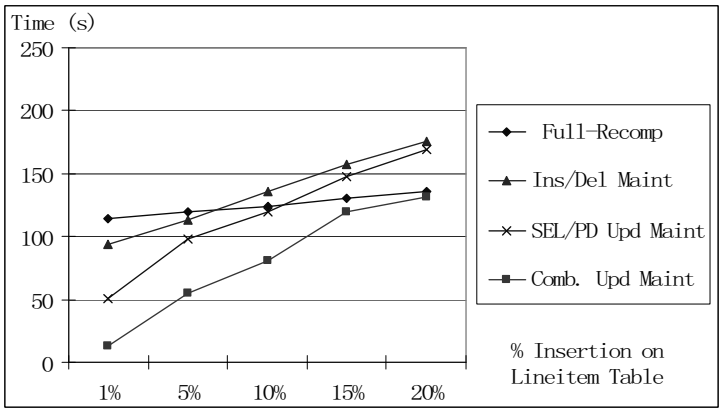


Figure 2.38: Maintenance of View2 under Insertion

We find that our combined update propagation rules again outperform the SELECT pushdown and the insert/delete maintenance methods. Here the maintenance plan generated by combined update rules is:

$((GPIVOT(\sigma_{c'}(\triangle L) \bowtie L + \triangle L) \bowtie O \bowtie C) \overline{\supset\!\bowtie} MV$.

In comparison, the maintenance plan by SELECT pushdown is:

$((GPIVOT(\sigma_{c'}(\triangle L) \bowtie (L + \triangle L) + \sigma_{c'}(L) \bowtie \triangle L) \bowtie O \bowtie C) \overline{\supset\!\bowtie} MV$.

Clearly, the latter plan generates more join terms than the former. It

would generate even more join terms when the select condition is more complex. Thus both our cost analysis and experimental results confirm that our combined update rules are preferable incremental maintenance choices.

2.7.3 Maintaining Aggregate Views

Figure 2.39 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 cross-tab view, namely, with each output tuple containing the summary information from 1992 to 1996 for each customer. The size of this view is 100,000 rows with 12 columns. The corresponding view definition in SQL used in our experiments is depicted in Figure 2.40.

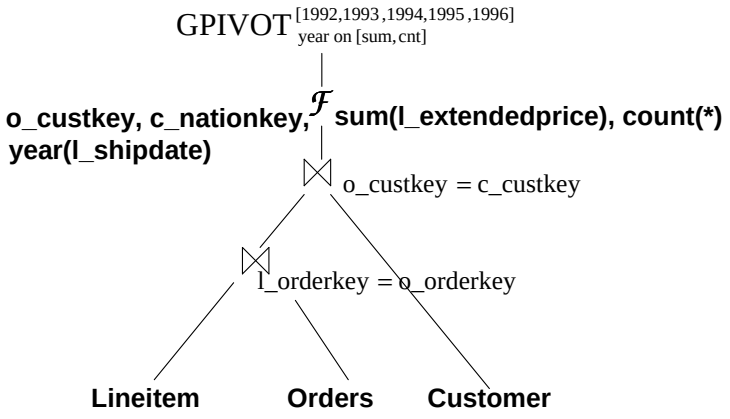


Figure 2.39: Aggregate Materialized View Definition for View3

Note that the cost analysis comparing the update propagation rules

```
CREATE  VIEW  view3
SELECT   o_custkey,c_nationkey,
         sum(case((year(l_shipdate) = 1992),l_extendedprice,⊥)) as 1992**sum,
         count(case((year(l_shipdate) = 1992),l_extendedprice,⊥)) as 1992**cnt,
         ....
         sum(case((year(l_shipdate) = 1996),l_extendedprice,⊥)) as 1996**sum,
         count(case((year(l_shipdate) = 1996),l_extendedprice,⊥)) as 1996**cnt,
FROM   Lineitem, Orders, Customer
WHERE   l_orderkey = o_orderkey AND o_custkey = c_custkey
         AND (year(l_shipdate) = 1992 OR ... OR year(l_shipdate) = 1996)
GROUPBY   o_custkey, c_nationkey
```

Figure 2.40: Aggregate Materialized View Definition View3 in SQL

with the insert/delete propagation rules for GROUPBY is the same for GPIVOT as in Section 2.7.2. Hence, in this section, we will directly discuss the experimental results to confirm such finding.

Maintenance under Source Deletions: We first consider the case of deleting tuples from the Lineitem table. The following three methods can be used to refresh the view. The first method is to perform full re-computation. The second method is to perform incremental maintenance using the update rules for GPIVOT and using the insert/delete propagation rules for GROUPBY, since GROUPBY is not at the top of the view query. The third method is to use combined update propagation rules for both GPIVOT and GROUPBY as in Figure 2.25.

Figure 2.41 depicts the maintenance results. Here y-axis denotes the execution cost of the maintenance expression in seconds. x-axis denotes the percentages of deletion on the Lineitem table. As can be seen from the figure, the maintenance method using our combined update rules (in

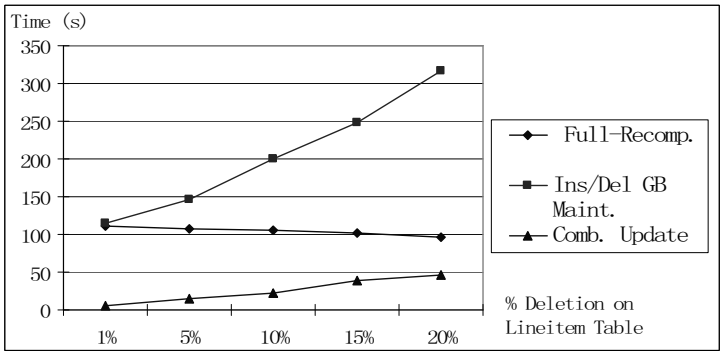


Figure 2.41: Maintenance of View3 under Deletion

Figure 2.25) performs one order of magnitude better than the one using the insert/delete rules for GROUPBY [Qua96]. The reason is that the insert/delete propagation rules for GROUPBY [Qua96] are non-trivial. They 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.

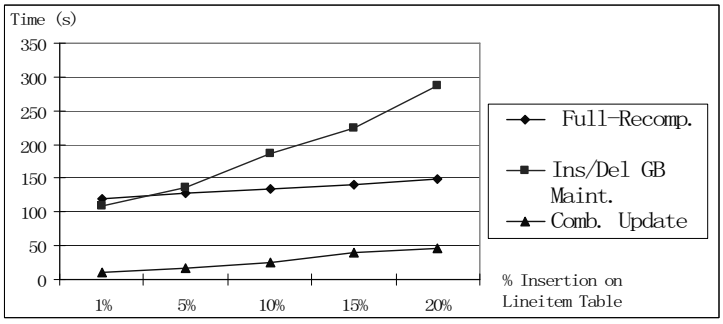


Figure 2.42: Maintenance of View3 under Insertion

Maintenance under Source Insertions: Next, Figure 2.42 depicts the maintenance results under the 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 preferable incremental maintenance choices than the maintenance method that considers GPIVOT and GROUPBY separately.

2.7.4 Summary of Experiments

In this section, first we have presented the cost analysis of various our proposed maintenance rules. We show that 1) the update propagation rules are preferable choices compared to the insert/delete propagation rules, and 2) the combined update propagation rules are preferable choices compared to considering each operator separately. Overall, our extensive experimental results validate our cost analysis and confirm the effectiveness of our proposed maintenance strategies.

2.8 Related Work

Incremental view maintenance has received considerable attention from the database community for the last decade [GMS93, GL95, MQM97, CGL+96, SBCL00]. In [GMS93], the authors propose algorithms for incremental view maintenance under bag semantics. They also support recursive views in Datalog. In [GL95], 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. Note that since PIVOT requires a key to exist in the source table, we assume set seman-

tics for non-aggregate views and bag semantics for aggregate views (since group-by columns form a key).

PIVOT is similar to GROUPBY in many ways [CGGL04]. In [Qua96], the authors propose the insert/delete and update propagation rules for the 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 SPJ+GROUPBY views. Or in other words, they do not support general views with any number of GROUPBY operators that can be anywhere in the query tree. Fortunately, we illustrate in this work that the PIVOT operator has a lot of interesting properties since it keeps the detailed data. As shown in this chapter, 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 flexible than the situation observed for the GROUPBY operator [CS94]. As a result, we illustrate that it is possible to derive an efficient maintenance plan. In fact, the same approach can be applied for efficient maintenance of general aggregate views with arbitrary number of GROUPBY operators.

In [CGGL04], 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 chapter, we address another important aspect of the PIVOT and UNPIVOT operators, namely, incremental view maintenance. We also show the necessity

of the GPIVOT definition for efficient view maintenance as well as for the optimization of queries with even just simple PIVOTs.

The PIVOT operator defined in [CGGL04] has slight semantic difference than our definition here. In that work, a pivoted output row may contain all $\perp$ columns. In this case, when we have a maintained pivoted tuple $(K, \perp, ..., \perp)$, we cannot simply delete it. One solution is to extend the definition of such a PIVOT operator to also include a column COUNT(*), which computes the number of rows for each K. We delete the view tuple $(K, \perp, ..., \perp, CNT)$ only when CNT becomes 0. When GPIVOT is above a GROUPBY operator, we can extend the GPIVOT definition to include a SUM(count(*)) column. We delete the view tuple $(K, \perp, ..., \perp, SUM(count(*)))$ only when SUM(count(*)) becomes 0.

The PIVOT and UNPIVOT operators studied in this chapter are first-order, since the output columns are pre-determined in the query by specifying the interested values. In [LSS99], 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 by all distinct values. The incremental maintenance SchemaSQL views was first studied in [KR02]. However, the technique is primarily tuple-based and not efficient for batch updates. In this work, we study the first-order version of such operators and thus we are able to derive efficient maintenance plans. It is an interesting future work to extend our proposed algorithms to support the maintenance of such higher-order pivot and unpivot operators, such as FOLD and UNFOLD [LSS99].

Chapter 3

Views with Complex Aggregate Functions

3.1 Our Contributions

In this chapter, we will propose a generic and comprehensive solution framework for management of views with complex aggregate functions as motivated in Section 1.3.2. In summary, the main contributions of this work are as follows:

- We propose a workarea function model for management of views containing complex aggregate functions. This framework greatly increases the system's extensibility to add the support of new functions, which is an especially useful feature for user-defined functions [WZ00].
- Based on this model, we propose a generic strategy for maintaining

aggregate materialized views with complex aggregate functions.

- We introduce a view matching algorithm to answer queries using such views also in a generic fashion using our workarea function model.

- Our workarea function model can also be extended to support multi-dimensional view maintenance and view matching. We can now also efficiently stack the computation of a multidimensional query.

- We have implemented our techniques in a prototype system of IBM DB2 UDB [Cha98]. The extensive experimental study demonstrates orders of magnitude performance improvement for incremental view maintenance and stacking computation for multi-dimensional queries.

The rest of this chapter is organized as follows. Section 3.2 presents the workarea function model. In Sections 3.3, we describe the techniques for incremental view maintenance with algebraic functions. Section 3.4 introduces a view matching algorithm for answering queries using such views. Section 3.5 describes how to support algebraic functions with multidimensional operators. We discuss the experimental results in Section 3.6. Section 3.7 reviews the related work.

3.2 Workarea Function Model

3.2.1 Properties of Aggregate Functions

In [GBLP96], the authors classify the aggregate functions into three categories in terms of their incremental maintainability, namely, distributive, algebraic and holistic. The definitions of these functions in [GBLP96] are as follows. Consider two data sets $\{X_i|i = 1..n\}$ and $\{Y_j|j = 1..m\}$.

- A function F is *distributive* if there is a function G such that $F(\{X_i\} \cup \{Y_j\}) = G(F(\{X_i\}), F(\{Y_j\}))$.
- A function F is *algebraic* if there are functions G and H such that $F(\{X_i\} \cup \{Y_j\}) = G(H(\{X_i\}), H(\{Y_j\}))$, where H is a function that computes M values.
- A function F is *holistic* if there is no constant bound on the size of the storage requirement for computing the sub-aggregate on $\{X_i\}$.

Note that since holistic functions, such as *Median*, are not incrementally maintainable, in this work, we will only consider distributive and algebraic functions. While the above definitions clearly state the properties of different types of aggregate functions, they do not provide a clear picture how the views with such aggregate functions can be managed. For example, it is not straightforward how the function H that computes M values is defined and used for view maintenance or view matching. In this section, we will introduce a fine-grained model, called *workarea function model*, that captures the properties of the incrementally maintainable functions while enabling generic view management techniques for such functions.

3.2.2 Workarea Functions

There are basically two ways to incrementally compute an aggregate function. The first method is to maintain the function F *directly*. That is, two maintenance functions, namely, f_F^+ and f_F^-, are required to incrementally compute F under either insert or delete case. Obviously, for distributive aggregate functions, no other extra function is required for f_F^+ and f_F^-. For algebraic aggregate functions, some *auxiliary* functions now become necessary.

Table 3.1 describes the functions that are maintained directly. Here NVar is Variance*Count and NCov is Covariance*Count. n denotes Count. s denotes Sum. nv denotes NVar. nc denotes NCov. s_x or s_y denotes the Sum of variable x or y. $\triangle$ denotes the computation of the function from the insert delta, e.g., $\triangle nv$ denotes the computation of nv from the insert delta. Similarly, $\triangledown$ denotes the computation of the function from the delete delta.

Workarea Func F	Aux. Func	Maint. Func f_F^+	Maint. Func f_F^-
Count(x)	-	$n + \triangle n$	$n - \triangledown n$
Sum(x)	-	$s + \triangle s$	$s - \triangledown s$
NVar(x)	Count, Sum	$f^+_{NVar}(nv, \triangle nv, s, \triangle s, n, \triangle n)$ (1)	$f^-_{NVar}(nv, \triangledown nv, s, \triangledown s, n, \triangledown n)$ (2)
NCov(x,y)	Count, Sumx, Sumy	$f^+_{NCov}(nc, \triangle nc, s_x, \triangle s_x, s_y, \triangle s_y, n, \triangle n)$ (3)	$f^-_{NCov}(nc, \triangledown nc, s_x, \triangledown s_x, s_y, \triangle s_y, n, \triangledown n)$ (4)
...	...	...	...

Table 3.1: Workarea Function Table

As can be seen from Table 3.1, since *Sum* and *Count* are distributive functions, they do not need any auxiliary function. *NVar* and *NCov* are al-

gebraic functions. Their maintenance functions require two auxiliary functions, namely, *Sum* and *Count*. Note that any of such auxiliary functions *must* also be defined in Table 3.1, otherwise the maintenance logic is incomplete since these auxiliary functions need to be maintained as well. The maintenance functions for *NVar* and *NCov* are defined in Equation (1) to (4) [CGL83]. Finally, we call the functions defined in the first column of Table 3.1 as *workarea* functions.

$$f^{+}_{NVar}() = nv + \triangle nv + \frac{n}{\triangle n*(n+\triangle n)} * (\frac{\triangle n}{n} * s - \triangle s)^2 \quad (1)$$

$$f^{-}_{NVar}() = nv - \bigtriangledown nv - \frac{n}{\bigtriangledown n*(n-\bigtriangledown n)} * (\frac{\bigtriangledown n}{n} * s - \bigtriangledown s)^2 \quad (2)$$

$$f^{+}_{NCov}() = nc + \triangle nc + \frac{n}{\triangle n*(n+\triangle n)} * (\frac{\triangle n}{n} * s_x - \triangle s_x) * (\frac{\triangle n}{n} * s_y - \triangle s_y) \quad (3)$$

$$f^{-}_{NCov}() = nc - \bigtriangledown nc - \frac{n}{\bigtriangledown n*(n-\bigtriangledown n)} * (\frac{\bigtriangledown n}{n} * s_x - \bigtriangledown s_x) * (\frac{\bigtriangledown n}{n} * s_y - \bigtriangledown s_y) \quad (4)$$

3.2.3 Derived Functions

The second method is to maintain function F *indirectly* by defining it based on the *workarea* functions. As depicted in Table 3.2, a function F is defined as a scalar function f over a set of workarea functions in Table 3.1. The maintenance of function F is achieved by first maintaining its workarea functions and then computing the new value of F from the maintenance results.

For example, $Variance$ is defined as $\frac{NVar}{Count}$, where *NVar* and *Count* are workarea functions defined in Table 3.1. Since the maintenance of *NVar* also requires *Sum* as in Table 3.1, all *(NVar, Sum, Count)* are the supporting workarea functions for *Variance*. *Variance* is thus maintained by first maintaining its workarea functions *(NVar, Sum, Count)* and then computing the

Derived Func F	Workarea Func W^F	Scalar Func f
Stddev(x)	NVar, Count, Sum	$f_{Stddev} = \sqrt{\frac{NVar}{Count}}$
Variance(x)	NVar, Count, Sum	$f_{Var} = \frac{NVar}{Count}$
Regr_Intercept(x,y)	Sumx, Sumy, Count, NVarx, NVary, NCov	$f_{Icpt} = \frac{Sumy}{Count} - \frac{NCov}{NVarx} * \frac{Sumx}{Count}$
Regr_Slope (x,y)	Sumx, Sumy, Count, NVarx, NVary, NCov	$f_{Slope} = \frac{NCov}{NVarx}$
...	...	...

Table 3.2: Derived Function Table

new value from the maintenance results.

Finally, as a general rule for function design, we should design a small set of *core* workarea functions such that they can be used to support a large class of other functions. The reason is simply that the derived function is much easier to implement (as only one scalar function is required). Such design is also beneficial for view matching as a small set of workarea functions in the view can be used to answer a variety of derived functions in the query. This workarea function model serves the basis for aggregate view management as we will elaborate in the next few sections.

3.3 View Maintenance using Workarea Functions

We first explore how to incrementally maintain the view based on the workarea function model in Section 3.2.

3.3.1 View Creation

Given an aggregate function F in the view, we first need to determine the method to maintain this function and if any additional function is necessary. Note that these additional functions need to be maintained as well. All such information can be exploited from our workarea function model in Section 3.2. That is, if F is a workarea function, then we maintain it directly using f_F^+, f_F^- in Table 3.1. If F is a derived function, then we maintain it indirectly by first maintaining its workarea functions and then compute the new F value from the maintained workarea functions.

Note that for both methods, some auxiliary workarea functions may be necessary to be added into the view in order to incrementally maintain F. Furthermore, these auxiliary workarea functions have to be incrementally maintained as well. As a result, more functions would have to be added. In fact, such information can be easily pre-derived and stored by adding all necessary functions into the *Auxiliary functions* in Table 3.1 and into the *Workarea functions* in Table 3.2. For example, in Table 3.2, although the definition of $Regr\_Slope$ only needs $NCov$ and $NVarx$ two workarea functions, other workarea functions, such as $Sumx$ and $Sumy$ are also included in order to incrementally compute $NCov$ and $NVarx$.

Based on the above discussions, we need to add the following workarea functions into the view definition when creating an aggregate materialized view. For any workarea function F, we add its auxiliary functions (Table 3.1) into the view. For any derived function F, we add its workarea functions (Table 3.2) into the view. Note that this step can be done auto-

matically using the workarea function model without any user actions. For example, when user creates the view (1.1) in Section 1.3.2, this view definition will be automatically rewritten by adding W^{slope}, i.e., the workarea function for $Regr\_slope$, which consists of six functions shown in Table 3.2.

$$
\begin{array}{ll}
\text{CREATE VIEW} & SalesAnalysis' \text{ AS} \\
\text{SELECT} & o\_custkey,\ regr\_slope(l\_extendedprice, l\_quantity)\ as\ qtyonprice, \\
 & W^{slope}(l\_extendedprice, l\_quantity)\ as\ wa,\ count(*)\ as\ cnt \\
\text{FROM} & lineitem,\ orders \\
\text{WHERE} & l\_orderkey = o\_orderkey \\
\text{GROUP BY} & o\_custkey
\end{array}
\tag{3.1}
$$

3.3.2 Incremental View Maintenance

In this section, we will first describe the existing view maintenance framework and then show how to extend this framework to support the maintenance of complex aggregate functions. In [CGL+96, MQM97], the authors proposed to maintain the views in two steps, namely, the *Propagate* phase and the *Apply* phase as shown in Figure 3.1. The *Propagate* phase computes the *final delta* from the base changes which represents the net effect of the changes to the view. The *Apply* phase integrates the *final delta* into the view. We now describe how to extend this basic framework to support complex aggregate functions using our workarea function model.

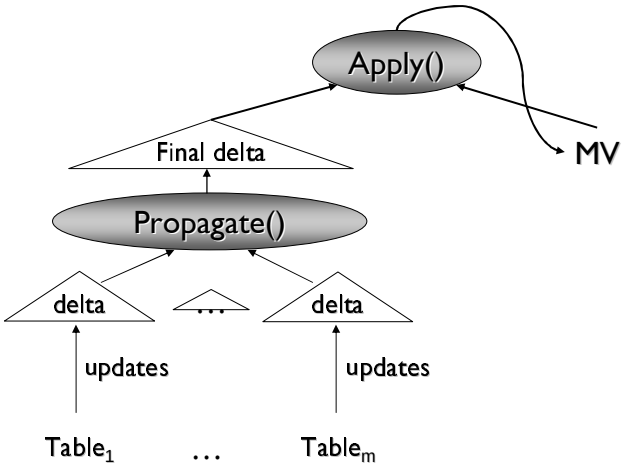


Figure 3.1: Incremental View Maintenance Framework

Propagate Phase

The *Propagate* phase computes the final delta [MQM97] from the base changes which represent the net effect of the changes to the view. A number of algorithms [GL95, GMS93] discuss the propagation of the deltas through each operator, such as select, join, group-by, etc.

Take the view (3.1) for example, assume there are some inserts $\triangle L$ on Lineitem table. The *final delta* is computed as in Query (3.2), which also includes the workarea functions as a result of the rewriting of view (3.1).

$$
\begin{array}{ll}
\text{CREATE VIEW} & FinalDelta \text{ AS} \\
\text{SELECT} & o\_custkey, \\
 & regr\_slope(l\_extendedprice, l\_quantity), \\
 & W^{Slope}(l\_extendedprice, l\_quantity), \\
 & count(*) \\
\text{FROM} & \triangle L,\ orders \\
\text{WHERE} & l\_orderkey\ =\ o\_orderkey \\
\text{GROUP BY} & o\_custkey
\end{array}
\tag{3.2}
$$

Apply Phase

The Apply phase will evaluate an equi-join between the view and the *final delta* on the group-by columns. Note that a left outer-join may be required as the insert *final delta* may create new groups to the view. If all the tuples of one group are deleted, then that group should be deleted (COUNT(*) is hence required.). If both the view and the *final delta* contain tuples of the same group, these two tuples will be combined to update the corresponding tuple in the view. For any workarea function F in Table 3.1, we can fetch the corresponding f_F^+ or f_F^- to incrementally compute F. For any derived function F in Table 3.2, we can compute its new value from maintained workarea functions.

A key feature of our maintenance framework is that, while user still needs to implement all workarea related functions, there is no need to hard code in the maintenance algorithm for any specific function. In order to add the support of new functions, we just need to insert the corresponding entries into Table 3.1 and 3.2. Clearly such extensibility is a very useful feature for user-defined aggregate functions [WZ00].

3.4 View Matching using Workarea Functions

View matching algorithms [CKP95, GL01, LMS95] decide if and how a view can be used to answer a query. [GHQ95, SDJL96, CNS99, ZCL+00] studied the problems of how to answer aggregate queries using aggregate views.

However these prior work only focus on handling distributive functions, such as *Sum* and *Count*. In this section, we introduce a generic view matching algorithm for how to answer queries with complex aggregate functions using views based on our workarea function model.

3.4.1 View Matching Background: Matching Framework

We first briefly review the existing view matching framework [ZCL+00] that our work is based on. This framework is built upon the Query Graph Model (QGM) [HFLP89], which is a structural representation of the SQL statements. In QGM, a query is represented as a rooted directed graph consisting of rectangular boxes. Each box implements one relational operator based on the specified input and output columns. In particular, the leaf boxes are the base tables. The intermediate boxes are the intermediate query results labeled by its operation, such as SELECT, GROUPBY, etc. SELECT box represents the select-project-join (SPJ) query, while GROUPBY box computes the aggregates, or multidimensional aggregates such as CUBE and Rollup [GBLP96]. The edges represent the data flow from the output columns of one box to the input columns of another box. The root box represents the final query result.

$$\begin{array}{ll} \textsf{SELECT} & o\_custkey,\ Sum(l\_extendedprice)\ as\ price \\ \textsf{FROM} & lineitem,\ orders \\ \textsf{WHERE} & l\_orderkey\ =\ o\_orderkey \\ \textsf{GROUP BY} & o\_custkey \end{array} \tag{3.3}$$

Figure 3.2 depicts the QGM for the SQL query (3.3). The leaf boxes

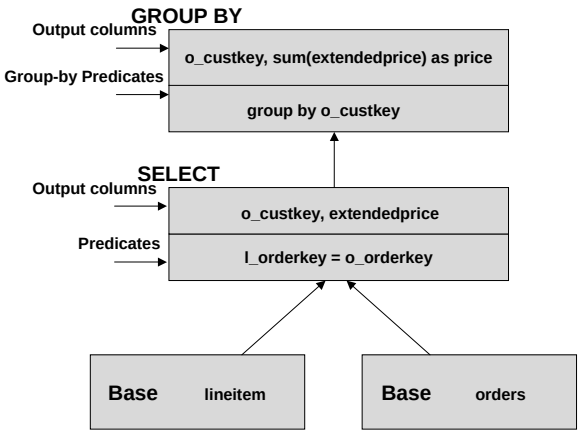


Figure 3.2: Query Graph Model (QGM) for Query (3.3)

are the base tables *lineitem* and *orders*. The next SELECT box evaluates an equi-join between two base tables. Finally, we perform aggregation in the GROUPBY box with group-by column *o_custkey*. It outputs two columns, namely, *o_custkey* and *sum of price*.

A general query matching framework based on QGM model is introduced in [ZCL$^+$00]. The framework is a bottom-up matching of pair of the query QGM and the view QGM boxes. The overall algorithm works as follows. Given a query QGM and a view QGM, we start matching from the bottom of the two QGMs. A view QGM box X is called (*subsumee*) if it provides all the data necessary to compute the corresponding query QGM box Y (*subsumer*). Adjustment is needed if the two boxes do not match exactly, which is called *compensation*. Such compensation makes the *subsumee* and *subsumer* semantically equivalent and has to be pulled up along the matching phases. The compensation to the root of the view QGM gives the rewritten query. The essence of this approach is that at any time, we only

need to consider the current two QGM boxes and the last compensation.

The general matching conditions of two boxes include first, the two boxes must be of the same type, and second, the *subsumer* expression can be derived from the output of the *subsumee*. For two SELECT boxes, the local predicates of the *subsumee* box must be semantically equivalent or weaker than that of the *subsumer* box. The *subsumee* box may contain *extra join* with the tables that do not exist in the *subsumer* box. However, such join must be an equi-join on key and foreign key in order to preserve the cardinality. The *subsumer* box may contain *rejoin* with the tables that do not exist in the *subsumer* box. Such *rejoin* should be included in the compensation and the join columns have to be in the final output of the *subsumer*.

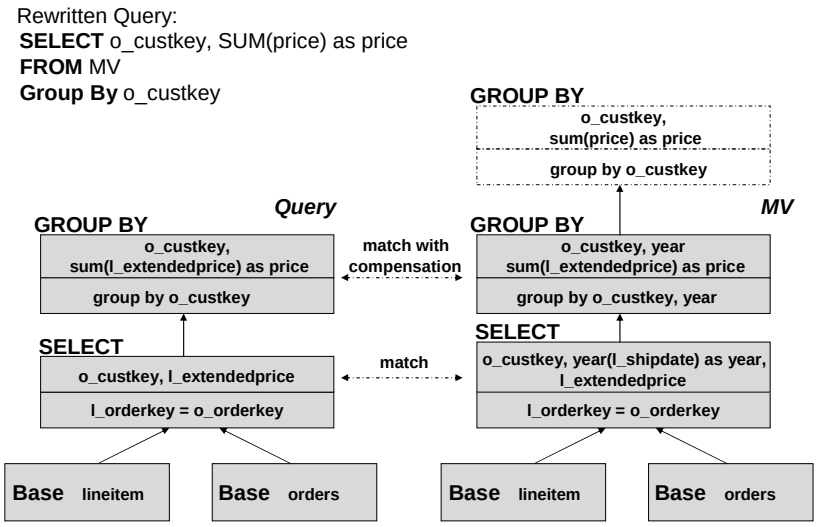


Figure 3.3: Matching Framework: An Example

Figure 3.3 describes a concrete example of such matching phase. The left part is the QGM for query (3.3). The right part with solid rectangles is

the QGM of a view defined in (3.4).

$$
\begin{array}{ll}
\text{CREATE VIEW} & V \text{ AS} \\
\text{SELECT} & o\_custkey,\ year(l\_shipdate)\ as\ year, \\
 & \ Sum(l\_extendedprice)\ as\ price, \\
\text{FROM} & lineitem,\ orders \\
\text{WHERE} & l\_orderkey\ =\ o\_orderkey \\
\text{GROUP BY} & o\_custkey,\ year(l\_shipdate)
\end{array}
\tag{3.4}
$$

The matching starts with the base tables and then the two SELECT boxes. The two join predicates match while the view QGM box contains extra columns which can be ignored. Then the matching goes up for the two GROUP BY boxes. The group-by columns in the view must be a superset of (or equal to) that in the query. If they are not equivalent, then *re-aggregation compensation* is required as the dotted box in Figure 3.3. Furthermore, an exact match of the aggregate function is expected, i.e., $Sum(X)$ in the query expects $Sum(X)$ in the view as well. The rewritten query is shown in the left-top in Figure 3.3.

3.4.2 Matching without Re-aggregation

From this section, we will describe our proposed view matching algorithm using workarea function model. We will only consider the matching of the two GROUPBY boxes since the aggregate functions are evaluated inside the GROUPBY box. In general, our proposed algorithm handles two matching categories, namely, matching *without re-aggregation* over the view and matching *with re-aggregation* over the view.

The general matching rules of the first category are that:

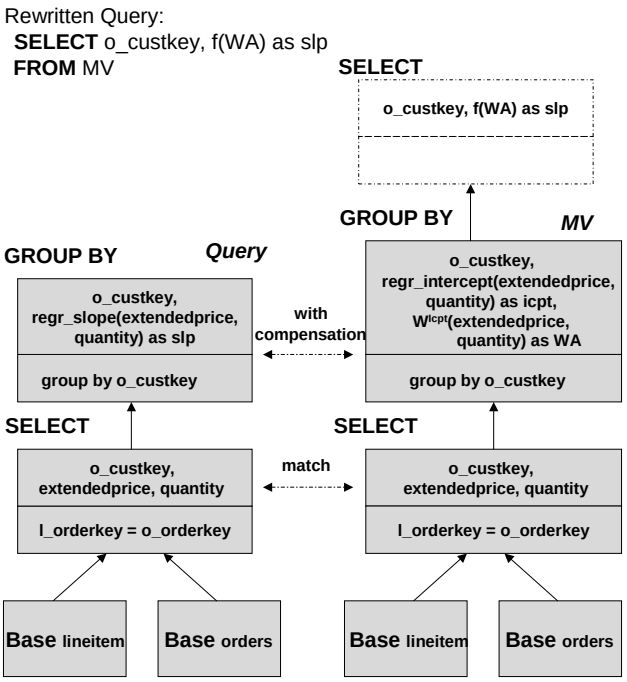


Figure 3.4: Match without Re-aggregation

- The sub-graphs of the query and the view match each other with SELECT-only compensation and the predicates of such SELECT-only compensation must be on the group-by columns.
- The group-by columns of the query and the view match *exactly*.

In this case, re-aggregation is not necessary. In fact, the aggregate functions in the query can be answered using the *workarea* functions in the view if there is no exact function match.

Figure 3.4 describes such an example. The left part is the query that computes *regr_slope*, while the view on the right part computes *regr_icpt*. A naive *exact* function match would fail in this case. However, note that the

workarea function for *regr_icpt* is also evaluated in the view for incremental maintenance, which can also be used to compute *regr_slope*. Hence the general rule is to compute *regr_slope* using this workarea function by the scalar function f in Table 3.2. The compensation and rewritten query is displayed in the figure.

Similar to the incremental view maintenance techniques in Section 3.3, the matching algorithm only needs to look up the corresponding entries in the workarea function table. No hard-coding for any specific function is necessary in order to enable such matching. This example clearly confirms the flexibility and extensibility of our workarea model for both view maintenance and view matching.

3.4.3 Matching with Re-aggregation

The second category is that the matching between the GROUPBY boxes of the query and the view requires *re-aggregation* compensation. This may happen when either the group-by columns of the query and the view do not match or the query has some *rejoin* with some additional tables that do not exist in the view. The row multiplicity may be affected due to such rejoins and hence re-aggregation is necessary.

Re-aggregation Compensation without Rejoin

We first consider the case when the query does not contain join with the tables that do not exist in the view, i.e., no rejoin in the query. More specifically, the general matching rules of this category are that:

- The sub-graphs of the query and view match with SELECT-only compensation (which does not contains join with other tables, i.e., no re-join).

- The group-by columns in the query and view GROUPBY boxes do not match exactly. In particular the view GROUPBY box must contain more group-by columns than that in the query box.

In this case, we have to re-aggregate over the view to the same granularity of the query. We note that direct re-aggregation over derived functions, such as *variance*, is not defined in Table 3.2. The reason is that the computation of the derived function is completely dependent on its corresponding workarea functions. Hence we propose to instead re-aggregate over the corresponding workarea functions.

Such re-aggregation, equivalently speaking, can be viewed as computing the workarea function under multiple insertions. More specifically, assume one workarea function F and a set of its input values $(w_1, w_2, ...w_n)$. Re-aggregating over w_i can be considered as multiple insertions of the input dataset from which each w_i computes. Hence, we can naturally define the re-aggregate function f_F^{agg} as an iterative application of f_F^+.

We follow a similar three-step for aggregate function definition as in [GBLP96, WZ00], namely, $(init, iter, final)$, to define f_F^{agg}. Here the *init* step computes the result when there is only one data input. The *iter* step defines how to compute the new result based on the previous result and the current data input. The *final* step defines how to compute the final result once all the input data are consumed.

Based on this formulation, our re-aggregate function f_F^{agg} can be easily defined as $(F, f_F^+, \_)$. That is, initially, the result is simply the input workarea function value. After that, we can apply f_F^+ to incrementally compute the new workarea function value. We do not need any *final* step, since after each iterative step we get the new workarea function value.

We can extend the workarea function table (3.1) to also contain an entry for f_F^{agg} as shown in Table 3.3. Note that user need not populate this entry at all since such re-aggregate function can be automatically derived.

Workarea Func F	Aux. Func	f_F^+	f_F^-	f_F^{agg} (init,iter,final)
Count(x)	-	f_{Count}^+ ...	f_{Count}^- ...	$(n, f_{Count}^+, \_)$
Sum(x)	-	f_{Sum}^+ ...	f_{Sum}^- ...	$(s, f_{Sum}^+, \_)$
NVar(x)	Count, Sum	f_{NVar}^+ ...	f_{NVar}^- ...	$(nv, f_{NVar}^+, \_)$
NCov(x,y)	Count, Sumx, Sumy	f_{NCov}^+ ...	f_{NCov}^- ...	$(nc, f_{NCov}^+, \_)$
...	...	...	...	...

Table 3.3: Extending Workarea Function Table with Re-aggregate Functions

Figure 3.5 depicts an example for how to use such re-aggregate functions to answer queries. The group-by columns in the view GROUPBY box are $o\_custkey$ and $year$, which are more than that in the query GROUPBY box. Hence we need re-aggregation compensation. Here W^{agg} consists of total six re-aggregate functions, one for each workarea function. To answer $regr\_icpt$ in the query, we first re-aggregate over the workarea in the view. After that, we compute the $regr\_icpt$ from the resulting workarea using the scalar function f in Table 3.2. Those two compensation operations are

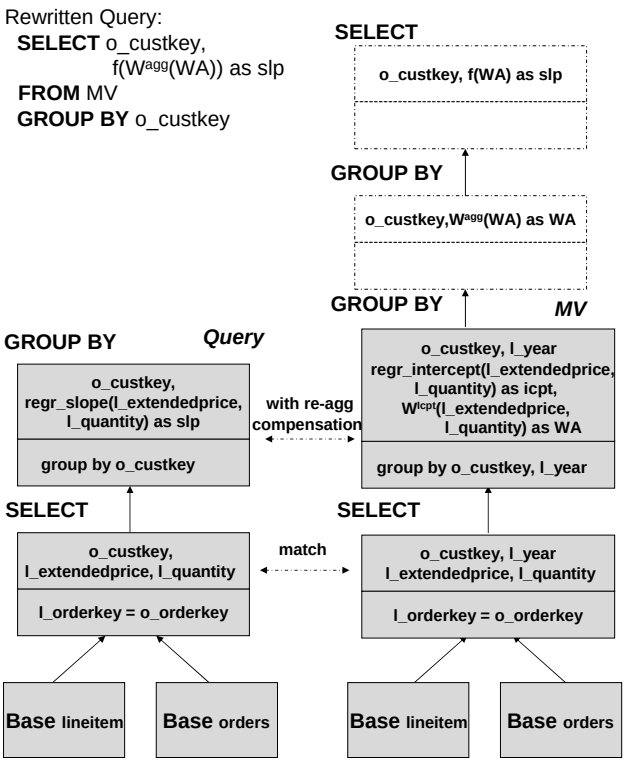


Figure 3.5: Re-aggregation Compensation without Rejoin

shown in the two dotted boxes in Figure 3.5.

Re-aggregation Compensation with Rejoin

Now let us consider the case when the matching of the GROUPBY boxes includes *rejoin* compensation from sub-graphs. In that case, we need a re-aggregation compensation unless the rejoin is an equi-join on key and foreign key and the group-by columns include the join key.

The matching compensation for such GROUPBY boxes would be first to pull up the compensation from the lower boxes, i.e., do a join between the

view and the extra base table (as well as other residual predicates). Then we re-aggregate over the join results.

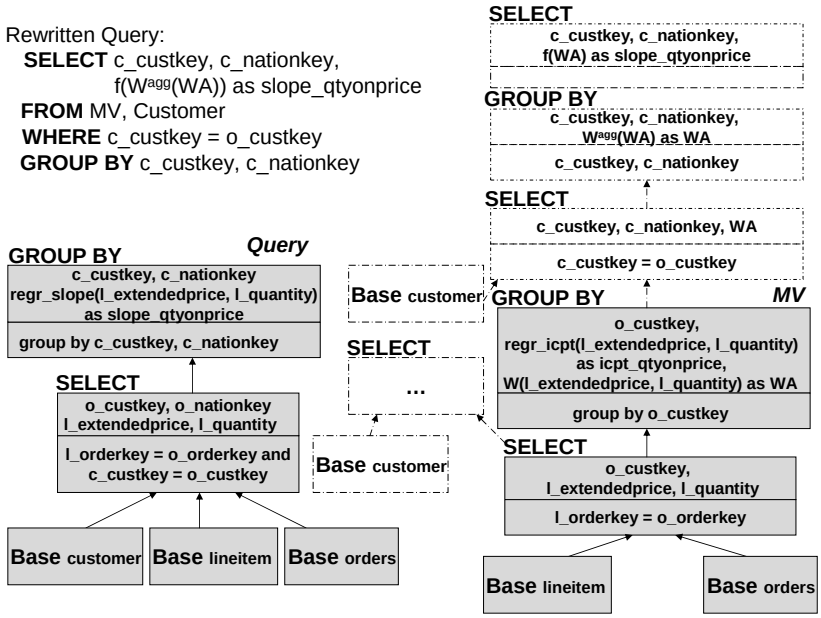


Figure 3.6: Re-aggregation Compensation with Rejoin

Figure 3.6 depicts such an example. Here the two SELECT boxes of the query and view do not match. The compensation includes a *rejoin* between the view and the table *Customer*. When the algorithm goes up to match the two GROUPBY boxes, it will first pull up the compensation from lower box, namely, evaluate a join between the view and *Customer*. It then re-aggregates over the workarea function and computes the final *regr_slope*. Note that in this particular example, if the *rejoin* with *Customer* is on key and foreign key (i.e., the cardinality is not changed), we can avoid such re-aggregation altogether.

3.4.4 Matching of Multidimensional Queries

The matching for multidimensional query is carried out similar to [ZCL+00]. First, the matching between a simple GROUPBY query and a multidimensional view is essentially to match the simple GROUPBY query with each *cuboid* of the multidimensional view. Here a cuboid is known as a simple group-by query block [GBLP96]. The techniques in Section 3.4.2 and 3.4.3 can be used to match the query and individual cuboid. The matching cuboid with fewest group-by columns will be selected from the multidimensional view due to its smallest size. For example, consider the following view and query:

$$
\begin{array}{ll}
\text{CREATE VIEW} & SalesAnalysis \text{ AS} \\
\text{SELECT} & o\_custkey,\ l\_year,\ l\_month, \\
 & regr\_slope(l\_extendedprice, l\_quantity)\ as\ slp, \\
 & W^{slope}(l\_extendedprice, l\_quantity)\ as\ wa, \\
 & count(*)\ as\ cnt \\
\text{FROM} & lineitem,\ orders \\
\text{WHERE} & l\_orderkey\ =\ o\_orderkey \\
\text{GROUP BY} & o\_custkey, ROLLUP(l\_year, l\_month)
\end{array}
\tag{3.5}
$$

$$
\begin{array}{ll}
\text{SELECT} & l\_year, regr\_intercept(l\_extendedprice, l\_quantity) \\
\text{FROM} & lineitem \\
\text{GROUP BY} & l\_year
\end{array}
\tag{3.6}
$$

Here the query (3.6) matches two cuboids of view $SalesAnalysis$ (3.5) (Note that here the join with Orders is on key and foreign key). The cuboid (o_custkey, l_year, NULL) from view $SalesAnalysis$ is picked to answer the query rather than the cuboid (o_custkey, l_year, l_month), since the size

of the cuboid (o_custkey, l_year, NULL) is always no larger than the size of cuboid (o_custkey, l_year, l_month) [ZCL+00]. The rewritten query is shown below.

$$\begin{array}{ll} \text{SELECT} & l\_year\ ,\ f(W^{agg}(WA)) \\ \text{FROM} & SalesAnalysis \\ \text{WHERE} & l\_month\ IS\ NULL\ AND\ l\_year\ IS\ NOT\ NULL \\ \text{GROUP BY} & l\_year \end{array} \tag{3.7}$$

Second, the matching between a multidimensional query and a simple GROUPBY view, on the other hand, requires a match exist between each cuboid of the query and the GROUPBY view. Intuitively, this requires the view to aggregate at a finer level compared to each cuboid of the query. The compensation will be a multidimensional query with the same grouping sets over the view.

While the above two cases are 1-n and n-1 matching, the matching between a multidimensional query with a multidimensional view is essentially a n-m matching. We require each cuboid of query to match some cuboid in the view. The matching between two cuboids can use the techniques in Section 3.4.2 and 3.4.3.

3.5 Cube Computation using Workarea Functions

We have addressed the incremental maintenance and view matching of multidimensional queries. Note that the processing of multidimensional queries can also benefit from our workarea function model.

There is a well-known technique, which we call *subset stacking*. This technique was first described in [GBLP96]. It utilizes the fact that a grouping set (GS^1) with groupby columns $e_1, e_2, ..., e_n$ might be used to compute any other grouping set (GS^2) whose group-by columns are a subset of $e_1, e_2, ..., e_n$.

Our workarea function model can be utilized generically to support this feature. Assume one cuboid Q is to compute function F over grouping sets GS^Q. Another cuboid Q' is to compute function F over grouping sets $GS^{Q'}$, where $GS^{Q'} \subset GS^Q$.

In order to compute Q' from Q instead of re-computing from scratch, we need to also compute the workarea function W^F for Q. The general strategy is to first re-aggregate over Q with group-by columns $GS^{Q'}$ and apply the workarea re-aggregate function $f^{agg}_{W^F}$ described in Section 3.4.3 over the workarea columns. After that, we compute function F for Q' from the resulting workarea using the scalar function f in Table 3.2.

For example, consider the following two cuboids.

$$
\begin{array}{ll}
\text{SELECT} & o\_custkey,\ l\_year,\ l\_month, \\
 & regr\_slope(l\_extendedprice, l\_quantity)\ as\ slp, \\
 & W^{slope}(l\_extendedprice, l\_quantity)\ as\ wa, \\
\text{FROM} & lineitem,\ orders \\
\text{WHERE} & l\_orderkey = o\_orderkey \\
\text{GROUP BY} & o\_custkey, l\_year, l\_month
\end{array}
\tag{3.8}
$$

$$
\begin{array}{ll}
\text{SELECT} & o\_custkey,\ l\_year \\
 & regr\_slope(l\_extendedprice, l\_quantity)\ as\ slp, \\
 & W^{slope}(l\_extendedprice, l\_quantity)\ as\ wa, \\
\text{FROM} & lineitem,\ orders \\
\text{WHERE} & l\_orderkey\ =\ o\_orderkey \\
\text{GROUP BY} & o\_custkey, l\_year
\end{array}
\tag{3.9}
$$

The grouping sets of cuboid (3.9) are a subset of that of cuboid (3.8). By our workarea function model, cuboid (3.9) can be computed from cuboid (3.8) in a stack fashion as follows:

$$
\begin{array}{ll}
\text{SELECT} & o\_custkey,\ l\_year \\
 & f(W^{agg}(WA))\ as\ slp, \\
 & W^{agg}(WA)\ as\ wa, \\
\text{FROM} & Cuboid\ (3.8) \\
\text{GROUP BY} & o\_custkey, l\_year
\end{array}
\tag{3.10}
$$

3.6 Experimental Evaluations

3.6.1 Implementation

We have implemented our proposed techniques in a prototype system of IBM DB2 UDB [Cha98]. We have two workarea tables (Table 3.1 and 3.2) for each aggregate function. They are extensible to accommodate any new aggregate functions. For each aggregate function, we create a hidden workarea column (of type Bit Data) which may contain multiple logical fields during the view creation time. During the propagation phase of view maintenance, we will compute the workarea functions over the deltas. During the apply phase of view maintenance, we first maintain the workarea functions

and then the derived functions. The appropriate maintenance functions are fetched on the fly from the two workarea tables. Note that our implementation also includes the support of multidimensional view maintenance, view matching and stacking cube computation. In this section, we will present the performance results for incremental view maintenance and stacking cube computation of multidimensional queries using our proposed techniques. The experiments are conducted on a TPC-H [TPC95] database with scalar factor 0.1, i.e., total around 100 Megabytes source data.

3.6.2 Incremental View Maintenance

In this section, we experimentally evaluate the incremental view maintenance in terms of performance. Basically, we compare our *incremental maintenance* method that uses the workarea-based techniques in Section 3.3 to the *full re-computation* method that re-computes the entire view.

There are several key aspects that will affect the incremental view maintenance performance, such as the size of view and the size of delta changes. We create four materialized views with different sizes. The general view definition is shown in Query (3.11).

$$
\begin{array}{ll}
\text{CREATE VIEW} & SalesAnalysis \text{ AS} \\
\text{SELECT} & [gb\_columns], \\
 & regr\_slope(l\_extendedprice, l\_quantity) \; as \; slp, \\
 & count(*) \; as \; cnt \\
\text{FROM} & lineitem, \; orders, \; customer \\
\text{WHERE} & l\_orderkey \; = \; o\_orderkey \; AND \\
 & o\_custkey \; = \; c\_custkey \\
\text{GROUP BY} & [gb\_columns]
\end{array}
\tag{3.11}
$$

The four materialized views differ in their group-by columns, i.e., [*gb_columns*] in Query (3.11). The [*gb_columns*] for the four views are

(1) *c_custkey, year(l_shipdate), month(l_shipdate)*,

(2) *c_custkey, year(l_shipdate)*,

(3) *c_custkey*,

(4) *c_nationkey*, respectively.

The sizes of the first three views are 50%, 10%, 1.6% compared to the fact table *lineitem* (600,000 rows), respectively. The last view has a constant size of 25 rows. They represent different sizes of materialized views used in practice.

First, we consider the deletion on the *lineitem* table [1]. In particular, we compare the view refreshing cost between full re-computation and incremental maintenance for each of the four views. The results are depicted in Figure 3.7. The x-axis represents the size of deletion in terms of the percentage to the original table. The y-axis represents the refresh cost in terms of execution time, which are normalized into relative units.

[1] We use the following SQL statement: 'delete * from lineitem where l_orderkey$\leq n$', where n controls the size of deletion.

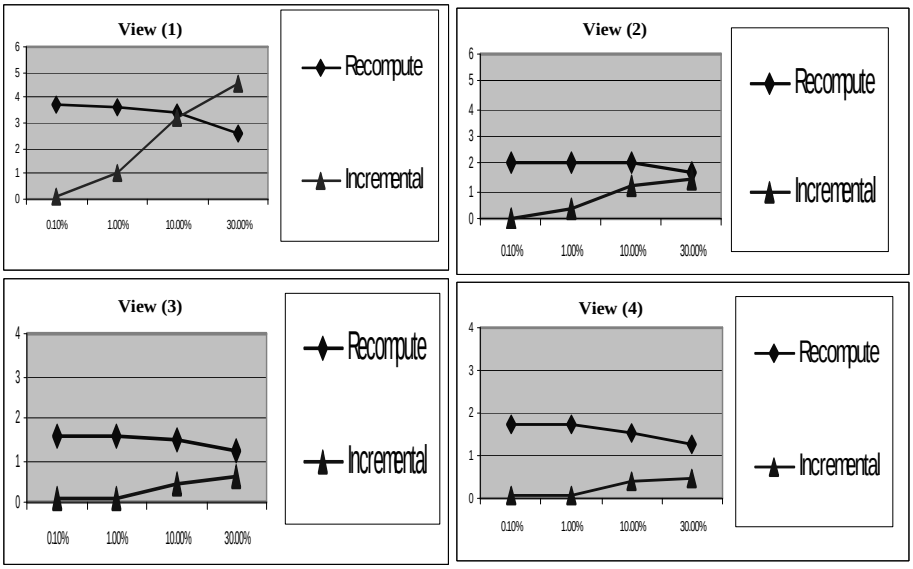


Figure 3.7: Refresh Cost under Deletion on Lineitem Table

There are a number of points that can be seen from Figure 3.7. First, the incremental maintenance approach significantly outperforms the full re-computation approach for all four views when the source changes are small, e.g., 0.1%. Second, the incremental maintenance cost increases as the size of delta change increases. The cross-point between the incremental maintenance and the full re-computation for view (1) is around 10% deletion. It increases to around 30% for view (2) and even higher for view (3) and (4). This indicates that incremental maintenance is more preferable for small-sized views. The reason is that the incremental view maintenance process consists of the *propagate* phase and the *apply* phase as described in Section 3.3. The *apply* phase will evaluate a join between the final delta and the view. The smaller the size of the view, the cheaper the join cost. In summary, our workarea-based view maintenance is a preferable method

for refreshing views when the change of the base table is small, which is common in data warehousing scenarios. Similar results can be found under the insertion case [2] as shown Figure 3.8.

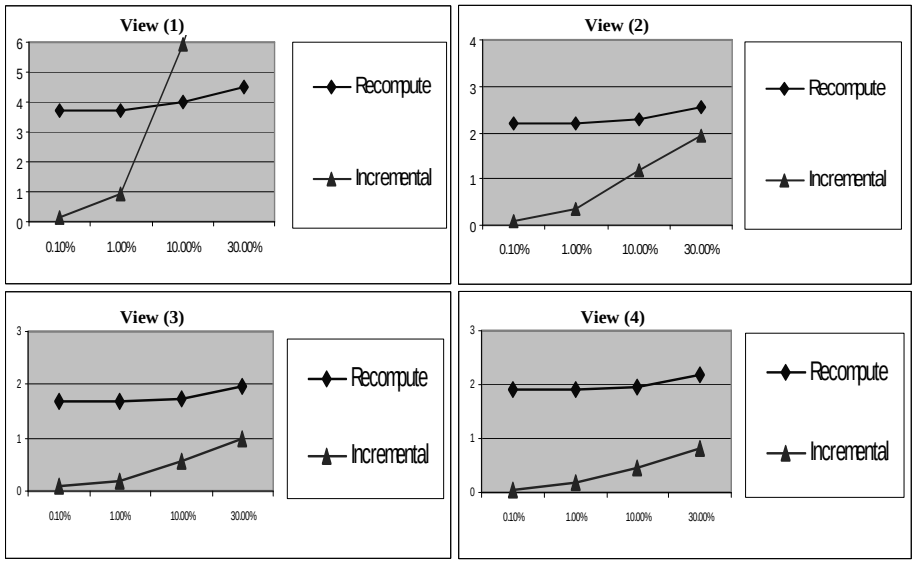


Figure 3.8: Refresh Cost under Insertion on Lineitem Table

We find that even under the same source delta size, the maintenance performance still could differ significantly from each other. This largely depends on how the view is affected by the source changes. Figure 3.9 depicts the maintenance of view (1) under 0.1% insertion on Lineitem table. The x-axis denotes the percentages of view tuples that are affected by this 0.1% insertion. The y-axis denotes the relative maintenance cost.

As can be seen from the figure, the larger the portion of the view that is affected, the more costly the incremental maintenance performance. The main reason is that the apply phase for incremental maintenance will eval-

[2] We generate the insert using the TPC-H data generator [TPC95].

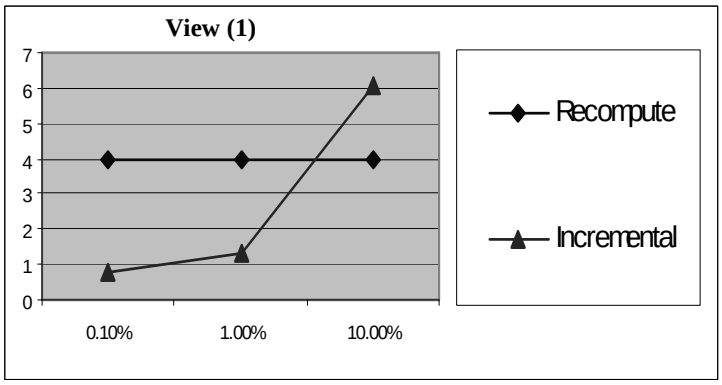


Figure 3.9: Refresh Cost under 0.1% Insertion on Lineitem Table

uate a join between the final delta and the view. If large portion of view is affected, then this join becomes more costly and more view tuples have to be updated.

Finally, our workarea-based view maintenance will add additional columns to the view. This also introduces extra overhead when we initially populate the view. In this experiment, we measure such extra cost by comparing the population of the view itself and the view with workarea columns. The population cost of the four views are depicted in Figure 3.10. The y-axis is the relative view population cost.

As can be seen, by adding the workarea columns, we increase the view population cost. However, such extra cost is relatively small. It is within 10% for all four views since the number of rows is the major factor for the query performance, which remains unchanged. Furthermore, since the view population is a one-time cost, i.e., it requires to be done only once initially, such extra cost is insignificant.

In conclusion, our workarea-based maintenance framework enables ef-

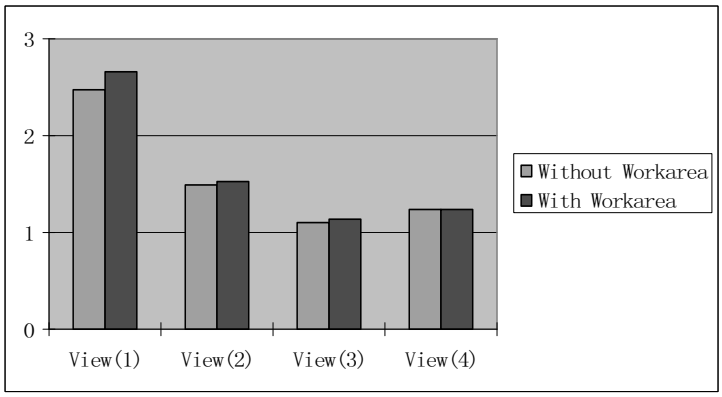


Figure 3.10: View Population Cost

ficient incremental maintenance of many practically useful views.

3.6.3 Stacking Computation of Multidimensional Queries

We now experimentally evaluate the cube computation techniques in Section 3.5. Consider the following multidimensional query (3.12):

$$
\begin{array}{ll}
\text{SELECT} & c\_custkey, year, month \\
 & regr\_slope(l\_extendedprice, l\_quantity)\ as\ slp, \\
 & count(*)\ as\ cnt \\
\text{FROM} & lineitem,\ orders,\ customer,\ nation \\
\text{WHERE} & l\_orderkey = o\_orderkey\ AND \\
 & c\_custkey = o\_custkey\ AND \\
 & c\_nationkey = n\_nationkey \\
\text{GROUP BY} & c\_custkey,\ ROLLUP(year, month)
\end{array}
\tag{3.12}
$$

We can either compute each cuboid from scratch, or we can stack the computation of each cuboid using workarea functions. In particular, we can first compute cuboid $Q_1 = (c\_custkey, year, month)$ with workarea

function W^{slope}. Then we compute cuboid $Q_2 = (c\_custkey, year)$ based on Q_1 by aggregating the workarea function. Finally, we compute cuboid $Q_3 = (c\_custkey)$ based on Q_2 by further aggregating the workarea function. The experimental comparison of these two methods are depicted in Figure 3.11.

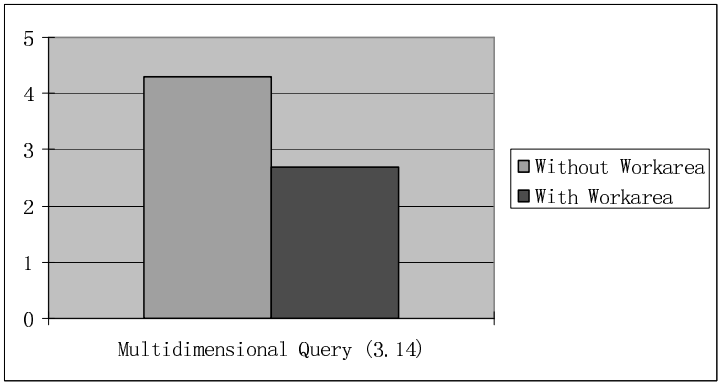


Figure 3.11: Stacking Cube Computation of Query (3.12)

As can be seen, the stacking of cube computation significantly outperforms the naive one. Note that when the grouping sets are more complex, or when an entire cube has to be computed, we expect the performance gain will be even larger. Thus our workarea-based method enables efficient stacking of cube computation for a large class of aggregate functions.

3.7 Related Work

Gray et al. [GBLP96] classifies the aggregate functions into three categories in terms of their stacking computation for multidimensional queries, namely, distributive, algebraic and holistic. The algebraic function is de-

fined as a global aggregate function over a local aggregate function that computes n values. Despite the similarities, there are some key differences compared to our model. Their model does not address the incremental maintenance of aggregate functions (under delete case) or how to define the aggregate functions to enable view matching. The n-value is similar to our workarea concept but it does not illustrate the property of the local aggregate function, global aggregate function nor the sub-aggregate functions that compute part of the n-value. Our work presents a finer modeling of aggregate functions. Compared to prior (init, iter, final) style for user-defined aggregate function definition [WZ00], our model provides a more efficient implementation for incrementally maintainable functions. Re-using existing functions becomes straightforward. Finally, based on our model, we are able to address a number of issues in a generic way, such as incremental view maintenance, view matching, multidimensional query processing, etc.

Incremental view maintenance has received considerably attentions from the database community for the last few years. Blakeley et al. [BLT86] defines when a SPJ view is incrementally maintainable. Griffin et al. [GL95] describes an algebraic framework for propagating deltas through each operator. Colby et al. [CGL$^+$96] introduces a deferred view maintenance algorithm separating the propagation phase and apply phase. Salem et al. [SBCL00] proposes a deferred maintenance algorithm capable of tuning the refresh window. [MQM97, CCH$^+$98, LSPC00, Qua96] studied how to maintain the aggregate or multidimensional views from theoretical or commercial systems point of views. These work primarily focus on the incremental

maintenance of distributive aggregate functions, such as sum and count. Palpanas et al. [PSCP02] proposes a selective re-computation approach to handle any type of aggregate function. The basic idea is to only selectively recompute those affected groups. Our proposed techniques avoid any recomputation and thus performs considerably faster for incrementally maintainable functions. In that work, the initial idea of using workareas for incremental view maintenance has been discussed. However, no details are provided and view matching is not considered.

Answering queries using views [CKP95, GL01, LMS95] has also been studied extensively in the literature. It has a wide range of applications. [GHQ95, SDJL96, CNS99, ZCL$^+$00] studied the problems of how to answer aggregate queries using aggregate views. These prior works consider how to answer the query with distributive functions using views. Oracle [ora] supports some complex aggregate functions such as Standard deviation and Variance. However, unlike our approach, the solution is not general. Our proposed view matching algorithm is based on our workarea function model. This offers a generic framework and flexibility to add new functions. Another clear advantage is that we need not implement any additional functions to enable such matching.

Recent SQL extension includes the introduction of three complex group functions, namely, grouping sets, rollup and cube, which allows multidimensional grouping can be expressed in a single group-by clause. The result of such complex group functions is essentially a union of multiple group-bys. Our proposed workarea model is orthogonal to the existing work on the incremental maintenance of multidimensional aggregate

views [MQM97], multidimensional query/view matching [ZCL+00] and stacking computation of multidimensional queries [AAe96]. A significant portion of those problems, i.e., with complex statistical measurements are supported by our techniques.

Chapter 4

Views in Dynamic Environments

4.1 Our Contributions

In this chapter, we propose a comprehensive solution for handling the new types of maintenance conflicts under both source data and schema changes, as motivated in Section 1.3.3. This solution enables data sources to autonomously commit all types of updates, which provides great flexibility for data integration in loosely-coupled environments, such as the Data Grid [JR03]. To our knowledge, this is the first complete solution to the view maintenance anomaly problems. In summary, the contributions of this work are:

(1) We identify three different types of maintenance anomalies, namely, those caused by the *data updates*, by the *data-preserving* schema changes

(such as for example, rename and restructuring operations that result in equivalent view rewritings), and by the *non-data-preserving* schema changes (such as for example, drop operations that result in non-equivalent view rewritings).

(2) For the first and the second types of anomalies, we extend the existing *compensation algorithms* for concurrent data updates, such as SWEEP [AESY97], to also handle *data-preserving* schema changes in order to restore the correct maintenance query results.

(3) We illustrate that the third type of anomalies, namely, those caused by *non-data-preserving* schema changes, are due to the violations of dependencies between the maintenance processes. We develop *detection algorithms* to first detect and *correction algorithms* to then correct any violated dependencies. This way we solve the anomalies caused by *non-data-preserving* schema changes.

(4) Given that our dependency correction algorithm above may generate mixed batches of data updates and schema changes, which cannot be handled by existing algorithms, we propose a *view adaptation* algorithm to handle such mixed batches. Furthermore, we show why such merged processing solves the anomaly problems and prove the correctness of this adaptation algorithm.

(5) We establish proofs of the correctness of our proposed *compensation* and our *dependency correction* algorithms. Furthermore, we also prove the correctness of our overall solution framework that it is capable of

maintaining views correctly even under a mixture of anomalies.

(6) As a proof of feasibility, we have implemented the above solution in our DyDa [CZC+01] prototype system. We have experimentally studied the impact of different types of anomalies and their corresponding solutions on maintenance performance. The experimental results show that our new concurrency handling strategy imposes a minimal overhead on data update processing while allowing for the extended functionality for view maintenance even under concurrent schema changes.

In the next section, we present the background material necessary for the remainder of the chapter. Section 4.3 describes our proposed architecture of the DyDa framework and explains our overall concurrency control strategies. Section 4.4 introduces a compensation algorithm to solve the concurrency caused by data updates and data-preserving schema changes. Section 4.5 formalizes the dependencies between the maintenance processes and their relationship to the concurrency caused by drop schema changes. An algorithm is proposed to detect and correct the violated dependencies. Section 4.6 introduces a new view adaptation algorithm to adapt multiple distributed schema changes and data updates required by the dependency correction algorithm. Section 4.7 discusses the experimental results. Section 4.8 reviews related work, while Section 4.9 concludes the paper.

4.2 Background Material

4.2.1 View Maintenance Techniques Revisited

We distinguish between three view maintenance tasks, namely, View Maintenance, View Synchronization and View Adaptation. View Maintenance [ZGMHW95, AESY97, SBCL00] maintains the view extent under source data updates. In contrast, View Synchronization [LNR02] aims at rewriting the view definition, or in a more general sense, evolving the schema mappings when the schema of a source has been changed [VMP03, YP05]. Thereafter, View Adaptation [NR99, GMRR01] incrementally adapts the view extent to again match the newly changed view definition. In this section, we first give a brief review of existing techniques. Then we provide an abstract model for these techniques that provides us with the necessary machinery to study the anomaly problems and later prove our solution techniques correct. Utilization of this abstraction highlights that our solution framework is a generic solution approach for resolving the anomalies. Our solution framework could fairly easily be plugged into any existing view system.

View Maintenance

View maintenance aims to incrementally maintain the view extent under a source data update (DU). This area has been extensively studied in the past [GMS93, ZGMHW95, SBCL00]. The idea is to issue a maintenance query based on the data update to calculate the delta change on the view extent. In Example 1, an *incremental maintenance query* (Query (1.4)) [ZGMHW95]

is issued for maintaining $\Delta Catalog$. In other words, we compute $\Delta V = \Delta Catalog \bowtie StoreItems$.

In this work, we generalize this maintenance process using the following abstraction: "$r(VD)r(DS_1)r(DS_2)...r(DS_n)\ w(MV)c(MV)$", where VD is the view definition, DS_i is the data source with index i, $r(DS_i)$ is the query sent to DS_i, $w(MV)$ and $c(MV)$ are write and commit to the view, respectively. For brevity, we denote such a maintenance process for DU as $M(DU)$. Note that any view maintenance algorithm would fit into this model in the sense that in general they need to issue queries to each individual source. For example, full re-computation is one maintenance method and it obviously fits into this model.

View Synchronization

View Synchronization [NLR98, LNR02], on the other hand, aims at evolving the view definition when the schema of the base relation has been changed. In that work [NLR98, LNR02], two primitive types of source schema changes (SCs) that may affect the view defined upon them are considered: *RenameSC* that renames the source attributes or relations and *DropSC* that deletes attributes or relations. Note that the addition of relations or attributes does not change the views. To maintain *RenameSC*, we can simply modify the corresponding view definition by using the new names. To handle *DropSC*, the basic idea is to find some alternative source to replace the dropped data.

The schema mapping evolution techniques in [VMP03, YP05] consider more flexible semi-structured data. In that work, more types of primitive

schema changes are considered. For example, beyond adding or removing elements, we can also add or remove constraints, or restructure the schema. In Figure 1.5, the BookStore database is *restructured*, which is a *data-preserving* schema change that breaks one table into two. The original *StoreItems* table is equivalent to the newly decomposed ones, namely, to *Store* $\bowtie$ *Item*. Thus we can rewrite the view as shown in Query (4.1).

```
CREATE VIEW   BookInfo AS
SELECT        Store, Book, I.Author, Price, Pub-
              lisher, Category, Review
FROM          Store S, Item I, Catalog C                (4.1)
WHERE         S.SID = I.SID AND I.Book = C.Title
```

In this work, we in general capture this view synchronization (or mapping evolution) process by the representation: $r(VD)w(VD)w(MV)c(MV)$. Note that here VD represents view definition and is an in-memory data structure. $w(VD)$ modifies that in-memory view definition in order to generate the subsequent maintenance query. The actual physical update of the view schema and view definition (e.g., updating the system catalog) is done in $w(MV)$, where MV is the materialized view.

Finally, note that the view synchronization process does not require the extent of the rewritten view to be always exactly equivalent to the original one. This is a reasonable assumption for information integration over a large scale and dynamic data sources [LNR02, VMP03, YP05]. In the EVE system [LNR02], for example, the authors suggest to replace the dropped

data by some alternative sources.

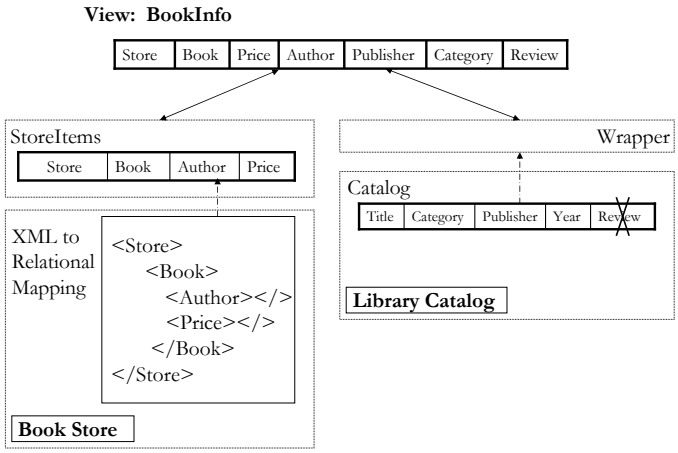


Figure 4.1: Drop of Review Attribute

For example, in Figure 4.1, the $Review$ attribute has been dropped. In this case, we may find some alternative library database that can be used for replacement. The main purpose of finding such alternative information is in the spirit of aiming to keep as much of the original view data as possible. Such decision can be made based on the source containment relationships, e.g., *equivalent, subset* or *superset relationships*, defined by the data integration administrator [LNR02]. For example, we may find an alternative source that contains similar review information and rewrite the view correspondingly as shown in Figure 4.2.

In comparison, the mapping evolution strategies [VMP03, YP05] do not attempt to find a replacement for the dropped data. They instead focus on how the mapping evolves for the remaining data. Hence for the drop operation in Figure 4.1, the result of mapping evolution may simply drop the $Review$ attribute in the view.

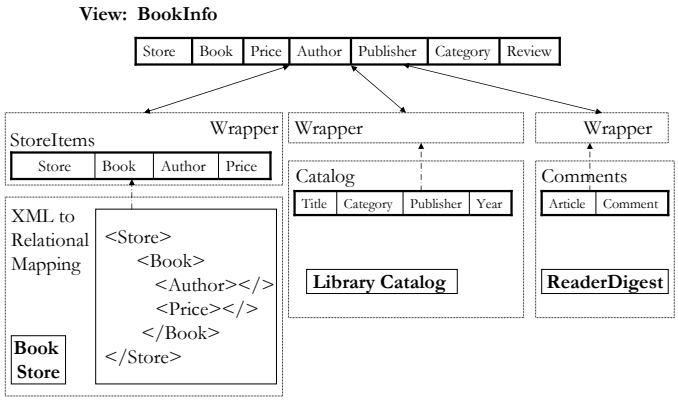


Figure 4.2: View Rewriting for Drop of Review Attribute

Whenever this view definition rewrite is a non-equivalent one, then this makes the next step, namely, view adaptation, necessary.

View Adaptation

As mentioned above, if the rewriting of the view definition is not equivalent to the one before, we need to adapt the view extent as well in order to be consistent with the new view definition. Such technique is called view adaptation [NR99, GMRR01]. Obviously, full re-computation is one method for view adaptation. Several incremental techniques [NR99, GMRR01] have also been proposed to incrementally adapt the view extent after the rewriting of the view definition.

In this work, we distinguish between two types of schema changes that differ in whether the result of view rewriting is *equivalent* or not. If the rewritten view is equivalent to the one before, then there is no need to do view adaptation work. For example, *schema restructuring*, such as

RenameSC [1] and the *normalization* operation in Figure 1.5, are both *data-preserving* schema changes. Thus the rewriting view is equivalent in both cases.

In the rest of this chapter, without loss of generality, we use *RenameSC* to represent data-preserving schema changes that result in equivalent rewritings of views and use *DropSC* to represent non-data-preserving schema changes that result in non-equivalent rewriting of views. The full maintenance process for a *DropSC* including both view synchronization and view adaptation can be generalized as "$r(VD)w(VD)r(DS_1)\, r(DS_2)...r(DS_n)w(MV)\ c(MV)$". For brevity, we denote such a maintenance process for $DropSC$ as $M(DropSC)$. The maintenance of a *RenameSC* can be generalized as "$r(VD)w(VD)\ w(MV)c(MV)$", denoted as $M(RenameSC)$.

As a final remark, these generalizations of various maintenance processes by sequence of read/write steps in this section are independent of any particular maintenance algorithms being applied to such steps. This allows us to make our proposed concurrency control strategy in this chapter general and thus applicable to a wide range of view maintenance algorithms [ZGMHW95, LNR02, AESY97, VMP03].

4.2.2 Types of View Maintenance Anomalies

We now formally define the anomaly problems in Example 1.

[1] This may not be true for *higher order* views, such as SchemaSQL views [KR04], where the data as well as the meta data can be queried. In this case, *RenameSC* can result in the data changes in the views. In this work, we only consider first order SQL views under bag semantics.

Definition 1 *Assume one update $w(DS_i)$ at source DS_i and one update $w(DS_j)$ at source DS_j. Maintenance of neither update has finished yet. We say that the update $w(DS_j)$ conflicts with the maintenance $M(w(DS_i))$ iff the source update $w(DS_j)$ is committed at DS_j* **before** *the query $r(DS_j)$ of $M(w(DS_i))$ is answered. We call such a conflict view maintenance anomaly.*

Note that the anomaly would never occur for $M(RenameSC)$ because no maintenance queries will be generated (Section 4.2.1). In other words, no anomaly would ever occur when maintaining the schema changes that result in an *equivalent* rewriting of view. However, the anomaly could occur during either $M(DU_1)$ or $M(DropSC_1)$ due to some other concurrent DU_2, $RenameSC_2$ or $DropSC_2$.

Based on the types of $w(DS_i)$ and $w(DS_j)$, we distinguish between three types of maintenance anomalies (six cases) as listed in Table 4.1. Among them, the anomalies I are caused by a concurrent data update DU, while the anomalies II are caused by a concurrent $RenameSC$. The anomalies of type III are due to a concurrent $DropSC$. The two cases in Example 1 are anomalies of types I and II, respectively.

Type	Cases
I	A DU_2 conflicts one maintenance query of $M(DU_1)$
	A DU conflicts one maintenance query of $M(DropSC)$
II	A $RenameSC$ conflicts one maintenance query of $M(DU)$
	A $RenameSC$ conflicts one maintenance query of $M(DropSC)$
III	A $DropSC$ conflicts one maintenance query of $M(DU)$
	A $DropSC_2$ conflicts one maintenance query of $M(DropSC_1)$

Table 4.1: Three Types of Maintenance Anomalies.

Notice that for the anomalies of type I in Table 4.1, we still could get some query results returned, however the results may be incorrect. For the anomalies of II and III in Table 4.1, we may not even be able to get any query result back due to the schema inconsistency between the maintenance query and the underlying sources. In this chapter, we will introduce a comprehensive solution framework that successfully solves all three types of anomalies.

4.3 View Management Framework

Figure 4.3 depicts the architecture of our DyDa view management framework. In this framework, similar to most prior work [ZGMHW95, AESY97], we assume that all data source transactions are local to their respective sources. Also every data update and schema change at a data source is reported to the view manager once committed (or alternatively the changes can be detected and extracted by the wrapper). The view manager collects and stores these source updates in the *Update Message Queue (UMQ)*.

The view manager is responsible for maintaining the views under source updates. For this, it makes use of three types of general view management services, namely, view maintenance (VM), synchronization (VS) and adaptation (VA) as introduced in Section 4.2.1. These three view management techniques generate the maintenance logic, i.e., maintenance queries, to allow the system to handle individual source data updates and schema changes. Such maintenance queries will be processed by the Query Processor. The Query Processor sends the maintenance query to the remote

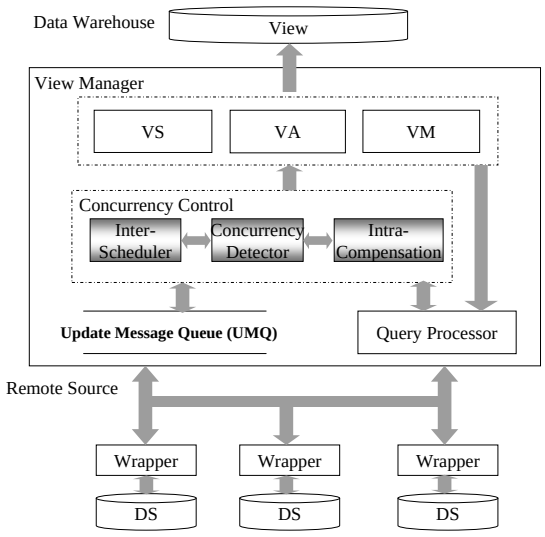


Figure 4.3: Architecture of DyDa Framework

sources and assembles the results. However, the anomaly problems as described in Section 4.2.2 may occur. In Figure 4.3, we highlight the concurrency control modules designed to solve the anomaly problems by shading them. We propose different strategies to handle the three types of anomalies listed in Table 4.1. In particular, we distinguish between two categories of concurrency control strategies, namely, the intra-maintenance compensation strategy and the inter-maintenance scheduler strategy.

Intra-maintenance compensation strategy solves the anomalies of type I and II. Its key property is that it will not abort any current maintenance process. Prior work in the literature [ZGMHW95, AESY97, ZGMW96, SBCL00] has focused on solving anomaly I by *compensating* the maintenance query result. Our *Intra-Compensation* module in Figure 4.3 extends this method

to compensate the maintenance query also in the case of conflicting *data-preserving* schema changes, such as *RenameSC* and *Normalization* in Figure 1.5. The main advantage of this technique is that we can solve the anomalies caused by *data-preserving* schema changes without aborting the current maintenance process (see Section 4.4).

Inter-maintenance scheduler strategy solves the anomalies of type III caused by *non-data-preservation* schema changes. In this case, aborts cannot be avoided. It requires to *abort* the current maintenance process and then to *globally* reschedule the maintenance processes. This issue arises due to the fact that most existing view maintenance algorithms process the source updates simply based on their arrival order. However, we will illustrate that such order is no longer appropriate and that in fact it may cause anomalies of type III. We formally identify that such anomalies are due to the violation of dependencies between the maintenance processes. We then introduce a dynamic maintenance scheduler that corrects such anomalies by rescheduling the maintenance order (see Section 4.5).

4.4 Intra-Compensation for Anomalies I and II

As a first step, we need to determine which source updates are concurrent to the current maintenance query. Similar to the prior work [AESY97, ZGMHW95], we rely on the following FIFO assumption in order to detect concurrent updates.

Assumption 1 *The network communication between an individual data source and the view manager is FIFO* [2].

Assumption 1 guarantees that when we receive one maintenance query result from a particular source, all concurrent updates at that source must have already arrived at the view manager. For instance, in Example 1, the concurrent source data update ΔS (1.a) or the schema change 'Normalize StoreItems table' (1.b) must arrive at the view manager before the erroneous maintenance query result.

By this assumption, we are able to identify the anomalies when receiving the maintenance query results. If the concurrent updates contain data updates, then anomaly of type I occurs. If the concurrent updates contain $RenameSCs$ (or data-preserving schema changes), then anomaly of type II has occurred. Finally, if the concurrent updates contain $DropSC$s (or non-data-preserving schema changes), then anomaly of type III has occurred.

When the concurrent updates are only data updates, i.e., only anomaly I occurs, a number of algorithms in the literature [AESY97, ZGMHW95, ZGMW96] propose to use compensation queries to remove the erroneous tuples from the query results. Take the SWEEP algorithm in [AESY97] for example. Suppose the maintenance query result is $\Delta R_j \bowtie R_i'$ instead of $\Delta R_j \bowtie R_i$. This erroneous query result can be corrected by $\Delta R_j \bowtie R_i' - \Delta R_j \bowtie \Delta R_i$, where ΔR_i are the concurrent data updates.

[2] It may not always hold in a real environment for some network where the transfer order is not guaranteed. However, we can solve this problem by timestamping each message [LS99]. For simplicity, in this paper, we assume Assumption 1 always hold.

However, this basic idea would fail if there is any concurrent *data-preserving schema change*, i.e., when anomaly II occurs. The reason is that the maintenance query now faces a schema conflict and may fail to return any query results. A straightforward solution is to rewrite the maintenance query using the new schema, e.g., using the new schema names. The issue now is that the concurrent data updates may be of a different schema if any schema change occurred in between.

Example 2 *In Example 1, assume the three updates, $\Delta StoreItem$ (1.a), the 'Normalization of StoreItems' (1.b) and $\Delta Item$ (1.c) are concurrent to the maintenance query $\Delta Catalog \bowtie StoreItems$. We may rewrite the query to $\Delta Catalog \bowtie Store \bowtie Item$ to address the schema change (1.b). While this rewritten maintenance query succeeds, we now still have to remove the effects of $\Delta StoreItem$ and $\Delta Item$ from the query results. Note that since $StoreItem$ does not ever occur in the current view definition, we cannot directly apply the existing data update compensation methods in [AESY97, ZGMHW95, ZGMW96].*

The idea to resolve this issue is to generate a sequence of compensation query for each individual delta. Such compensation query is assumed to be consistent with the delta schema "at that time". In Example 2, we first generate a compensation query $\Delta StoreItem \bowtie \Delta Catalog$, since the view schema at the time when $\Delta StoreItem$ occurred was $StoreItem \bowtie Catalog$. Next we generate a compensation query $\Delta Item \bowtie Store \bowtie \Delta Catalog$, since the view schema at the time when $\Delta Item$ occurred became $Item \bowtie Store \bowtie Catalog$.

We now give a brief analysis why this strategy works. The normalization schema change (1.b) only changes the query interface (schema) of the $StoreItem$ table but not the actual data. In this case, we can view $StoreItem$ as a virtual data source and any subsequent data changes as virtual deltas on $StoreItem$. Here $\Delta StoreItem$ directly shows the change of the virtual $StoreItem$ table, while $\Delta Item \bowtie Store$ indirectly shows the change to the virtual $StoreItem$ table. Any maintenance query sent to the virtual source $StoreItem$ can be corrected by compensating these virtual deltas using traditional compensation technologies [AESY97, ZGMHW95, ZGMW96] (we shall prove later).

Algorithm 1 depicts the pseudo-code of this IntraCompensation algorithm. The *QueryProcessing* function accepts the original maintenance query Q and possibly its rewritten query Q' (due to data-preserving schema changes, see line 6) as input parameters. It returns the correct query result QR after compensation. It first sends the maintenance query Q' (equal to Q originally) to the source(s). If this query failed due to data-preserving schema changes, then we rewrite this query and process the new query Q' again (lines 6 and 7). If this query failed due to non-data-preserving schema changes (line 10), then we have to reschedule the maintenance processes (Section 4.5). Finally if the query succeeds, then we apply $IntraComp$ algorithm in Algorithm 1 to solve the anomalies of type I and II. Note that if there is no non-data-preserving schema change, then the rewritten query will be guaranteed to eventually succeed after taking all data-preserving schema changes into account.

Algorithm 1 QueryProcessing and IntraCompensation Algorithms

```
1: Boolean QueryProcessing(Query Q, Query Q', QueryResults QR)
2: Boolean success = Issue(Q', QR);
   //Execute query Q', return TRUE if success, results in QR
3: ConUpdateSet CD=IdentifyConUpdates(QR); // Assumption 1
4: if success=FALSE then
5:    if CD does not contain DropSC then
6:       Q'=Rewrite(Q, CD);
7:       success = QueryProcessing(Q,Q',QR);
8:       return success;
9:    else
10:      InterScheduler();
11:      return FALSE;
12:   end if
13: else
14:   success = IntraComp(Q,QR,CD);
15:   return success;
16: end if
17: End Function
1: Boolean IntraComp(Query Q, QueryResult QR, ConUpdateSet CD)
2: ConDUSet CData;
3: ConSCSet CRen;
4: Boolean success = TRUE;
5: if CD does not contain RenameSC then
6:    success = Compensation_Algorithm(Q, QR, CD);
7: else
8:    while (CD not empty AND success)
9:       SameSchema(CD, CData, CRen);
10:      success = Compensation_Algorithm(Q, QR, CData);
11:      Q=Rewrite(Q, CRen);
12:   end;
13: end if
14: return success;
15: End Function
```

$IntraComp$ algorithm first detects if there is any concurrent data-preserving schema change. If not, it applies any existing compensating algorithm [AESY97, ZGMHW95, ZGMW96] for data updates to solve anomaly I. If any concurrent data-preserving schema change is identified, then the algorithm generates a sequence of compensation queries for the data updates of the same schema. In Example 2, the initial maintenance query is $StoreItems \bowtie \Delta Catalog$. The $IntraComp$ algorithm first extracts the concurrent updates, namely, $\Delta StoreItem$ and $Normalization$. Then we generate the compensation query as $\Delta StoreItem \bowtie Catalog$ and rewrite the maintenance query to $Item \bowtie Store \bowtie \Delta Catalog$ based on the normalization update. Finally we extract the data update $\Delta Item$ and generate the compensation query as $\Delta Item \bowtie Store \bowtie \Delta Catalog$.

Theorem 1 *Assume the data source state DS evolves to the state DS_u by updates $w(DS)$. The correct maintenance query result $r(DS)$ can be generated from $r(DS_u)$ using Algorithm 1 if $w(DS)$ contains only data updates and data-preserving schema changes.*

Proof: We first assume $w(DS)$ are: "$\Delta DS, DS \rightarrow DS^1, \Delta DS^1, DS^1 \rightarrow DS^2, ..., DS^{n-1} \rightarrow DS^n, \Delta DS^n$", where $DS^{i-1} \rightarrow DS^i$ are data-preserving schema changes for source DS, assuming from the source schema DS^{i-1} to DS^i, ΔDS^i are data updates at source DS on the schema DS^i.

The state of data source DS evolves correspondingly as follows: $DS \stackrel{\Delta DS}{\longrightarrow} DS' \stackrel{DS \rightarrow DS^1}{\longrightarrow} DS^1 \stackrel{\Delta DS^1}{\longrightarrow} DS^{1'} ... \stackrel{DS^{n-1} \rightarrow DS^n}{\longrightarrow} DS^n \stackrel{\Delta DS^n}{\longrightarrow} DS^{n'} = DS_u$. Here $DS^{i'}$ is the state after some data updates ΔDS^i applied to state DS^i, while DS^i corresponds to the state after some schema changes $DS^{i-1} \rightarrow DS^i$.

We further assume that the maintenance query $r(DS)$ is rewritten to $r^1(DS^1)$ after $DS \rightarrow DS^1$, to $r^2(DS^2)$ after $DS^1 \rightarrow DS^2$, ..., to $r^n(DS^n)$ after $DS^{n-1} \rightarrow DS^n$.

Based on the above notations, we note that when the schema change is data-preserving, we must have $r^i(DS^{i'}) = r^{i+1}(DS^{i+1})$. In other words, while the schema change $DS^i \rightarrow DS^{i+1}$ may change the query interface from r^i to r^{i+1}, the query results remain unchanged.

The rewritten query $r^n(DS^{n'})$, or $r^n(DS_u)$ succeeds since it is consistent with the current schema of the source DS. However, it includes the effects of any concurrent data updates, i.e., ΔDS, ΔDS^1, ..., ΔDS^n.

The compensation of ΔDS^n is straightforward. In fact, a number of existing solutions [AESY97, SBCL00, ZGMHW95] could be used here to compensate the query result $r^n(DS^{n'})$ to $r^n(DS^n)$ by removing the effect of ΔDS^n.

Next, since $r^n(DS^n) = r^{n-1}(DS^{n-1'})$ (as the schema change is data-preserving), we can apply the same technique to remove the effect of ΔDS^{n-1} from $r^{n-1}(DS^{n-1'})$ to get $r^{n-1}(DS^{n-1})$. The later equals $r^{n-2}(DS^{n-2'})$. By repeating the same process, finally, we can compensate ΔDS to get the desired query results $r(DS)$. ∎

Now from the proof itself, we can identify the properties of the schema changes that must hold in order for Algorithm 1 to work. That is, there must exist an equivalent rewriting of the maintenance query, i.e., from r^i to r^{i+1}, $r^i(DS^{i'}) = r^{i+1}(DS^{i+1})$. In other words, the query results must remain unchanged after view rewriting. Clearly, non-data-preserving schema

changes do not have this property. They thus cannot be solved by this strategy. We instead propose an extended solution in the next section.

4.5 InterScheduler for Anomaly III

Algorithm 1 guarantees that if there is no concurrent $DropSC$ (non-data-preserving schema changes), the compensation solution can successfully generate the correct maintenance query result *within* one maintenance process. However, if any concurrent $DropSC$ exists, i.e., anomalies of type III, then the solution no longer works. Figure 4.1 depicts such an example. Here the *Review* attribute of *Catalog* table is dropped. The maintenance query, $\Delta StoreItem \bowtie Catalog$, will fail and cannot be compensated using Algorithm 1 since the *Review* data is gone.

In this section, we demonstrate that the reasons for this are the violations of dependencies between maintenance processes. We first formalize the dependencies and point out their relationship to the anomalies III. Then we propose algorithms to solve them.

4.5.1 Dependencies among Maintenance Processes

Concurrent Dependency

There are two cases for anomalies of type III, namely, the maintenance process $M(DU_1)$ or $M(DropSC_2)$ conflicts with another $DropSC_3$. More formally, assume DU_1 or $DropSC_2$ occur at source DS_i and $DropSC_3$ occurs at DS_j, respectively. Their maintenance processes, namely, $M(DU_1)$, $M(DropSC_2)$ and $M(DropSC_3)$, are generalized as "$r_1(VD)r_1(DS_1)r_1(DS_2)$

$...r_1(DS_n)\ w_1(MV)c_1(MV)$", "$r_2(VD)w_2(VD)r_2(DS_1)r_2(DS_2)...\ r_2(DS_n)$ $w_2(MV)c_2(MV)$", and "$r_3(VD)\ w_3(VD)\ r_3(DS_1)\ r_3(DS_2)...r_3(DS_n)w_3(MV)$ $c_3(MV)$" respectively. (see Section 4.2.1). By Definition 1, a conflict between $r_1(DS_j)/DropSC_3$ or between $r_2(DS_j)/DropSC_3$ may arise.

Notice that there is also a read-write conflict on the view definition between these maintenance processes, i.e., $r_1(VD)/w_3(VD)$ and $r_2(VD)/w_3(VD)$. Interestingly, this conflict on the view definition is the reason for the conflict between $r_1(DS_j)/DropSC_3$ and also between $r_2(DS_j)/DropSC_3$. The rationale is that the queries $r_1(DS_j)$ and $r_2(DS_j)$ have been constructed based on the reads of the view definition $r_1(VD)$ and $r_2(VD)$, respectively. For instance, in Example 1.b, the maintenance query (2) is constructed over the $Catalog$ database based on the view definition query (1). If this view definition read $r_1(VD)$ (or $r_2(VD)$) conflicts with $w_3(VD)$, the constructed query $r_1(DS_j)$ (or $r_2(DS_j)$) may no longer reflect the actual schema of DS_j.

Definition 2 *Let $w(DS_i)$ and $w(DS_j)$ denote two updates committed on data sources DS_i and DS_j. The view manager has not finished maintenance for either of them. We say that maintenance process $M(w(DS_i))$ is* ***concurrent dependent (CD)*** *on maintenance process $M(w(DS_j))$, denoted by $M(w(DS_i)) \xleftarrow{cd} M(w(DS_j))$ iff $w(DS_i)$ is either a DU or $DropSC$ and $M(w(DS_i))$ contains a read view definition operation, while $w(DS_j)$ is a $DropSC$ and $M(w(DS_j))$ contains a write view definition.*

Concurrency dependency defines the relationship between maintenance processes over a critical resource, namely, the *view definition*. For instance, in Figure 4.1, there is a concurrent dependency "$M(\Delta S) \xleftarrow{cd} M(DropSC)$".

Note that there are several differences between the *concurrent dependencies* and *wait-for* dependencies in traditional transactions [BHG87]. First, the conflict is on the view definition not on the actual data tuples. Second, even if the maintenance of a sequence of updates is processed in a serial fashion, dependencies between them may still occur. The rationale is that the source updates are committed autonomously and thus may conflict with any ongoing maintenance process. Third, the dependency direction is always from a write to a read of the view definition since the concurrent schema change may invalidate the old view definition and consequently any of the ongoing maintenance processes.

Semantic Dependency

The materialized view is maintained consistent if it reflects some valid state of each data source [ZGMHW95]. Assume the state of DS evolves as $DS \xrightarrow{\Delta^1} DS^1 \xrightarrow{\Delta^2} DS^2$. It is important to maintain Δ^1 and Δ^2 in that order. If we maintain Δ^2 first, then the view reflects the data source state as $DS \xrightarrow{\Delta^2} (DS')$, which does not equal to either DS^1 or DS^2. In this case, the *strong consistency* [ZGMHW95], in which the view reflects the valid state of data sources in the same order, cannot be achieved. Furthermore, the view consistency may not even *converge*, i.e., the final state may be invalid too. For example, assume there are two updates from the same relation, a rename B to C and then a rename C to D. If we reverse their maintenance order, we cannot correctly maintain the view.

Thus it is necessary to preserve the processing order of updates from shared resources such as the same tuple, the same attribute or the same

relation in the examples above. For simplicity, we employ the conservative method of the same relation here. We now formally define this as a **semantic dependency** (**SD**).

Definition 3 *Assume two updates $w_1(DS_i)$ and $w_2(DS_i)$ from data source DS_i, then $M(w_2(DS_i))$ is **semantic dependent (SD)** on $M(w_1(DS_j))$, denoted by: $M(w_2(DS_i)) \xleftarrow{sd} M(w_1(DS_i))$ iff $w_1(DS_i)$ and $w_2(DS_i)$ refer the same relation and $w_1(DS_i)$ is committed before $w_2(DS_i)$.*

4.5.2 Dependency Properties

The two types of dependencies, *concurrent dependency* and *semantic dependency*, share an important property, namely, both represent constraints on the maintenance order between updates. Hence we now abstract them as one common concept.

Definition 4 *For two updates m_1 and m_2, we define $M(m_2)$ is **dependent on** $M(m_1)$, denoted by $M(m_2) \leftarrow M(m_1)$ if either $M(m_2)$ is* concurrent dependent *on $M(m_1)$ by Definition 2 or $M(m_2)$ is* semantic dependent *on $M(m_1)$ by Definition 3.*

Definition 5 *Given two updates m_1 and m_2 in the Update Message Queue (UMQ). If m_1 precedes m_2 in the UMQ, then we denote this by "$pos(m_1, UMQ) \prec pos(m_2, UMQ)$". We define the **dependency relationship** between $M(m_1)$ and $M(m_2)$ to be:*

1). ***independent** iff there is no dependency between $M(m_1)$ and $M(m_2)$ by Definition 4.*

2). ***safe*** *iff pos(m1, UMQ)* $\prec$ *pos(m2, UMQ) and all dependency orders between* $M(m_1)$ *and* $M(m_2)$ *by Definition 4 are* $M(m_2) \leftarrow M(m_1)$.

3). ***unsafe*** *iff pos(m1, UMQ)* $\prec$ *pos(m2, UMQ) and there is at least one dependency* $M(m_1) \leftarrow M(m_2)$.

Consider the example in Figure 4.1. The concurrent dependency is $M(\Delta S) \xleftarrow{cd} M(DropSC)$. However, since the $pos(DU, UMQ) \prec pos(DropSC, UMQ)$, this dependency is *unsafe* by Definition 5. Next, it is obvious that if $M(m_2)$ is dependent on $M(m_1)$, then the maintenance $M(m_1)$ *must* be processed *before* $M(m_2)$. For a *semantic dependency*, the required order is obvious as discussed in Section 4.5.1. For a *concurrent dependency*, as shown in Section 4.5.1, the write view definition operation has to be done *first* to solve the read-write conflict on the view definition. Since the concurrent schema change invalidates the view definition, rewriting it becomes critical.

Theorem 2 *An anomaly of type III occurs during the maintenance* $M(w(DS_i))$ *only if there is at least one* unsafe concurrent dependency $M(w(DS_i)) \xleftarrow{cd} M(w(DS_j))$.

Proof: An anomaly of type III implies an *unsafe* dependency, but not vice versa. The proof is straightforward. If an anomaly III occurs during the maintenance of $w(DS_i)$, by Definition 1, then there is a $DropSC$ denoted as $w(DS_j)$ that conflicts with $M(w(DS_i))$. By Definition 2, there is *concurrent dependency* $M(w(DS_i)) \xleftarrow{cd} M(w(DS_j))$. Since $M(w(DS_i))$ is scheduled before $M(w(DS_j))$, this *concurrent dependency* is unsafe.

However, an unsafe concurrent dependency does not necessarily lead to anomaly. The reason is that the maintenance query $r(DS_j)$ might have already been answered before $w(DS_j)$. ∎

Definition 6 *A Dependency Graph is a directed graph G=(V,E) with the set of nodes V denoting all updates m_i in the UMQ and with the set of directed edges E denoting the dependencies $e(m_i, m_j)$ between two updates m_i and m_j iff a* concurrent dependency *or* a semantic dependency *exists between $M(m_i)$ and $M(m_j)$.*

The complexity of identifying *concurrent dependencies* between maintenance processes is $O(mn)$, where m is the number of $DropSC$ and n is the number of updates. The reason is that each *concurrent dependency* involves at least one $DropSC$. In the worst case, one $DropSC$ would have one *concurrent dependency* to all other updates. Second, the complexity of building *semantic dependencies* between updates is $O(n)$, where n is the number of updates. To achieve this, we can create one bucket for each data source and scan the list of updates once. Thus the time complexity of building a *dependency graph* is $O(mn)$ + $O(n)$, i.e., $O(mn)$.

4.5.3 Cyclic Dependencies

A set of dependencies may comprise a cycle as illustrated by the example below. This is similar to the deadlock problem in the traditional serializability theory [BHG87]. Given the source relations from Figure 4.1, let us refer to the drop of $Review$ attribute as SC_1. Now assume the $StoreItems$

chooses not to map the $Author$ attribute as shown in Figure 4.4. We now refer this change as SC_2.

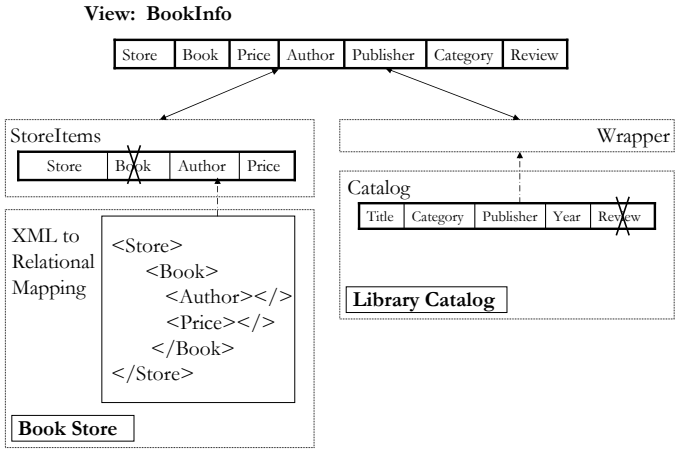


Figure 4.4: Example of Cyclic Dependencies

Assume these schema changes SC_1 and SC_2 have already been committed at their data sources. If we process SC_1 first, the view definition may be rewritten as in Figure 4.2. The basic idea is to find an alternative source to replace the data [LNR02]. The rule for finding such alternative source is based on the source containment relationship defined by user [LNR02].

However, this new view definition is no longer consistent with the sources since the attribute $Author$ no longer exists in $StoreItems$ table due to SC_2. Similarly, if we process SC_2 first, the view definition may be rewritten into Query (4.2), i.e., simply drop the $Author$ attribute, since we may not be able to find any alternatives. Again, the view definition is not valid since the attribute $Review$ is no longer available due to SC_1. We call such a situation a *cyclic* dependency.

$$\begin{array}{ll} \text{CREATE VIEW} & BookInfo' \text{ AS} \\ \text{SELECT} & Store,\ Book,\ Price,\ Publisher,\ Category,\ Review \\ \text{FROM} & StoreItems\ S,\ Catalog\ C \\ \text{WHERE} & S.Book\ =\ C.Title \end{array} \tag{4.2}$$

By Definition 2, there are concurrent dependencies $M(SC_1) \xleftarrow{cd} M(SC_2)$ and $M(SC_2) \xleftarrow{cd} M(SC_1)$. This comprises a cycle. Intuitively, the reason is that all these updates in a "cycle" are competing to modify the source schema. If the view manager rewrites the view definition based on any subset of them, the rewritten view definition is still not consistent with the underlying sources.

4.5.4 Detection and Correction of Unsafe Dependencies

After we detect an *unsafe* dependency between the maintenance processes, we need to reschedule the maintenance processes to turn the *unsafe* dependencies into *safe* ones (or equivalently speaking, we need to reorder the updates in the UMQ). We achieve this by sorting the *dependency graph* constructed during the detection phase.

Theorem 3 *Given a fixed number of updates, if the dependency graph is acyclic, we can obtain a maintenance order with all dependencies being safe.*

Theorem 3 holds since given an acyclic dependency graph, we can simply apply a *topological sort* algorithm [Tar72] to obtain a partial order of nodes. The complexity is $O(n + e)$, where n is the number of nodes (updates) and e is the number of edges (dependencies). This way we obtain an order of updates that has all dependencies in their safe direction.

However, if the dependency graph is cyclic as shown in Section 4.5.3, the *topological sort* algorithm cannot generate a partial order [Tar72]. For this, we first identify all cycles in the dependency graph (similar to identifying *strong connected components* in [Tar72], with complexity also $O(n+e)$). Traditional transaction processing [BHG87] breaks the cycle (or deadlock) by removing one of the nodes in the cycle, in other words, aborting one of the transactions. However, this strategy is not appropriate here because the source update is autonomous and hence not abortable. Instead of removing one node, we propose to *merge* these nodes (updates) into a *merged* one to be processed at one time. The intuition of the merge operation is that we have to take *all* the schema changes into account such that we can generate a new view that is consistent with *all* the sources. Otherwise, if the new view is not consistent with one or more sources, the maintenance query may fail.

The main idea of this merge operation is that for those merged maintenance processes, we actually reschedule their read/write operations in order to resolve their conflicts on view definition. We now give an example to show why cyclic anomalies are solved through merging. Assume two maintenance processes $M(SC_1)$ and $M(SC_2)$ with cyclic dependencies are $r_1(VD)w_1(VD)r_1(DS)w_1(MV)$ and $r_2(VD)w_2(VD)r_2(DS)w_2(MV)$, respectively. The merged processing of these two changes, namely, $M(SC_1\&SC_2)$, is actually as follows: $r_1(VD)w_1(VD)w_2(VD)r_3(VD)r_1(DS)w_1(MV)r_2(DS)$ $w_2(MV)$

That is, we first rewrite the view definition VD using $w_1(VD)$ and $w_2(VD)$. After that, based on the new view definition $r_3(VD)$, we start

the adaptation steps $r_1(DS)$ and $r_2(DS)$, which are now consistent with the source schema. As can be seen, there is no conflict in the maintenance process of $M(SC_1 \& SC_2)$, as the conflicting operations are appropriately ordered. Note that such reordering of operations is clearly always possible. This however requires a new view adaptation algorithm capable of atomically batching such combined updates by first performing all the view rewritings. This algorithm will be described in Section 4.6.

After removing all cycles in the dependency graph, we can apply the *topological sort* again to the now acyclic dependency graph to obtain a maintenance order with all dependencies being safe, as shown in Algorithm 2.

Algorithm 2 InterScheduler

```
1: Procedure InterScheduler()
2: Queue UMQ;
3: UMQ.Build_Dependency_Graph();
4: UMQ.TopologySort_with_CycleMerge();
5: End Procedure
```

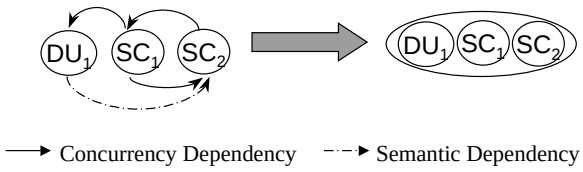


Figure 4.5: Examples of Unsafe Dependency Correction

Figure 4.5 depicts this dependency correction algorithm for our running example. Assume in the view (1) of Example 1, three updates occur, namely, first the data update DU_1 (ΔS on $StoreItems$ table) then the two schema changes SC_1 and SC_2 in Section 4.5.3, in that order. The maintenance of DU_1 failed since it encounters a schema conflict when processing the maintenance query. The $QueryProcessor$ module in Algorithm 1 found

that an anomaly of type III occurred due to SC_1. The InterScheduler in Algorithm 2 is thus invoked to solve this anomaly. As can be seen, the InterScheduler algorithm involves two steps, namely, detection and correction, to solve the anomaly of type III. It first identifies and builds the dependency graph. Since DU_1 and SC_2 are from the same source, there is a semantic dependency between them. Several *concurrent dependencies* are *unsafe* initially, such as $DU_1 \longleftarrow SC_1$ and $SC_1 \longleftarrow SC_2$. Then Figure 4.5 illustrates the dependency correction step by merging these three nodes into one big node to make the dependency graph acyclic. The final schedule is to maintain these updates altogether in one batch and the current maintenance of DU_1 is then *aborted*.

Theorem 4 *Given a number of updates, ΔX_1, ..., ΔX_m, no anomaly of type III will occur if the maintenance order is scheduled by the InterScheduler in Algorithm 2.*

Proof: The InterScheduler reorders the maintenance processes such that all the dependencies are safe. By Theorem 2, no anomaly III will occur if there are no unsafe dependencies. ∎

Finally, the idea to distinguish between *data-preserving* and *non-data-preserving* schema changes is to avoid always to abort the maintenance process whenever there is any concurrent schema change. The idea of rewriting the maintenance query in Algorithm 1 essentially is similar to the idea of dependency correction, namely, processing the *data-preserving* schema changes first.

4.6 View Adaptation for Merged Update Sets

In Section 4.5.4, the correction algorithm generates a maintenance order for a number of updates. However, if there exist cycles in the dependency graph, some updates will be *merged*. Such *combined* updates now could contain both schema and data updates over different data sources. State-of-the-art view maintenance algorithms [AESY97, SBCL00, LNR02] cannot maintain such mixed updates in one maintenance process.

As stated earlier in Section 4.5.4, the goal to merge the maintenance processes is to reorder their conflicting operations in order to resolve the anomalies. In particular, we must rewrite the view definition for all the schema changes first and then adapt the view extent. Our view adaptation algorithm follows the same principle. Below we introduce a batch extension of previous view maintenance algorithms, in particular, the EVE view synchronization algorithm [LNR02], for processing such merged updates. We assume first-order SQL views under bag semantics.

4.6.1 Preprocessing Step of the Source Updates

After *merging* cyclic dependent updates as described in Section 4.5.4, we have a *complex* update containing updates from multiple sources. We first partition these updates based on the data source DS_i that they originate from. After that, we further partition the updates from the same data source DS_i into two subgroups, namely, the data updates group defined as $< DU_i >$ and the schema changes group defined as $< SC_i >$. Without loss of generality, we define such a merged update as: $U = \{(< SC_1 >, <$

$DU_1 >), (< SC_2 >, < DU_2 >), ..., (< SC_n >, < DU_n >)\}$, where n is the number of data sources.

For such a merged update, first, the schema changes in $< SC_i >$ can sometimes be combined, e.g., if rename A to B and then B and C occur in the same data source, we could simply rename A to C. Second, the data updates may be inconsistent with their schema due to some schema changes between them. For example, assume two inserts into the same relation with a drop attribute in between, the latter tuple will have fewer columns. Thus our first step is to preprocess these updates in U from the same source to adjust these differences in order to enable us to maintain them in one batch.

Unifying Schema Changes: Given a group of schema changes $< SC_i >$ from one data source DS_i, we rewrite $< SC_i >$ to an equivalent group. This would optimize the maintenance, because those SC_i that have been removed need not be processed. Table 4.2 shows all possible combinations T(SC_1, SC_2) between two SCs (SC_1, SC_2) with SC_1 the row entry and SC_2 the column entry. Here R, S, T represent relations, while R.a, R.b, R.c represent attributes. If the entry of the combination in Table 4.2 is empty, then this means that these two operations cannot be combined. Hence both of them will be kept.

	$S \rightarrow T$	drop R	drop S	$R.b \rightarrow R.c$	drop R.b
$R \rightarrow S$	$R \rightarrow T$	-	drop R	-	-
$R.a \rightarrow R.b$	-	drop R	-	$R.a \rightarrow R.c$	drop R.a

Table 4.2: Combination Rules between Two SCs

Finally, we define $< SC'_i >$ as the set of schema changes after combining the schema changes in $< SC_i >$ pair wise using the rules above.

Unifying Data Updates: We then try to combine the data updates in $< DU_i >$, some of which might be of different schemata. To achieve this, we define: $< DU'_i >= \pi_{attr(R_i) \cap attr(R'_i)}(< DU_i >)$. That is, we project on the common attributes of both the original relation R_i and the new relation R'_i. These common attributes are actually the original R_i's attributes minus the dropped ones. The purpose of this projection is to make the $< DU'_i >$ schemata consistent with each other while still correct for maintenance. We justify this below.

Lemma 1 *All DU'_i must have the same schemata.*

Proof: We prove this by contradiction. Suppose that one tuple A contains one more attribute than another tuple B. This extra attribute must be either an added attribute of A or a dropped attribute of B. Note that the added attribute would only appear in the new state of relation R'_i, while the dropped attribute will only appear in the old state of relation R_i. Thus such attribute will not appear in $attr(R'_i) \cap attr(R_i)$. ∎

Example 3 *Assume a view V(A,B,C) defined as $R_1(A, B) \bowtie R_2(A, C)$. Suppose relation $R_2(A, C)$ has updates: +(3,4), add attribute D, +(4,5,6), drop attribute C, and -(5,7). We have R_2(A,C) and R'_2(A,D) and $attr(R_2) \bowtie attr(R'_2)$ = $\{A\}$. We get $< DU'_2 >= \pi_A < DU_2 >=< +(3), +(4), -(5) >$, which are schemata consistent. Now let's examine the new view definition V^{new}. A possible rewriting*

might be $V^{new}(A,B,C) = R_1(A,B) \bowtie \pi_{A,C}(R_2'(A,D) \bowtie R_3(A,C))$. *Since only attribute* $R_2'.A$ *is involved in the view definition* V^{new}, *we confirm that* $< DU_2' >$ *is sufficient for the view maintenance.*

Using the process described above, we can convert U into $U' = \{(< SC_1' >, < DU_1' >), (< SC_2' >, < DU_2' >), ..., (< SC_n' >, < DU_n' >)\}$. We characterize the relationship between $< SC_i' >$ and $< DU_i' >$ as follows:

- If $< SC_i' >$ contains *"Drop Relation* R_i*"*, then $< DU_i' >= \emptyset$ and $< SC_i' >$ = Drop Relation R_i.
- If $< SC_i' >$ contains a *"Drop Attribute"* operation, then both $< SC_i' >$ and $< DU_i' >$ might not be empty.
- If $< SC_i' >$ contains no $DropSC$, $< DU_i' >$ = $< DU_i >$.

In the next section, we will show that we can also safely have $< DU_i' >= \emptyset$ when $< SC_i' >$ contains a *"Drop Attribute"*.

4.6.2 Incremental View Adaptation Step

Now we are ready to incorporate these modified updates U' into the view. Recall that the view manager process involves two steps to incorporate schema changes, namely, view rewriting and then view adaptation. Below we describe how to maintain U' using these two steps.

View Rewriting by View Synchronization: We first apply view synchronization to all $< SC_i' >$ in U', i=1..n. That is we rewrite the view definition for each schema change in $< SC_i' >$ [LNR02].

Assume the old view definition as $V = R_1 \bowtie R_2 \bowtie ... \bowtie R_n$ and the new view definition as $V^{new} = R_1^{new} \bowtie R_2^{new} \bowtie ... \bowtie R_n^{new}$, where R_i^{new} is the rewriting of R_i in the new view definition. We now describe the possible outcomes of such view rewriting.

If the updates contain *drop relation*, then by Section 4.6.1 we know that $< SC_i' >$ has only that *drop relation* after merging the schema changes. Thus the rewriting of this relation in the view definition is a replaced relation [3] [LNR02]. If the updates contain *drop attributes*, then either the attribute is simply dropped in the view (such as the drop of $Author$ attribute in Section 4.5.3), or alternative tables and additional joins may be needed (such as the drop of $Review$ attribute in Section 4.5.3). If the updates do not contain any $DropSC$, then R_i remains unchanged in the new view definition. In summary, we have each new source relation as:

$$R_i^{new} = \begin{cases} \pi_{R_i} R_i^{new} & : \ Drop\ Rel \\ \pi_{R_i}(R_i \bowtie R_i^1 \bowtie R_i^2 ... \bowtie R_i^m) & : \ Drop\ Attr\ w.\ Replacement \\ \pi_{R_i - Attr} R_i & : \ Drop\ Attr\ w.o.\ Replacement \\ R_i & : \ No\ DropSC \end{cases}$$

Let us take the cyclic dependency example in Section 4.5.3. The correct view definition taking both SC_1 and SC_2 into consideration is defined in Query (4.3). This definition is now consistent with each data source.

[3] Note that if we cannot find such a replacement, the view will become invalid.

CREATE VIEW	BookInfo AS	
SELECT	Store, Book, Price, Publisher, Category, Comment as Review	(4.3)
FROM	StoreItem S, Catalog C, Comments M	
WHERE	S.Book = C.Title AND C.Title = M.Article	

Incremental View Adaptation: After the view is rewritten, one straightforward way to maintain the view content is to do a full recomputation based on the new view definition. In order to incrementally adapt the view extent, we need to determine the delta change. Here the old view extent is $V = R_1 \bowtie R_2 \bowtie ... \bowtie R_n$ and the new view extent is $V^{new} = R_1^{new} \bowtie R_2^{new} \bowtie ... \bowtie R_n^{new} = (R_1 + \Delta R_1) \bowtie (R_2 + \Delta R_2) \bowtie ... \bowtie (R_n + \Delta R_n)$. Comparing the old and the new view extent, the delta change is:

$$\begin{aligned} \Delta V &= \Delta R_1 \bowtie R_2 \bowtie ... \bowtie R_i \bowtie ... \bowtie R_n \\ &+ R_1^{new} \bowtie \Delta R_2 \bowtie R_3 \bowtie ..... \bowtie R_i \bowtie ... \bowtie R_n + ... \\ &+ R_1^{new} \bowtie ... \bowtie R_{i-1}^{new} \bowtie \Delta R_i \bowtie R_{i+1} \bowtie ... \bowtie R_n + ... \\ &+ R_1^{new} \bowtie ... \bowtie R_i^{new} \bowtie ... \bowtie R_{n-1}^{new} \bowtie \Delta R_n. \end{aligned} \tag{4.4}$$

The ΔR_i is derived below based on the definition of R_i^{new}. Here R_i' is the new state of R_i after source updates.

$$\triangle R_i = \begin{cases} \pi_{attr(R_i)}(R_i^{new}) - R_i & : \ Drop\ Rel \\ \pi_{attr(R_i)}(R_i' \bowtie R_i^1 \bowtie \cdots \bowtie R_i^m) - R_i & : \ Drop\ Attr\ w.\ Replacement \\ \pi_{R_i - Attr} < DU_i > & : \ Drop\ Attr\ w.o.\ Replacement \\ < DU_i > & : \ No\ DropSC \end{cases}$$

4.6.3 Correctness of Adaptation Algorithm

Theorem 5 *Assume a view V is defined as $R_1 \bowtie R_2 ... \bowtie R_n$. We further assume a complex update U, which includes ΔR_1, ..., ΔR_n. Each ΔR_i evolves the table state as: $R_i \xrightarrow{\Delta R_i} R_i'$. The new view V^{new} based on our batching algorithm is consistent to the new source state R_i' in terms of both its schema and data.*

Proof: Assume the updates to be maintained in a batch are ΔR_1, ΔR_2, ..., ΔR_n. Each ΔR_i includes two update sets, namely, $< DU_i >$ and $< SC_i >$. The unifying step for $< SC_i >$ in Section 4.6.1 is trivial, i.e., the resulting $< SC_i' >$ is equivalent to $< SC_i >$. Obviously, after taking all the schema changes SC_i' into account, the resulting view V^{new} is consistent to the new source state R_i' in terms of its schema.

Next, we prove that V^{new} is consistent to the data state of R_i'. Assume the rewritten view as $V^{new} = R_1^{new} \bowtie R_2^{new} \bowtie ... \bowtie R_n^{new}$. Obviously, a full recomputation of V^{new} will reflect the correct data state of R_i'.

To incrementally maintain V^{new} from $V = R_1 \bowtie R_2 \bowtie ... \bowtie R_n$, it is known to compute ΔR_i^{new} as the set difference between R_i^{new} and R_i then apply Equation (4.4) to correctly compute ΔV. ∎

Theorem 6 *The proposed techniques achieve strong consistency for view maintenance.*

Proof: The proof of the overall solution can be derived based on all the theorems developed so far. We start with the static case, assuming there

are n source updates ΔU_1, ..., ΔU_n. The Inter-Scheduler technique in Section 4.5 will schedule these maintenance processes that is free of anomaly III (Theorem 4). Assume the resulting order of updates for maintenance is $\Delta U'_1$, ..., $\Delta U'_m$. The correctness of each maintenance $M(\Delta U'_i)$ without any anomalies is guaranteed by Theorem 5, which will generate a number of maintenance queries. Moreover, even if there are anomalies I and II when processing these maintenance queries, they can be compensated by Theorem 1. Hence, the materialized view reflects the correct state after each $M(\Delta U'_i)$. Finally, given the *semantic dependency* constraints posed by the scheduler, there is a partial order between $\Delta U'_i$. Thus the materialized view achieves *strong consistency* but not *complete consistency* because some of the intermediate states may be missing due to the merge step.

Now we consider the dynamic case, i.e., new updates occur during the maintenance. If the new updates are DUs or $RenameSC$s, then they can be compensated by Theorem 1. If the new updates are $DropSC$s, then the maintenance is aborted and we reschedule the maintenance processes. We then turn back to the static case. ∎

4.7 Experimental Evaluation

4.7.1 Experiment Testbed

We have implemented the above techniques in our DyDa [CZC+01] system, using Java as development language and Oracle8i as view servers and data source servers. In our experimental setting, there are six sources evenly distributed over three different source servers with one relation

each. Each relation has four attributes and contains 100, 000 tuples. The materialized join views are defined on these six source relations. They contain all twenty four attributes and reside on a fourth server. All experiments are conducted on four Pentium III PCs with 256MB memory each, running Windows NT and Oracle8i.

4.7.2 Individual Update Processing

We first study the individual update processing of view maintenance. We distinguish between two classes of updates, i.e., data updates and schema changes. We further distinguish between $RenameSC$ and $DropSC$, because we expect that they have significant differences in maintenance costs.

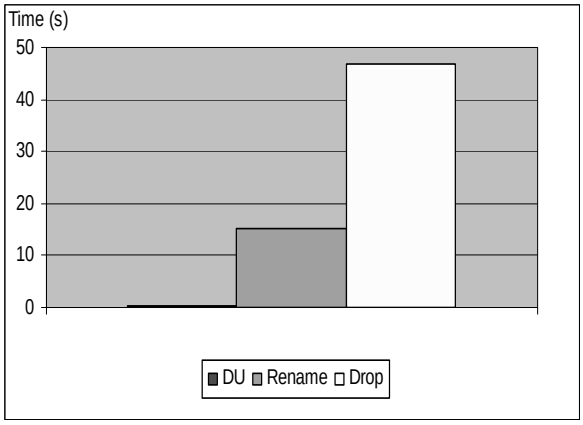


Figure 4.6: Comparison of Individual Update Processing Types

Figure 4.6 depicts the average view maintenance cost under our basic experimental setting, measured in seconds (depicted on the y-axis) for different types of updates. We find that the cost for DU maintenance is the least while the cost of $DropSC$ maintenance is significant. This is because

the latter invokes both VS and VA modules. This observation provides us some intuition that it is more costly to abort and re-process a $DropSC$.

In the next three experiments, we will study the system performance under various kinds of anomalies described in Section 4.2.2, all of which are supported in our DyDa system.

4.7.3 Study of Compensation for Anomaly I

Note that our DyDa system extends the ordinary view manager functionality to also deal with concurrent schema changes. We first study the overhead that such extended functionality may bring to the normal system's data update processing. We examine our *QueryProcessor* algorithm in Algorithm 1. When there is no concurrent $DropSC$, we can avoid the construction of dependency graph and correction step. Thus the extra cost to the existing data update maintenance algorithms is very small by having a $DropSC$ flag. In this experiment, we compare DyDa to SWEEP [AESY97] as the view maintenance algorithm under a number of concurrent data updates from distributed sources to measure the extra overhead that the DyDa framework may be imposing.

Figure 4.7 depicts the total view maintenance cost measured in seconds (depicted on the y-axis) under different numbers of source data updates (depicted on the x-axis). From the result, we find that the extra cost is almost negligible (less than 3%) for a number of data updates in our environmental settings. We thus conclude that the DyDa system imposes little extra cost on data update processing while offering added support for concurrent schema change processing.

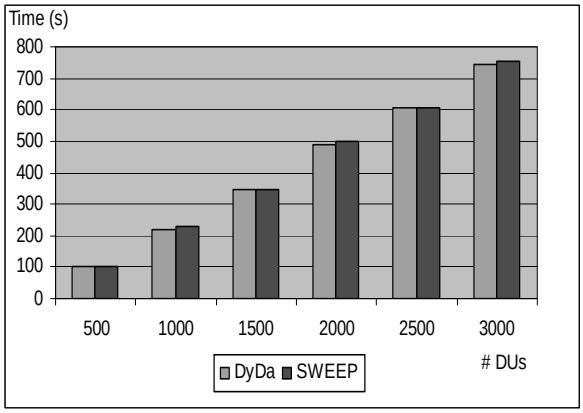


Figure 4.7: SWEEP vs. DyDa on Data Update Processing

4.7.4 Abort Cost of Anomalies II and III

Recall that a maintenance query may fail due to the existence of some concurrent schema changes, i.e., anomalies II and III. Once a concurrent $DropSC$ causes the query failure, the view manager has to abort all previous maintenance work and redo it again imposing some extra cost on the view maintenance process. While in comparison, if a concurrent $RenameSC$ occurs, we simply rewrite the query using new names and try the new query again without aborting any prior effort as described in Algorithm 1. The extra abort cost of anomaly II is thus less than that of the anomaly III.

In this experiment, we study the cost of all four cases of anomalies II and III. To observe the exact abort cost, we employ controlled cases here, i.e., one data update processing aborted by one schema change and one schema change processing conflicts with another one. Two different environmental settings are compared. First, we measure the maintenance cost of all updates by spacing them far enough so that each source update oc-

curs after the completion of the previous view maintenance step. This way they will not interfere with each other. This can be considered to be the minimum cost as no concurrency handling cost would arise. Second, we allow the anomalies to occur by spacing the updates close enough and measure the cost.

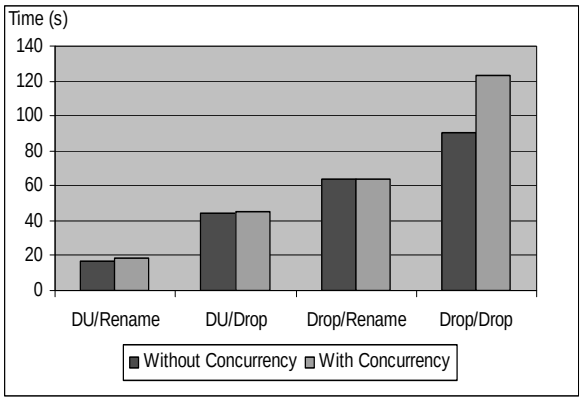


Figure 4.8: Abort Cost for Anomaly II and III

Figure 4.8 depicts the total view maintenance cost in terms of seconds (depicted on the y-axis) for the different types of anomalies II and III. We find that the extra cost of aborting $M(DropSC)$ by another $DropSC$ is more significant than any others. The reason is the complete abort of $M(DropSC)$, which is the most expensive maintenance process as observed in Section 4.7.1.

We also find that the abort cost of anomalies type II is small because the loss is just one maintenance query. All other maintenance efforts completed to that point can be kept since we can simply rewrite the query and try again. Finally, since data update maintenance is the least costly process,

even if it's completely aborted and redone again, the cost is still insignificant.

4.7.5 Mixed Update Processing

We now study how DyDa performs in an environment composed of a random mixture of both data updates and schema changes. We employ a mixture of updates with three $DropSC$, three $RenameSC$ and one hundred data updates over all six sources. We apply a worst case study here, i.e., no schema changes could be combined and no data updates could be discarded as described in Section 4.6.1. If so, the performance of our techniques should be better than we will find here. In this experiment, we vary the time interval between the $DropSC$, $RenameSC$ and data updates.

Figure 4.9 depicts the abort cost and the overall maintenance cost (which includes the abort cost) when only varying the time interval between $DropSC$ from 0s to 45s. 0s means that all updates flood into the view manager before any maintenance kicks in. From Figure 4.9, we see that this case has the best performance. This is because the system is able to correct all *unsafe* dependencies at once for all updates. Thus no anomalies type III would occur during maintenance processing. When the time interval between $DropSC$ increases, the new $DropSC$ could break the ongoing maintenance work. Hence the cost increases. The cost reaches a high peak when the new $DropSC$ always occurs near the end of the current $M(DropSC)$, resulting in the maximum abort cost. After the interval is larger than the maintenance time, there is no conflict between the $DropSC$. Hence the cost significantly decreases.

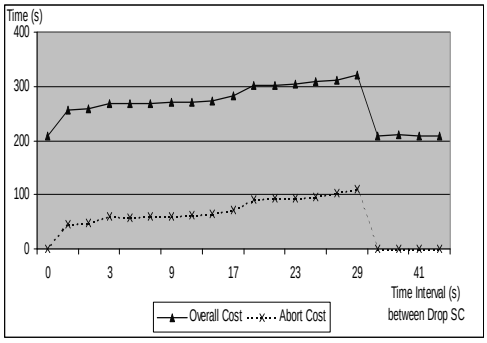


Figure 4.9: Varying Time Interval between DropSCs

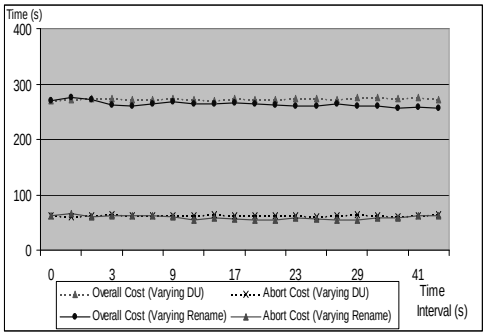


Figure 4.10: Varying Time Interval between DUs and RenameSCs

Next, we fix the time interval between the $DropSC$ to 10s, but we vary the time interval between data updates and $RenameSC$, respectively. Figure 4.10 depicts the abort cost and the overall maintenance cost for both cases. From the figure, we see that the system performance as well as the abort cost remain stable because the concurrency between $DropSC$ is not affected. Although the rate of the other two types of concurrency may vary, it seems not to affect the system performance much as observed in Sections 4.7.3 and 4.7.4.

4.8 Related Work

Schema mapping [MBR01, MHH00] specifies how to map the data from one schema to another to achieve interoperability of heterogeneous data sources. A variety of modern applications requires schema mapping as foundations, such as data integration for heterogeneous sources, XML to relational mapping or semantic Web [LGMT03]. With the popular usage

of WWW, the application environment becomes increasingly complex and dynamic. The data sources may change their schema, semantics as well as their query capabilities. In correspondence, the mapping or view definition must be maintained to keep consistent. In EVE [LNR02] system, the view definition evolves after the source schema changes. In [VMP03] the authors propose to incrementally adapts the schema mapping to the new source or target schema or constraints.

Maintenance of materialized view has been extensively studied in the past few years [AESY97, SBCL00, ZGMHW95, CGL+96, GL95, LMSS95]. However, most of these works assume a static schema. This is no longer a valid assumption in the dynamic environment. While in [AESY97, SBCL00, ZGMHW95], the authors proposed *compensation-based* solutions to remove the effect of concurrent data updates from query results, these solutions would fail under source schema changes. [ZR01] assumes a fixed synchronization protocol between the view manager and data sources to resolve the concurrency problem. This restricts the autonomy of sources in that the sources have to wait before applying any schema change. Our proposed solution successfully drops this restricting assumption.

In this work, we identify that the new concurrency problems are caused by the read-write conflicts on the view definition. Unlike traditional serializability theory [BHG87] that has full control to schedule the read/write operations to resolve the conflicts, in our context, the write (source update) is autonomous and hence locking is not an appropriate technique. Since the write is unabortable, we propose to process the related updates altogether to resolve the deadlock using a novel view adaptation algorithm.

Chapter 5

Conclusions and Future Work

5.1 Summary of Contributions

The overall contribution of this dissertation is to consider broader types of views than prior work that are useful for OLAP and data warehouse applications.

In particular, first, we consider views with PIVOT and UNPIVOT operators, which are of great interest in practice. We propose a novel framework for both query optimization and incremental maintenance of views with PIVOT and UNPIVOT operators. We find that a generalized operator, GPIVOT, not only has more powerful semantics but is also crucial to achieve the above goals. We propose the combination rules, swapping rules and various propagation rules for GPIVOT and GUNPIVOT in order to derive an efficient maintenance plan. Extensive performance evaluations confirm the effectiveness of our approach.

Beyond the contributions to solve the incremental maintenance of views

with PIVOT and UNPIVOT operators, our methodology, for the first time, also shows that the query transformation plays an important role for efficient view maintenance. Based on this pre-step of query transformation, our combined propagation rules for multiple operators demonstrate an effective method for efficient maintenance of more complex views.

The second contribution of this dissertation is that we propose a workarea approach for supporting views with complex aggregate functions. With our general and extensible workarea function model, we resolve a number of issues related to complex aggregate functions. First, we propose an incremental view maintenance method using workareas. Second, we also propose a generic view matching algorithm for answering queries using such views. Third, our workarea model efficiently supports the stacking of cube computation. Our real implementation in a prototype of IBM DB2 UDB proves the feasibility of our approach. Significant performance gain and extra functionalities are realized by our approach.

Beyond the contributions for managing views with complex aggregate functions, our workarea model also generically serves the basis for supporting user-defined aggregate functions. By our workarea function model, the user-defined aggregate functions can be more easily defined and implemented (one scalable function as opposed to always init-iter-term three functions [WZ00]). The sharing among user-defined aggregate functions can also easily be exploited.

The third contribution of this dissertation is that we illustrate that the view maintenance anomaly problems in a loosely-coupled environment correspond to the unsafe dependencies between source updates. We cate-

gorize the different types of dependency relationships that cause the anomaly problem. Then we propose a suite of detection methods for unsafe dependencies. We also introduce a dependency correction solution to eliminate unsafe dependencies. Finally we propose *Dyno* algorithm that combines both detection and correction strategies into one integrated solution. We show the correctness of *Dyno*, namely, that it enables the integrator to handle concurrent data and schema changes in a dynamic context.

The essence of this approach is that, Dyno is a general strategy for handling view maintenance concurrency problems, which is independent of any specific view maintenance algorithms or even the underlying data model. Hence it has the potential to be plugged into any view system.

5.2 Future Work

5.2.1 Answring Queries using Views with GPIVOT and GUNPIVOT Operators

There are a number of promising future directions beyond the work in Chapter 2, e.g., optimization and execution of GPIVOT and 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 [LSS99] and query matching for such views.

We now describe some initial ideas on query matching part for such views. One approach is to extend the view matching framework in Section 3.4, such that we can re-use the existing framework for other relational

operators and extend it by developing techniques for handling GPIVOT and GUNPIVOT operators.

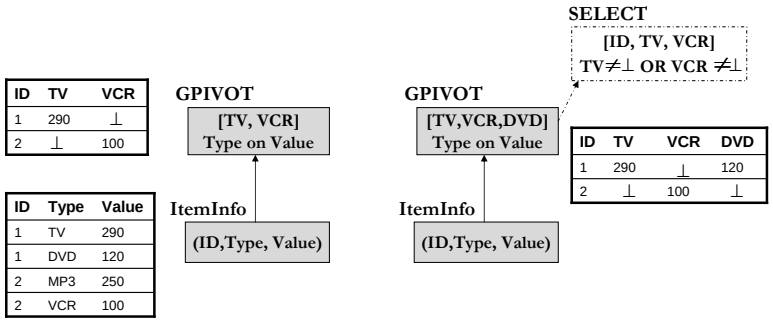


Figure 5.1: Matching Example for Views with GPIVOT

Figure 5.1 shows one matching example. The view definition is on the left side, while the query is on the right side. As can be seen, a bottom-up matching will start from two base tables. Then we try to match between two GPIVOT operators. Since the query contains more pivot parameters than the view does, we can compute the query from the view as shown on the top of the right side of the figure.

Figure 5.2 shows another matching example for aggregate views. The view definition is on the left side, while the query is on the right side. As can be seen, the matching between two group-by operators require compensation as shown in the figure. The matching between two GPIVOT operators require to first pull up the lower compensation as in Section 3.4, which results in aggregation over each pivoted output column. Then we match the two GPIVOT operators similar to the last example. The final rewriting is shown on the top of the right side of the figure. From these two examples, we can see that the views with GPIVOT operators can be used

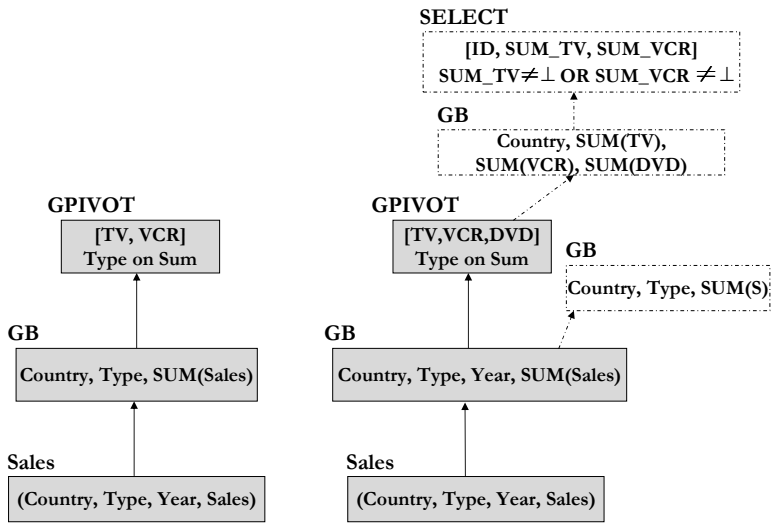


Figure 5.2: Matching Example for Aggregate Views with GPIVOT

to answer the queries containing such operators. Also it seems promising to extend the existing matching framework to support such views.

5.2.2 Top-K Aggregate, Window Aggregate and More

There are also a number of promising future directions beyond the work in Section 3. For instance, how to efficiently maintain holistic aggregate functions, as a simple example, we may want to keep Top-K values in order to efficiently maintain Min/Max under deletion. The challenge is how to pick the right K such that we can maximally reduce the full recomputation without maintaining too many extra data [YYY+03].

Also we note that windowing functions are frequently used by many applications (e.g., computing average sales every three months) [GBLP96]. How to efficiently compute windowing functions as well as how to manage

views with windowing functions are important issues. The challenge is that the insertion or deletion of one data point may affect many windows simultaneously if such changes are not ordered on the window attributes.

Lastly, experience with our experiments shows that, while the incremental view maintenance improves the view refreshing performance, it may also decrease the quality of view content due to floating point computation [ANS87b]. For example, the incremental maintenance of *variance* may result in significant error compared to full recomputation. For instance, the variance for the dataset $\{6.0, 8.0, 8.0*10^3\}$ under single-precision system [ANS87a] (about 7-decimal digits precision) is $1.42 * 10^7$ initially. After removing the data $8.0 * 10^3$, the incremental computation of variance results in 1.67 rather than 1.0 by recomputing from $\{6.0, 8.0\}$. The relative error is more than 60%. The resulting variance may even become negative, which is obviously incorrect. It is interesting future work for how to measure such errors in order to achieve both quality and performance for view maintenance.

5.2.3 XML and XPATH/XQuery View

There are many open and interesting research issues beyond the topics addressed in this dissertation. While this dissertation shows the first step to extend the views with complex aggregate functions and PIVOT/UNPIVOT operators, there are much richer sets of interesting views to be considered in practice.

For example, since the data sources are heterogeneous, complex extract-transform-load (ETL) processes are often necessary to achieve the integra-

tion of such data. How to manage views with some common ETL operations is still an unexplored but important problem. Furthermore, the source data models may not even be relational. In this case, a more flexible semi-structured query language, such as XPATH [W3C99] and XQuery [W3C05], would be necessary for the purpose of integrating such data.

The challenges of maintaining views defined using these languages are multi-fold. First, the update of the XML document is more complex than the relational insert, delete or update. The update may involve complex tree pattern predicate, followed by insertion, deletion or update of subtrees [TIHW01].

Second, unlike relational data, XML document is ordered. Such order requirement adds significant complexity for view maintenance. The reason is that while we may be able to compute some new trees to be inserted into the view, the hard part is to know *where* to insert them.

Third, XQuery supports result restructuring. This complicates the view maintenance problem further. The reason is that after restructuring, we no longer know where these data come from the original document, or in the words, the original structure information is lost after restructuring.

Appendix A

Correctness Proofs of Swapping Rules in Section 2.5

Equation (7) in Section 2.5.1:

$$\sigma_{\text{“}a_1^{i_1}**\ldots a_m^{i_1}**B_{l_1}\text{”}\ cp\ \text{“}a_1^{i_2}**\ldots a_m^{i_2}**B_{l_2}\text{”}}(GPIVOT^{[\{(a_1^i,\ldots,a_m^i)\}]}_{[A_1,\ldots,A_m]\ on\ [B_1,\ldots,B_n]}(V)) =$$
$$GPIVOT^{[\{(a_1^i,\ldots,a_m^i)\}]}_{[A_1,\ldots,A_m]\ on\ [B_1,\ldots,B_n]}(\pi_{K^1}[\sigma_{(A_1,\ldots,A_m)=(a_1^{i_1},\ldots,a_m^{i_1})}(V) \bowtie_{(K^1=K^2\ \wedge\ B^1_{l_1}\ cp\ B^2_{l_2})}$$
$$\sigma_{(A_1,\ldots,A_m)=(a_1^{i_2},\ldots,a_m^{i_2})}(V)] \bowtie V)$$

Proof for Equation (7):

$$\sigma_{\text{“}a_1^{i_1}**\ldots a_m^{i_1}**B_{l_1}\text{”}\ cp\ \text{“}a_1^{i_2}**\ldots a_m^{i_2}**B_{l_2}\text{”}}(GPIVOT^{[\{(a_1^i,\ldots,a_m^i)\}]}_{[A_1,\ldots,A_m]\ on\ [B_1,\ldots,B_n]}(V)).$$
$$= (\pi_{K^1}[\sigma_{(A_1,\ldots,A_m)=(a_1^{i_1},\ldots,a_m^{i_1})}(V) \bowtie_{(K^1=K^2\ \wedge\ B^1_{l_1}\ cp\ B^2_{l_2})} \sigma_{(A_1,\ldots,A_m)=(a_1^{i_2},\ldots,a_m^{i_2})}(V)]) \bowtie$$
$$(GPIVOT^{[\{(a_1^i,\ldots,a_m^i)\}]}_{[A_1,\ldots,A_m]\ on\ [B_1,\ldots,B_n]}(V)).$$

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

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

Equation (8) in Section 2.5.1:

$$GPIVOT^{[\{(a_1^i,\ldots,a_m^i)\}]}_{[A_1,\ldots,A_m]\ on\ [B_1,\ldots,B_n]}(\sigma_{A_u=x\wedge B_v=y}(V)) =$$
$$\sigma_{not\ all\ \perp}(\pi_{K,\{case(\text{“}a_1^{i_1}**\ldots a_u^{i_1}\ldots **a_m^{i_1}**B_1\text{”},\ldots,\text{“}a_1^{i_1}**\ldots a_u^{i_1}\ldots **a_m^{i_1}**B_n\text{”})\}})($$
$$GPIVOT^{[\{(a_1^i,\ldots,a_m^i)\}]}_{[A_1,\ldots,A_m]\ on\ [B_1,\ldots,B_n]}(V)) \quad (8)$$

Proof for Equation (8): *(1) First we prove that both sides of Equation (8) generate the same set of key values. The left side outputs the 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 *iff there exists at least one row in* V *that satisfies* $_{K=k\wedge A_u=x\wedge B_v=y}$.

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

(2) Next we prove that for each key value k*, both sides generate the same row. For any column* $\text{“}a_1^{i_1}**\ldots **a_m^{i_1}**B_j\text{”}$*, the left side of Equation (8) outputs* $\pi_{B_j}(\sigma_{K=k\wedge A_1=a_1^{i_1}\wedge\ldots\wedge A_u=a_u^{i_1}=x\ldots\wedge A_m=a_m^{i_1}\wedge B_v=y}(V))$. *The right side of Equation (8) outputs* $\pi_{B_j}((\sigma_{a_u^{i_1}=x\wedge B_v=y})(\sigma_{K=k\wedge A_1=a_1^{i_1}\wedge\ldots\wedge A_u=a_u^{i_1}\ldots\wedge A_m=a_m^{i_1}}(V)))$. *Hence, both sides output the same value for any column* $\text{“}a_1^{i_1}**\ldots **a_m^{i_1}**B_j\text{”}$.

By (1) and (2), we know that Equation (8) always holds. ∎

Equation (9) in Section 2.5.4:

$$_{K'}\mathcal{F}_{f(\{\text{“}a_1^i**\ldots**a_m^i**B_j\text{”}\})}(GPIVOT^{[\{(a_1^i,\ldots,a_m^i)\}]}_{[A_1\ldots A_m]\ on\ [B_1\ldots B_n]}(V)) =$$
$$GPIVOT^{[\{(a_1^i,\ldots,a_m^i)\}]}_{[A_1,\ldots,A_m]\ on\ [f(B_1),\ldots,f(B_n)]}(_{K',A_1,\ldots,A_m}\mathcal{F}_{f(B_1),\ldots,f(B_n)}(V)) \quad (9)$$

Proof for Equation (9): *We assume the output parameters for GPIVOT are* $\{(a_1^1, ..., a_m^1), ..., (a_1^p, ..., a_m^p)\}$.

(1) Since both sides of Equation (9) 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 (9) outputs the 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 (9) outputs the 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 K' *value* k'*, both sides of Equation (9) generate the same row. We now assume* $\{k_l\} = \pi_K(\sigma_{K'=k'\wedge((A_1,...,A_m)=\ (a_1^1,...,a_m^1)\vee\ ...\vee(A_1,...,A_m)\ =(a_1^p,...,a_m^p))}(V))$*,* $l = 1..q$*. Then for some* $i \leq m$ *and* $j \leq n$*, we let* $r_l = \pi_{B_j}(\sigma_{K=k_l\wedge(A_1,...,A_m)=(a_1^i,...,a_m^i)}(V)), l = 1..q$*. If there is no satisfied row in* V*, we let* $r_l = \perp$.

We first consider the left side of Equation (9). In particular, we consider the column $f(\text{“}a_1^i * *a_2^i ... * *a_m^i * *B_j\text{”})$ *for a given* k'*. The data to be aggregated are* $\{r_l\}, l = 1, .., q$*, i.e., the column outputs* $f(\{r_l\})$.

Now we consider the right side of Equation (9), which equals $\rtimes_{i=1}^{p}\pi_{K',f(B_1),...,f(B_n)}(\sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)}\ (_{K',A_1,...,A_m}\mathcal{F}_{\ f(B_1),...,f(B_n)}(V))) = \rtimes_{i=1}^{p}\pi_{K',f(B_1),...,f(B_n)}(_{K',A_1,...,A_m}\mathcal{F}_{\ f(B_1),...,f(B_n)}(\sigma_{(A_1,...,A_m)=(a_1^i,...,a_m^i)}V))$*. Hence, for a given value* k' *and* $(a_1^i, ..., a_m^i)$*, the inner GROUPBY for column* B_j *computes* $f(\pi_{B_j}(\sigma_{K'=k'\wedge(A_1,...,A_m)=(a_1^i,...,a_m^i)}(V)))$*. Here, we note that* $\sigma_{K'=k'\wedge\ (A_1,...,A_m)=\ (a_1^i,...,a_m^i)}(V)$ *must equal to* $\cup_{l=1}^{q}(\sigma_{K=k_l\wedge(A_1,...,A_m)=\ (a_1^i,...,a_m^i)}(V))$ *based on the definition* k_l.

Hence, the inner GROUPBY for column B_j actually computes:

$f(\pi_{B_j}(\cup_{l=1}^{q}(\sigma_{K=k_l \wedge (A_1,...,A_m)=(a_1^i,...,a_m^i)}(V)))$. *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.*

Since the aggregate function f disregards $\perp$ value and outputs $\perp$ when all the data in a group are $\perp$, then $f(\{r_l\}) = f(\sigma_{r_l \neq \perp}\{r_l\})$.

By (1) and (2), we thus establish the proof for Equation (9). ∎

Equation (10) in Section 2.5.5:

$$GUNPIVOT_{[\{\text{“}a_1^i **...a_m^i **B_j\text{”}\}]}(\\ GPIVOT^{[\{a_1^i,...,a_m^i\}]}_{[A_1...A_m] \text{ on } [B_1...B_n]}(V)) = (\sigma_s(V)) \qquad (10)$$

Proof for Equation (10): *First, obviously, GUNPIVOT outputs a table schema that equals to that of 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 $(a_1^i, ..., a_m^i)$, then both sides of Equation (10) will not include it. 2) if $(a_1, ..., a_m)$ does equal to some $a_1^i, ..., a_m^i$, then GPIVOT outputs $(k_1, ..., b_1, ..., b_n, ...)$, with the column name for each column b_j as "$a_1 * *.. * *a_m * *B_j$". The next GUNPIVOT outputs row $(k_1, a_1, ..., a_m, b_1, ..., b_n)$. Note that here we assume not all $(b_1, ..., b_n)$ are $\perp$. If not, the predicate σ_s should be extended to choose those rows whose $\{B_j\}$ not all $\perp$. Hence both sides of Equation (10) contain that row. By 1),2), Equation (10) always holds.* ∎

Equation (11) in Section 2.5.5:

$$GUNPIVOT_{[\{G_1,...,G_l\}]}(GPIVOT^{[\{(a_1^i,...,a_m^i)\}]}_{[A_1...A_m] \text{ on } [B_1...B_n]}(V)) = \\ GPIVOT^{[\{(a_1^i,...,a_m^i)\}]}_{[A_1...A_m] \text{ on } [B_1...B_n]}(GUNPIVOT_{[\{G_1,...,G_l\}]}(V)) \qquad (11)$$

Proof of Equation (11): *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 $GUNPIVOT_{[\{G_1,...,G_l\}]}$ on this row will output p rows $(k_1, h_1^1, ..., h_q^1, a_1, ..., a_m, b_1, ..., b_n)$,..., $(k_1, h_1^p, ..., h_q^p, a_1, ..., a_m, b_1, ..., b_n)$.*

Hence, for this row v, the right side of Equation (11) 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_j column's name as "$a_1 * *..a_m * *B_j$".

For the left side of Equation (11), GPIVOT will first output $(k_1, g_1, ..., g_l, ..., b_1, ..., b_n, ...)$, with each b_j column's name as "$a_1 * *..a_m * *B_j$". Then 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 (11) generates same output for each input row v. Hence Equation (11) always holds. ∎

Equation (12) in Section 2.5.5:

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

Proof for Equation (12): *First, obviously, GPIVOT 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 GUNPIVOT 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 in-*

put and outputs row $(k_1, c_1, ..., c_{pn})$. *Hence both sides of Equation (12) contain that row. By 1) and 2), Equation (12) always holds.* ∎

Equation (13) in Section 2.5.6:

$$\sigma_{A_u=x \wedge B_v=y}(GUNPIVOT_{[\{\text{“}a_1^i**...a_m^i**B_j\text{”}\}]}(H)) =$$
$$GUNPIVOT_{[\{\text{“}a_1^i**..**x..a_m^i**B_j\text{”}\}]}($$
$$\pi_{K,\{case(\text{“}a_1^i**..**x..a_m^i**B_1\text{”},...,\text{“}a_1^i**..**x..a_m^i**B_n\text{”})\}}(H)) \quad (13)$$

Proof for Equation (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* $a_u = x \wedge b_v = y$. *On the right side of Equation (13), the case expression actually removes the rows that do not satisfy* $a_u = X \wedge b_v = y$. *Hence they are equivalent.* ∎

Equation (14) in Section 2.5.6:

$$GUNPIVOT_{[\{\text{“}a_1^i**...a_m^i**B_j\text{”}\}]}(\ \sigma_{\text{“}a_1^{i_1}**...a_m^{i_1}**B_{l_1}\text{”}\ cp\ \text{“}a_1^{i_2}**...a_m^{i_2}**B_{l_2}\text{”}}(H))$$
$$= \pi_K(\sigma_{\text{“}a_1^{i_1}**...a_m^{i_1}**B_{l_1}\text{”}\ cp\ \text{“}a_1^{i_2}**...a_m^{i_2}**B_{l_2}\text{”}}(H)) \bowtie$$
$$GUNPIVOT_{[\{\text{“}a_1^i**...a_m^i**B_j\text{”}\}]}(H) \quad (14)$$

Proof of Equation (14): *The proof of this rule is straightforward. On the right side of Equation (14), since the join is on the key* K, *we can push it in as:*

$$GUNPIVOT_{[\{\text{“}a_1^i**...a_m^i**B_j\text{”}\}]}($$
$$\pi_K(\sigma_{\text{“}a_1^{i_1}**...a_m^{i_1}**B_{l_1}\text{”}\ cp\ \text{“}a_1^{i_2}**...a_m^{i_2}**B_{l_2}\text{”}}(H)) \bowtie (H))$$
$$= GUNPIVOT_{[\{\text{“}a_1^i**...a_m^i**B_j\text{”})\}]}($$

$\sigma_{\text{“}a_1^{i_1}**...a_m^{i_1}**B_{l_1}\text{”}\ cp\ \text{“}a_1^{i_2}**...a_m^{i_2}**B_{l_2}\text{”}}(H))$. ∎

Equation (15) in Section 2.5.8:

$$GUNPIVOT_{[\{\text{“}a_1^i**...a_m^i**B_j\text{”}\}]}(H) \bowtie_{B_l=X} (T) =$$
$$GUNPIVOT_{[\{\text{“}a_1^i**...a_m^i**B_j\text{”}\}]}(\pi_{K,\{case(\text{“}a_1^i**...a_m^i**B_1\text{”},...,}$$
$$\text{“}a_1^i**...a_m^i**B_n\text{”})\},X,Y(\ H \bowtie_{\text{“}a_1^1**...a_m^1**B_l\text{”}=X \vee ... \vee \text{“}a_1^p**...a_m^p**B_l\text{”}=X}\ T)) \quad (15)$$

Proof of Equation (15): *The proof of this rule is straightforward. The left side of Equation (15) outputs a row $r = (k, a_1, a_2, ..., a_m, b_1, ..., b_n, x, y)$ iff $b_l = x$. On the right side of Equation (15), the case expression actually removes the rows that do not satisfy $b_l = x$. Hence we conclude that they are equivalent.* ∎

Equation (16) in Section 2.5.8:

$$GUNPIVOT_{[\{\text{“}a_1^i**...a_m^i**B_j\text{”}\}]}(H \bowtie_{\text{“}a_1^{i_1}**...a_m^{i_1}**B_l\text{”}=X}\ T) =$$
$$\pi_K(H \bowtie_{\text{“}a_1^{i_1}**...a_m^{i_1}**B_l\text{”}=X}\ T) \bowtie$$
$$GUNPIVOT_{[\{\text{“}a_1^i**...a_m^i**B_j\text{”}\}]}(H) \qquad (16)$$

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, we can push it in as:*

$$GUNPIVOT_{[\{\text{“}a_1^i**...a_m^i**B_j\text{”}\}]}($$
$$\pi_K(H \bowtie_{\text{“}a_1^{i_1}**...a_m^{i_1}**B_l\text{”}=X}\ T) \bowtie (H)) =$$
$$GUNPIVOT_{[\{\text{“}a_1^i**...a_m^i**B_j\text{”}\}]}(H \bowtie_{\text{“}a_1^{i_1}**...a_m^{i_1}**B_l\text{”}=X}\ T).$$ ∎

Equation (17) in Section 2.5.9:

$${}_{K'}\mathcal{F}_{f(B_j)}(GUNPIVOT_{[\{\text{“}a_1^i**...a_m^i**B_j\text{”}\}]}(H)) =$$

$$_{K'}\mathcal{F}_{f(FB_j)}(GUNPIVOT_{[\{\text{“}a_1^i**...a_m^i**FB_j\text{”}\}]}($$
$$_{K''}\mathcal{F}_{\{\rho_{\text{“}a_1^i**...a_m^i**FB_j\text{”}}(f(\text{“}a_1^i**...a_m^i**B_j\text{”}))\}}(H)) \quad (17)$$

Proof of Equation (17): *Assume the group-by columns are $K' = (K'', A_{l_1}, ..., A_{l_q})$, with $K'' \subseteq K$ and $A_{l_r} \in \{A_1, ..., A_m\}$ for $r = 1, .., q$. On the right side of Equation (17), the inner group-by computes "$a_1^i * *...a_m^i * *FB_j$", which is equivalent to computing $f(B_j)$ for each group $(K'', A_1, ..., A_m)$ on the unpivoted data.*

Note that since $(K'', A_1, ..., A_m)$ is a superset of the next group-by columns, namely, $(K'', A_{l_1}, ..., A_{l_r})$, we can compute the group-by on $(K'', A_{l_1}, ..., A_{l_r})$ from the group-by result on $(K'', A_1, ..., A_m)$ using re-aggregation. This is known as a classic two-level aggregation strategy in [GBLP96]. ∎

Equation (18) in Section 2.5.9:

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

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 k_1 (we ignore other columns). The left side of Equation (18) first computes $(k_1, f(\{b_1^j\}), ..., f(\{b_n^j\}))$ and unpivots it 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) computes $\{(k_1, c_v^i, f(b_i^j))\}$. Thus it is identical to the left side. ∎

Bibliography

[AAe96] S. Agarwal, R. Agarwal, and P. M. Deshpande et.al. On the Computation of Multidimensional Aggregates. In *Proceedings of VLDB*, pages 506–521, 1996.

[AD98] S. Abiteboul and O. M. Duschka. Complexity of Answering Queries Using Materialized Views. In *Proceedings of PODS*, pages 254–263, 1998.

[AESY97] D. Agrawal, A. El Abbadi, A. Singh, and T. Yurek. Efficient View Maintenance at Data Warehouses. In *Proceedings of SIGMOD*, pages 417–427, 1997.

[ANS87a] ANSI/IEEE. *IEEE Standard for Radix Independent Floating Point Arithmetic, Std 754-1987 edition*. 1987.

[ANS87b] ANSI/IEEE. *IEEE Standard for Radix Independent Floating Point Arithmetic, Std 854-1987 edition*. 1987.

[ASX01] R. Agrawal, A. Somani, and Y. Xu. Storage and Querying of E-Commerce Data. In *Proceedings of VLDB*, pages 149–158, 2001.

[BDD+98] R. G. Bello, K. Dias, A. Downing, J. J. Feenan 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.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database System*. Addison-Wesley Pub., 1987.

[BLT86] J. A. Blakeley, P.-E. Larson, and F. W. Tompa. Efficiently Updating Materialized Views. *Proceedings of SIGMOD*, pages 61–71, 1986.

[CCH+98] L. S. Colby, R. L. Cole, E. Haslam, N. Jazayeri, G. Johnson, W. J. Mckenna, L. Schumacher, and D. Wilhite. Redbrick Vista: Aggregate Computation and Management. In *Proceedings of ICDE*, pages 174–177, 1998.

[CCPS03] S. Chen, R. Cochrane, H. Pirahesh, and R. Sidle. Incremental Automatic Summary Table Maintenance Using Work Areas. *Patent ARC20030030*, 2003.

[CCZ+04] S. Chen, J. Chen, X. Zhang, , and E. A. Rundensteiner. Detection and Correction of Conflicting Source Updates for View Maintenance. In *Proceedings of ICDE*, pages 436–448, 2004.

[CD97] S. Chaudhuri and U. Dayal. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Record*, 26(1):65–74, 1997.

[CGGL04] 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.

[CGL83] T. F. Chan, G. H. Golub, and R. J. Leveque. Algorithms for Computing the Sample Variance: Analysis and Recommendations. *The American Statistician*, 37(3), 1983.

[CGL+96] 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.

[Cha98] Donald D. Chamberlin. *A Complete Guide to DB2 Universal Database*. Morgan Kaufmann, 1998.

[CHS01] R. Chirkova, A. Y. Halevy, and D. Suciu. A Formal Perspective on the View Selection Problem. In *Proceedings of VLDB*, pages 59–68, 2001.

[CKP95] S. Chaudhuri, R. Krishnamurthy, and S. Potamianos. Optimizing Query with Materialized Views. In *Proceedings of ICDE*, pages 190–200, 1995.

[CLR04] S. Chen, B. Liu, and E. A. Rundensteiner. Multiversion-based View Maintenance over Distributed Data Sources. *ACM Transactions on Database Systems (TODS)*, 29(4):675–709, December 2004.

[CM77] A. K. Chandra and P. M. Merlin. Optimal Implementation of Conjunctive Queries in Relational Databases. In *Proceedings of STOC*, pages 77–90, 1977.

[CMR02] N. Colossi, W. Malloy, and B. Reinwald. Relational Extensions for OLAP. *IBM System Journal, Vol 41, No 4*, pages 714–731, 2002.

[CNS99] S. Cohen, W. Nutt, and A. Serebrenik. Rewriting aggregate queries using views. In *Proceedings of PODS*, pages 155–166, 1999.

[CNS03] S. Cohen, W. Nutt, and Y. Sagiv. Containment of Aggregate Queries. In *Proceedings of ICDT*, pages 111–125, 2003.

[CR05a] L. Chen and E. A. Rundensteiner. XQuery containment in presence of variable binding dependencies. In *Proceedings of WWW*, pages 288–297, 2005.

[CR05b] S. Chen and E. A. Rundensteiner. GPIVOT: Efficient Incremental Maintenance of Complex ROLAP Views. In *Proceedings of ICDE*, pages 552–563, 2005.

[CS94] S. Chaudhuri and K. Shim. Including Group-By in Query Optimization. In *Proceedings of VLDB*, pages 354–366, 1994.

[CSC+03] S. Chen, R. Sidle, L. S. Colby, R. Cochrane, and H. Pirahesh. Supporting Aggregate Functions with Workareas. Technical report, 2003.

[CW91] S. Ceri and J. Widom. Deriving Production Rules for Incremental View Maintenance. In *Proceedings of VLDB*, 1991. Also published in/as: IBM Almaden Res.Ctr., IBM Res.RP. RJ 8027, Mar.1991, 26pp.

[CZC+01] J. Chen, X. Zhang, S. Chen, A. Koeller, and E. A. Rundensteiner. DyDa: Data Warehouse Maintenance under Fully Concurrent Environments. In *Proceedings of SIGMOD, system demonstration*, page 619, 2001.

[DESR03] K. Dimitrova, M. El-Sayed, and E. A. Rundensteiner. Order-Sensitive View Maintenance of Materialized XQuery Views. In *International Conference on on Conceptual Modeling*, pages 144–157, 2003.

[EN89] R. Elmasri and S. B. Navathe. *Fundamental of Database Systems*. Benjamin/Cummings, 1989.

[GBLP96] 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.

[GHQ95] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-Query Processing in Data Warehousing Environments. In *Proceedings of VLDB*, pages 358–369, 1995.

[GK98] T. Griffin and B. Kumar. Algebraic Change Propagation for Semijoin and Outerjoin Queries. *SIGMOD Record*, 27(3):22–27, 1998.

[GL95] T. Griffin and L. Libkin. Incremental Maintenance of Views with Duplicates. In *Proceedings of SIGMOD*, pages 328–339, 1995.

[GL01] J. Goldstein and P. Larson. Optimizing Queries Using Materialized Views: A Practical, Scalable Solution. In *Proceedings of SIGMOD*, 2001.

[GLRG04] H. Guo, P. Larson, R. Ramakrishnan, and J. Goldstein. Relaxed Currency and Consistency: How to Say "Good Enough" in SQL. In *Proceedings of SIGMOD*, pages 815–826, 2004.

[GM99] H. Gupta and I. S. Mumick. Selection of Views to Materialize Under a Maintenance Cost Constraint. In *Proceedings of ICDT*, pages 453–470, 1999.

[GMR95] A. Gupta, I.S. Mumick, and K.A. Ross. Adapting Materialized Views after Redefinition. In *Proceedings of SIGMOD*, pages 211–222, 1995.

[GMRR01] A. Gupta, I. S. Mumick, J. Rao, and K. A. Ross. Adapting Materialized Views after Redefinitions: Techniques and a Performance Study. *Information Systems*, 2001.

[GMS93] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. In *Proceedings of SIGMOD*, pages 157–166, 1993.

[GRT99] S. Grumbach, M. Rafanelli, and L. Tininini. Querying Aggregate Data. In *Proceedings of PODS*, pages 174–184, 1999.

[GT00] S. Grumbach and L. Tininini. On the Content of Materialized Aggregated Views. In *Proceedings of PODS*, pages 174–184, 2000.

[Gup97] H. Gupta. Selection of Views to Materialize in a Data Warehouse. In *Proceedings of ICDT*, pages 98–112, 1997.

[HFLP89] L. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. Extensible Query Processing in Starburst. In *Proceedings of SIGMOD*, pages 377–388, 1989.

[HRU96] V. Harinarayan, A. Rajaraman, and J.D. Ullman. Implementing Data Cubes Efficiently. In *Proceedings of SIGMOD*, pages 205–216, 1996.

[JR03] A. Jagatheensan and A. Rajasekar. Data Grid Management Systems. In *Proceedings of SIGMOD*, page 683, 2003.

[Klu88] A. Klug. On Conjunctive Queries Containing Inequalities. *J. ACM*, pages 146–160, 1988.

[KR81] S. Koenig and R.Paige. A Transformational Framework for the Automatic Control of Derived Data. In *Proceedings of VLDB*, pages 306–318, 1981.

[KR02] A. Koeller and E. A. Rundensteiner. Incremental Maintenance of Schema-Restructuring Views. In *Proceedings of EDBT*, pages 354–371, 2002.

[KR04] A. Koeller and E. A. Rundensteiner. Incremental Maintenance of Schema-Restructuring Views in SchemaSQL. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 16:1096–1111, 2004.

[LGMT03] A. Y. Levy, Z. G.Ives, P. Mork, and I. Tatarinov. Piazza: Data management infrastructure for semantic web applications. In *Proceedings of WWW*, pages 556–567, 2003.

[LMS95] A.Y. Levy, A.O. Mendelzon, and Y. Sagiv. Answering Queries Using Views. In *Proceedings of PODS*, pages 95–104, May 1995.

[LMSS95] J. J. Lu, G. Moerkotte, J. Schue, and V. S. Subrahmanian. Efficient Maintenance of Materialized Mediated Views. In *Proceedings of SIGMOD*, pages 340–351, 1995.

[LNR02] A. M. Lee, A. Nica, and E. A. Rundensteiner. The EVE Approach: View Synchronization in Dynamic Distributed Environments. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, pages 931–954, 2002.

[LPT99] L. Liu, C. Pu, and W. Tang. Continual Queries for Internet Scale Event-Driven Information Delivery. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 11(4):610–628, 1999.

[LS93] A. Y. Levy and Y. Sagiv. Queries Independent of Updates. In *Proceedings of VLDB*, pages 171–181, 1993.

[LS99] T. Ling and E. Sze. Materialized View Maintenance Using Version Numbers. In *Proceedings of DASFAA*, pages 263–270, 1999.

[LSPC00] W. Lehner, R. Sidle, H. Pirahesh, and R. Cochrane. Maintenance of Automatic Summary Tables. In *Proceedings of SIGMOD*, pages 512–513, 2000.

[LSS96] L. V. S. Lakshmanan, F. Sadri, and I. N. Subramanian. SchemaSQL — A Language for Interoperability in Relational Multi-database Systems. In T. M. Vijayaraman et al., editors, *Proceedings of VLDB*, pages 239–250, 1996.

[LSS99] 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.

[MBR01] J. Madhaven, P. A. Bernstein, and E. Rahm. Generic Schema Matching with Cupid. In *International Conference on Very Large Data Bases*, pages 49–58, 2001.

[MHH00] R. J. Miller, L. M. Haas, and M. A. Hernández. Schema Mapping as Query Discovery. In *International Conference on Very Large Data Bases*, pages 77–88, 2000.

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

[Mil98] R. J. Miller. Using Schematically Heterogeneous Structures. In *Proceedings of SIGMOD*, pages 189–200, 1998.

[MQM97] 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.

[MS02] G. Miklau and D. Suciu. Containment and Equivalence for an XPath Fragment. In *Proceedings of PODS*, pages 65–76, 2002.

[MS05] B. Mandhani and D. Suciu. Query Caching and View Selection for XML Databases. In *Proceedings of VLDB*, 2005.

[NLR98] A. Nica, A. J. Lee, and E. A. Rundensteiner. The CVS Algorithm for View Synchronization in Evolvable Large-Scale Information Systems. In *Proceedings of International Conference on Extending Database Technology (EDBT'98)*, pages 359–373, Valencia, Spain, March 1998.

[NR99] A. Nica and E. A. Rundensteiner. View Maintenance after View Synchronization. In *International Database Engineering and Applications Symposium (IDEAS'99)*, pages 213–215, 1999.

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

[PL00] R. Pottinger and A. Y. Levy. A Scalable Algorithm for Answering Queries Using Views. In *Proceedings of VLDB*, pages 484–495, 2000.

[PSCP02] T. Palpanas, R. Sidle, R. Cochrane, and H. Piramesh. Incremental Maintenance for Non-Distributive Aggregate Functions. In *Proceedings of VLDB*, pages 802–813, 2002.

[PV99] Y. Papakonstantinou and V. Vassalos. Query Rewriting for Semistructured Data. In *Proceedings of SIGMOD*, pages 455–466, 1999.

[QGMW96] D. Quass, A. Gupta, I. S. Mumick, and J. Widom. Making Views Self-Maintainable for Data Warehousing. In *Conference on Parallel and Distributed Information Systems*, pages 158–169, 1996.

[Qua96] D. Quass. Maintenance Expressions for Views with Aggregation. In *Proceedings of the Workshop on Materialized Views: Techniques and Applications*, pages 110–118, 1996.

[QW91] X. Qian and G. Wiederhold. Incremental Recomputation of Active Relational Expressions. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 3(3):337–341, 1991.

[RPZ04] J. Rao, H. Pirahesh, and C. Zuzarte. Canonical Abstraction for Outerjoin Optimization. In *Proceedings of SIGMOD*, pages 671–682),, 2004.

[SBCL00] 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.

[SDJL96] D. Srivastava, S. Dar, H.V. Jagadish, and A.Y. Levy. Answering Queries with Aggregation Using Views. In *Proceedings of VLDB*, pages 318–329, 1996.

[SY81] Y. Sagiv and M. Yannakakis. Equivalence Among Relational Expressions with Union and Difference Operators. *J. ACM*, pages 633–655, 1981.

[Tar72] R. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM J. Computing*, 1(2), June 1972.

[TIHW01] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S Weld. Updating XML. In *Proceedings of SIGMOD*, pages 413–424, 2001.

[TPC95] TPC. Benchmark standard specification. May 1995.

[Ull89] J.D. Ullman. *Principle of Database and Knowledge-Base Systems*. Computer Science Press, 1989.

[VMP03] Y. Velegrakis, R. J. Miller, and L. Popa. Mapping Adaptation under Evolving Schemas. In *Proceedings of VLDB*, pages 584–595, 2003.

[W3C99] W3C. XML Path Language (XPath) Version 1.0. W3C Recommendation. http://www.w3.org/TR/xpath.html, November 1999.

[W3C05] W3C. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/, February 2005.

[WCL91] J. Widom, R. Cochrane, and B. G. Lindsay. Implementing Set-Oriented Production Rules as an Extension to Starburst. In *Proceedings of VLDB*, pages 275–285, 1991.

[Wid95] J. Widom. Research Problems in Data Warehousing. In *Proceedings of CIKM*, pages 25–30, 1995.

[WZ00] H. Wang and C. Zaniolo. Using SQL to Build New Aggregates and Extenders for Object- Relational Systems. In *Proceedings of VLDB*, pages 166–175, 2000.

[YL87] H. Z. Yang and P. Larson. Query Transformation for PSJ-Queries. In *Proceedings of VLDB*, pages 245–254, 1987.

[YP05] C. Yu and L. Popa. Semantic Adaptation of Schema Mapping after Schemas Evolve. In *Proceedings of VLDB*, 2005.

[YYY+03] K. Yi, H. Yu, J. Yang, G. Xia, and Y. Chen. Efficient Maintenance of Materialized Top-K Views. In *Proceedings of ICDE*, pages 189–200, 2003.

[ZCL+00] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, and M. Urata. Answering Complex Queries using Automatic Summary Tables. In *Proceedings of SIGMOD*, pages 105–116, 2000.

[ZGMHW95] Y. Zhuge, Héctor García-Molina, J. Hammer, and J. Widom. View Maintenance in a Warehousing Environment. In *Proceedings of SIGMOD*, pages 316–327, May 1995.

[ZGMW96] Y. Zhuge, Héctor García-Molina, and J. L. Wiener. The Strobe Algorithms for Multi-Source Warehouse Consistency. In *International Conference on Parallel and Distributed Information Systems*, pages 146–157, December 1996.

[ZPR02] X. Zhang, B. Pielech, and E. A. Rundensteiner. Honey, I shrunk the XQuery!: an XML Algebra Optimization Approach. In *Web Information and Data Management (WIDM)*, pages 15–22, 2002.

[ZR01] X. Zhang and E. A. Rundensteiner. Integrating the Maintenance and Synchronization of Data Warehouses Using a Cooperative Framework. In *Information Systems*, volume 27, pages 219–243, 2001.

[ZZL+04] D. Zilio, C. Zuzarte, S. Lightstone, R. Cochrane, and et.al. Recommending Materialized Views and Indexes with IBM DB2 Design Advisor. In *Proceedings of ICAC*, pages 180–188, 2004.