LINEAGE TRACING IN DATA WAREHOUSES

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Yingwei Cui
December 2001

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Professor Jennifer Widom
(Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Professor Hector Garcia-Molina

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Dr. Philip A. Bernstein

Approved for the University Committee on Graduate Studies:

Abstract

Data warehousing systems collect data from multiple distributed data sources and store integrated and summarized information in local databases for efficient data analysis and mining. Sometimes, when analyzing data at a warehouse, it is useful to "drill down" and investigate the source data from which certain warehouse data was derived. For a given warehouse data item, identifying the exact set of source data items that produced the warehouse data item is termed the *data lineage* problem.

This thesis presents our research results on tracing data lineage in a warehousing environment:

- Formal definitions of data lineage for data warehouses defined as relational materialized views over relational sources, and for warehouses defined using graphs of general data transformations.
- Algorithms for lineage tracing, again considering both relational and transformational warehouses, along with a suite of optimization techniques.
- Performance evaluations through simulations, and a lineage tracing prototype developed within the WHIPS (WareHousing Information Processing System) project at Stanford.
- Applying data lineage techniques to obtain improved algorithms for the well-known database view update problem.

Acknowledgements

First and foremost, I thank my mother and father for their love, support, and guidance.

I thank my husband, Hui, for being so supportive and for making my life colorful and balanced.

I thank my sister, Wen, who has always been my best friends and teacher.

I am especially grateful to my advisor, Jennifer Widom, for teaching me how to do research, how to write a paper, and how to balance work and family.

I would also like to thank the other members of the Stanford Database Group—especially Kevin Chang, Hector Garcia-Molina, Wilburt Labio, Chen Li, Jeff Ullman, Jun Yang, Yue Zhuge for helping me shape and refine my ideas.

I thank all the WHIPS developers for helping to build a platform on which my research is based: Reza Behforooz, Himanshu Gupta, Joachim Hammer, Wilburt Labio, Janet Wiener, Jun Yang, and Yue Zhuge.

I thank Yujie Cao for helping to implement our initial view update algorithms.

Finally, I thank my co-authors: Donald Cox, Wilburt Labio, Derek Lam, Hector Garcia-Molina, Janet Weiner, Jennifer Widom, and Jun Yang.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this thesis, we address the general problem of tracing data lineage in a warehousing environment. *Data warehousing* systems integrate information from multiple distributed *data sources* into a central repository called a *data warehouse* for efficient data analysis and mining. Sometimes it is useful for an analyst to look not only at the information in the warehouse, but also to investigate how certain warehouse information was derived from the sources. Tracing warehouse data items back to the source data items from which they were derived is termed the *data lineage* problem. Data lineage helps users understand warehouse data better, and can be crucial for in-depth data analysis and decision support.

1.1 Data Warehousing

The amount of relevant data available to enterprises is growing rapidly, and as it grows, using the data effectively becomes more important and more challenging. To make informed business decisions, analysts often need to pose complex ad-hoc queries over data spanning multiple distributed, heterogeneous sources. Answering such queries on demand can be difficult and inefficient: data from different sources must be extracted, integrated, and processed at query time, and the sources may be expensive to access or periodically unavailable.

One popular approach for enabling efficient data analysis and decision support over data spanning multiple sources is *data warehousing* [CD97, IK93, LW95, Wid95]. Instead of

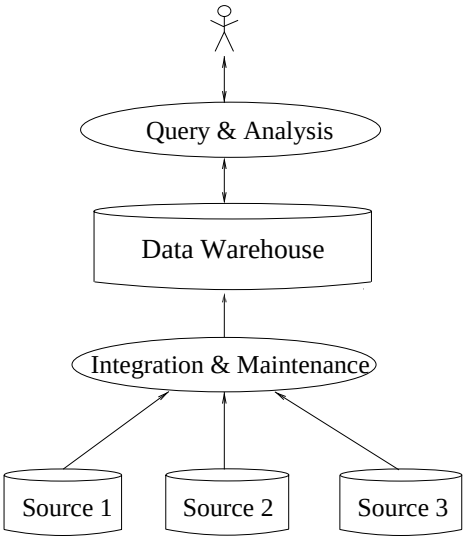


Figure 1.1: A basic data warehousing architecture

integrating data from sources on demand at query time, data warehousing systems collect data in advance, "cleanse" and integrate the data, and store the integrated information at a centralized data repository called a *data warehouse*. Once this initial *warehouse load* process is completed, users can perform analysis and mining directly over the warehouse data in an efficient and integrated manner. On the other hand, when the source data changes, it is necessary to perform *warehouse maintenance* to keep the warehouse data up-to-date.

Figure 1.1 shows the basic architecture of a data warehousing system with two major components: the Integration & Maintenance (IM) component, responsible for collecting and maintaining the warehouse data, and the Query & Analysis (QA) component, responsible for fulfilling the information needs of end users. Note that the two components are not independent. For example, what data the IM component should store depends on the expected needs of end users. In this thesis, we focus on extending the traditional capabilities of a QA component to include tracing data lineage, while taking both the QA and IM components into consideration for performance concerns over the source data before loading it into the warehouse.

We consider two warehousing scenarios: *relational* data warehouses and *transformational* data warehouses. In the relational scenario, the warehouse data is defined as relational *views* over relational sources specified using relational algebra. The warehouse

Program

program	unit_tuition
BS	$2K
MS	$2K
PhD	$2K
Dialin	$4K
Web	$6K
Audit	$1K

Student

student	program
Amy	MS
Bob	BS
Ed	Web
Jen	Web
Joe	BS
Ken	Web
Mike	MS
Sue	Dialin

Course

course	field
CS121	AI
CS144	Networking
CS145	Databases
CS154	Theory
CS221	AI
CS242	Theory
CS244	Networking
CS245	Databases

Enrollment

course	student	units
CS121	Joe	3
CS144	Bob	3
CS145	Bob	3
CS145	Jen	3
CS145	Ken	3
CS154	Joe	3
CS154	Mike	4
CS242	Amy	3
CS244	Sue	3
CS244	Amy	3
CS245	Amy	3
CS245	Ed	3
CS245	Ken	3

Figure 1.2: Source tables

views are *materialized* in the sense that the view contents derived from the source tables are physically stored at the warehouse. In the transformational scenario, the warehouse is constructed through graphs of general *data transformations*, which are arbitrary programs performing data cleansing, integration, and summarization tasks over the source data before loading it into the warehouse.

1.2 Data Lineage

Information stored at data warehouses is usually transformed, integrated, and summarized for easier detection of trends and patterns. For in-depth analysis, however, it is often useful to look not only at the information in the warehouse, but also to investigate how certain warehouse information was derived from the sources. Given a warehouse data item i, finding the exact set of source data items from which i was derived is termed the *data lineage* problem.

As an example, let us consider a university data warehouse constructed from four source tables containing information about academic programs, students, courses, and current course enrollments. Schemas and sample contents of the source tables are shown in Figure 1.2. From these sources, we can derive information about the tuition income of courses

IncomeByField

field	total_tuition
AI	$6K
Databases	$84K
Networking	$24K
Theory	$20K

Figure 1.3: Warehouse table

Program*

program	unit_tuition
BS	$2K
MS	$2K
Web	$6K

Student*

student	program
Amy	MS
Bob	BS
Ed	Web
Jen	Web
Ken	Web

Course*

course	field
CS145	Databases
CS245	Databases

Enrollment*

course	student	units
CS145	Bob	3
CS145	Jen	3
CS145	Ken	3
CS245	Amy	3
CS245	Ed	3
CS245	Ken	3

Figure 1.4: Lineage of $\langle$`Databases`, `$84K`$\rangle$

in each academic field, and store the result in a warehouse table `IncomeByField` (Figure 1.3). The warehouse table shows that courses in Databases made over three times as much money as courses in other fields. To figure out why database courses have achieved such financial success, the University Dean decides to trace the lineage of the warehouse data item $\langle$`Databases`, `$84K`$\rangle$. The lineage result is shown in Figure 1.4. It contains all database courses, their enrollments, and the associated student and payment information. We can then see from `Student`* that many of the students who take database courses are Web students. These students pay a much higher tuition rate than regular students, as seen in `Program`*. Here we see the benefit of tracing a specific warehouse data item back to the source data items from which it was derived.

Although from this example lineage tracing may appear easy, there are a number of significant challenges in semantics, algorithms, efficiency, and implementation, all tackled in this thesis. We outline the challenges and research contributions in Section 1.4, after discussing some further applications of data lineage in Section 1.3.

1.3 Lineage Applications

The primary general motivation for supporting data lineage is to enable further analysis of interesting or potentially erroneous warehouse data, as illustrated by the example in Section 1.2. Here we also discuss how some specific applications can benefit from lineage tracing techniques.

1.3.1 Data Analysis and Mining

Effective data analysis and mining requires facilities for data exploration at different levels [CD97]. The ability to select a portion of relevant warehouse data and "drill through" to its origins can be very useful for in-depth analysis and decision support, as shown in our earlier example. In addition, an analyst may want to check the origins of suspect or anomalous warehouse data, to verify the reliability of the sources or even repair erroneous source data. For example, a warehouse expense summary might show a suspiciously high figure for the purchase of 5 identical computers. Tracing the lineage of the summarized data might reveal from the source data that one of the computers costs 10 times as much as the others, an obvious data entry error (we hope).

1.3.2 Scientific Databases

In scientific environments, individual researchers share some commonly understood and accepted source data, manipulating it using different algorithms and ad-hoc experiments to derive new data, which is then added to the knowledge pool [HQGW93]. When examining other scientist's derived data, it is frequently useful to trace back to the commonly understood source data from which it originated.

1.3.3 Data Cleaning

Data extracted from sources often is "cleansed" by various transformations before being used to populate a data warehouse [GFS+01]. Data lineage helps locate the origins of cleansed data items, allowing the system to send reports about the cleansing process back to the sources, and debug transformations if anomalous output items are produced.

As a debugging example, suppose a librarian is writing a transformation to cleanse the data from a legacy bibliographic database. The transformation includes converting each author's first name into an initial (e.g., *Jennifer Widom* $\rightarrow$ *J. Widom*). When testing the transformation on an input data set, it produces:

```
[8] H. Garcia-Molina and J. D. and J. Widom,
    Database System Implementation, Prentice-Hall, 2000.
```

Tracing the lineage of this output data item produces:

```
[GUW00] Hector Garcia-Molina and Jeffrey D. Ullman and Jennifer Widom,
        Database System Implementation, Prentice-Hall, 2000.
```

By comparing the output data item and its lineage, it becomes obvious that the transformation is not properly handling names with middle initials. That is, instead of keeping both the middle initial and the last name, the transformation keeps only the middle initial.

1.3.4 Warehouse Load Resumption

Recovery from crashes during a data warehouse load or maintenance process (recall Section 1.1) can be extremely inefficient and expensive, since typically after a crash the warehouse is simply erased and reloaded from scratch. Data lineage enables the resumption of a warehouse load from where it was interrupted, without reloading or recomputing data that was successfully produced before the crash. Reference [LGMW00] introduces a type of data lineage and explains how it can be used to enable warehouse load resumption.

1.3.5 Materialized View Schema Evolution

When warehouse data is defined as materialized relational views (recall Section 1.1), it may become necessary to modify the view definitions (e.g., add a column) under certain circumstances [GMR95]. Data lineage can help "retrofit" existing view contents to the new view definition, without recomputing the entire view. For example, suppose in our `IncomeByField` warehouse view of Figure 1.3, we wish to add an additional column `students` indicating the number of students enrolled in classes for each field. Instead of recomputing the new view from scratch, it may be more efficient to compute values of the new column from the lineage of each tuple in the view. For example, from the lineage of the

original view tuple ⟨`Databases,$84K`⟩ shown in Figure 1.4, we can count the number of students in the `Enrollment` lineage table to obtain the value for column `students`.

1.3.6 View Update Problem

Tracing the origins of a given view data item is related to the well-known *view update problem* [BS81, DB78, Sto75]. In Chapter 4, we discuss this relationship in detail, and show how lineage tracing can be used to improve upon previous algorithms for certain aspects of view update.

1.4 Research Challenges and Thesis Contributions

We will consider two distinct data warehousing scenarios: warehouses defined by relational views and warehouses defined by general data transformations (Section 1.1). Under each scenario, we define data lineage formally, provide lineage tracing algorithms, and introduce optimization techniques for improved tracing performance. Based on our results for the relational view case, we have implemented a lineage tracing system prototype within the WHIPS [HGMW+95] data warehousing project at Stanford. We also have implemented and experimented with our algorithms for the transformational case, although not in the context of a significant data warehousing prototype.

1.4.1 Data Lineage for Relational Views

We first consider the relational scenario, where the warehouse data is defined as relational views over relational sources specified using relational algebra.

Lineage Definition and Tracing Algorithms

A relational view definition provides a mapping from the source data to the view data. Given a state of the source data, we can compute the corresponding view according to the view definition. However, determining the inverse mapping—from a view data item back to the source data that produced it—is not as straightforward. To determine the inverse

mapping accurately, we not only need the view definition, but we also need the source data and some additional information.

In Chapter 2, we formulate the data lineage problem in this environment by giving a declarative, inductive definition of data lineage for arbitrarily complex relational views defined using select, project, join, aggregation, union, intersection, and difference operators. We develop lineage tracing algorithms that use *tracing procedures* over source tables to retrieve data lineage for given view data items. The tracing procedures are generated automatically based on the view definition, and all queries in the procedures can be optimized easily by a standard database management system (DBMS). We first consider the problem under set semantics, then extend our results to handle bag (multiset) semantics.

Storing Auxiliary Views for Improved Tracing Performance

To compute the lineage of a view data item, our tracing procedures use the view definition and the source data, as well as possibly *auxiliary information* representing certain intermediate results in the view definition. In a distributed multi-source data warehousing environment, querying the sources during lineage tracing can be undesirable, difficult, or impossible: sources may be inaccessible, expensive to access, slow or expensive to transfer data from, and/or inconsistent with the views at the warehouse. By storing auxiliary information in the warehouse, we can reduce or entirely avoid source accesses during lineage tracing. There are numerous options for which auxiliary information to store, with significant performance tradeoffs. For example, storing a copy of all source data in the warehouse will improve lineage tracing by avoiding remote source queries altogether, but it also significantly increases warehouse storage cost and may introduce extra maintenance cost.

In Chapter 3, we start by introducing a family of schemes for storing *auxiliary views* to provide consistent and efficient lineage tracing for select-project-join (SPJ) views in a distributed warehousing environment. Our performance studies show that different schemes offer different advantages and thus are suitable for different settings, depending on system parameters such as network bandwidth and workload parameters such as query/update ratio. We then propose algorithms for selecting auxiliary views for arbitrary SPJ views with aggregation (ASPJ views). We suggest an initial search space of potentially beneficial

auxiliary views based on an *ASPJ view normal form*, and develop exhaustive and heuristic algorithms for exploring the search space and selecting a set of auxiliary views. We compare the optimality and running time of the algorithms using experiments based on the TPC-D benchmark [TPC96], as well as on a variety of synthetic views and statistics.

Applying Lineage to the View Update Problem

As mentioned briefly in Section 1.3.6, data lineage is closely related to the well-studied *view update problem*. The core of the view update problem is the ability to translate arbitrary updates on a view into updates on the relevant base (source) tables that the view is defined on. Many previous approaches, e.g., [Cle78, Kel86, LS91, RS79], can handle only limited types of views and updates, and/or they require participation from the view definer or the view updater in specifying view update translations.

In Chapter 4, we study the relationship between data lineage and the view update problem for deletions. We use techniques based on data lineage to devise a fully automatic algorithm for translating deletions against arbitrary SPJ views into deletions against the underlying database. Our algorithm finds a translation that is guaranteed to be *exact* (side-effect free) whenever an exact translation exists, using only the view definition at compile-time and the base data at view-update time.

1.4.2 Data Lineage for General Transformations

In deployed production data warehouses, instead of using relational views, data imported from the sources may be cleansed, integrated, and summarized through a sequence or graph of *transformations*. Many commercial warehousing systems provide tools for creating and managing such transformations as part of the *extract-transform-load* (*ETL*) process, e.g., [Inf, Mic, PPD, Sag]. The transformations may vary from simple algebraic operations or aggregations to complex procedural code. The lineage tracing problem for general data transformations is considerably more difficult and open-ended than the problem for relational views, because we no longer have the luxury of a fixed set of operators or the algebraic properties offered by relational views. Furthermore, since transformation graphs in real ETL processes can often be quite complex—containing as many as 60 or more

transformations—the storage requirements and runtime overhead associated with lineage tracing are very important considerations.

In Chapter 5, we formally define data lineage and provide lineage tracing algorithms for data warehouses specified through general transformations. Our algorithms take advantage of known structure or properties of transformations when present, but also work in the absence of such information, and they apply to single transformations, linear sequences of transformations, and arbitrary acyclic transformation graphs. We propose optimization techniques for improving tracing performance in the case of large transformation graphs. Our results can guide the design of data warehouses that enable effective lineage tracing.

1.4.3 Summary of Our Approach

Regardless of whether we are considering warehouses based on relational views or general transformations, our overall approach to lineage tracing can be divided into three major steps. At warehouse definition time, from the views or transformations we generate tracing procedures and determine what auxiliary information needs to be stored for lineage tracing. At warehouse load time, we compute the warehouse data as well as any auxiliary data determined in the first step. At lineage tracing time, tracing procedures are issued to the source and/or auxiliary data to compute the lineage result. All algorithms proposed in the thesis have been implemented, and experimental results are reported.

1.5 Related Work

In this section, we review previous work that is broadly related to the data lineage problem. Work related only to specific portions of the thesis is discussed in each chapter. Surveys on the general research topic of data warehousing appear in [CD97, LW95, Wid95].

Most previous work on data lineage considers *coarse-grained* (schema-level) lineage tracing based on *annotations* [BB99, HQGW93, LBM98]. Reference [LBM98] presents a solution for explicitly storing lineage (which they call *attribution*) of data items in query results based on a mediation architecture. The lineage information identifying which sources the warehouse data items were derived from, along with additional information such as

timestamps and source quality, is stored when queries are computed. In the warehousing context, [BB99] presents a scheme whereby the identifier of each warehouse data transformation is attached to all objects generated by the transformation, so that the user can trace which transformations produced each warehouse data object. This thesis focuses on tracing *fine-grained* (instance-level) lineage. We believe that using the annotation-based approaches for instance-level lineage tracing is likely to cause performance overhead at warehouse loading and maintenance time. Therefore, we use a "lazy" approach that calls tracing procedures only at lineage tracing time, in order to avoid such overhead. Our approach also imposes no extra requirements on warehouse view computations or data transformations, while annotation-based approaches often assume that these processes are capable of performing lineage annotations or can be modified to do so.

Reference [FJS97] uses a statistical approach to reconstruct base data from summary data and certain knowledge of constraints. This approach does not require access to the base data. However, it provides only estimated lineage information, and does not ensure accuracy. In contrast to all of the previous work discussed so far, we propose a solution that computes exact, fine-grained lineage using tracing procedures generated automatically from the warehouse definition.

In [WS97], a general framework is proposed for computing fine-grained data lineage in a transformational setting. The paper defines and traces data lineage for each transformation based on a *weak inverse*, which must be specified by the transformation definer. Lineage tracing through a transformation graph proceeds by tracing through each path one transformation at a time. In our approach for the transformational environment, the definition and tracing of data lineage is based on general transformation properties, and we specify an algorithm for combining transformations in a sequence or graph for improved tracing performance.

There has been growing interest in data lineage tracing in industry. Most On-Line Analytical Processing (OLAP) systems support a "drill-down" capability for multidimensional warehouse data, allowing aggregated data to be "unrolled" one level at a time within the warehouse [GBLP96]. A few products further support "drilling through" to the original transactional data, also stored in the warehouse [DB2, Pow]. Some on-line reporting systems allow report-to-report "drilling" based on annotations, e.g., by creating hyperlinks

from data in one report to lineage information in another report [Imp]. All of these products focus on specific types of views in specific settings, and do not enable lineage tracing through general relational views or transformations as we consider in this thesis.

1.6 Thesis Outline

The thesis proceeds as follows. Chapter 2 formulates the data lineage problem for relational views and provides lineage tracing algorithms. Chapter 3 introduces a variety of auxiliary view schemes to improve the performance of lineage tracing, and suggests algorithms for selecting auxiliary views for arbitrary SPJ views with aggregation, based on overall lineage tracing and view maintenance performance. In Chapter 4, we apply techniques based on data lineage to the view update problem, specifically to devise a fully automatic algorithm for translating view tuple deletions into source deletions. A complete treatment of lineage tracing in the presence of general data warehouse transformations is presented in Chapter 5. Chapter 6 concludes the thesis and discusses future work.

Chapter 2

Data Lineage for Relational Views

In this chapter, we formally define data lineage and develop lineage tracing algorithms for relational data warehouses. In particular we consider an environment where warehouse data is specified as relational materialized views over relational sources. Views are specified using relational algebra. We begin the chapter with running examples in Section 2.1 to motivate the data lineage problem. These examples are revisited throughout the chapter. We then present our relational view algebra in Section 2.2 and formally define data lineage in Section 2.3. Section 2.4 presents our lineage tracing algorithms for Select-Project-Join (SPJ) views, Section 2.5 extends the results to SPJ views with aggregation (ASPJ views), and Section 2.6 further handles views with set union and difference operators. In Section 2.7, we adapt our work to bag (multiset) semantics. In Section 2.8 we revisit the introductory example from Section 1.2 of Chapter 1. Finally, Section 2.9 discusses related work, and Section 2.10 concludes the chapter. The work presented in this chapter appeared originally in [CWW00].

2.1 Running Examples

Consider a data warehouse storing retail data derived from three source tables, whose schema and sample contents are shown in Figures 2.1–2.3. The `store` and `item` tables are self-explanatory. The `sales` table contains, for each store and item, the price of the item and number sold at that store.

s_id	s_name	city	state
001	Target	Palo Alto	CA
002	Target	Albany	NY
003	Macy's	San Francisco	CA
004	Macy's	New York City	NY
005	Safeway	Palo Alto	CA
006	Safeway	Denver	CO

Figure 2.1: `store` table

i_id	i_name	category
0001	binder	stationery
0002	pencil	stationery
0003	shirt	clothing
0004	pants	clothing
0005	pot	kitchenware
0006	plate	kitchenware

Figure 2.2: `item` table

s_id	i_id	price	num
001	0001	4	1000
001	0002	1	3000
001	0004	30	600
002	0001	5	800
002	0002	2	2000
002	0004	35	800
003	0003	45	1500
003	0004	60	600
004	0003	50	2100
004	0004	70	1200
004	0005	30	200

Figure 2.3: `sales` table

```
CREATE VIEW Calif AS
SELECT s_name, i_name, num
FROM   store, item, sales
WHERE  sales.s_id = store.s_id AND
       sales.i_id = item.i_id AND
       store.state = "CA"
```

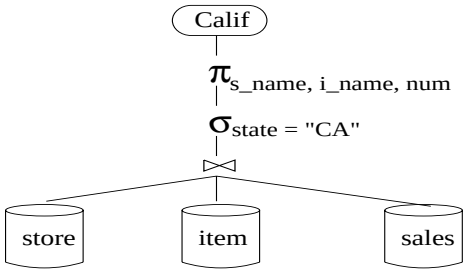


Figure 2.4: View definition for `Calif`

Example 2.1.1 (Lineage of SPJ View) Let us start with a simple example. Suppose an analyst wants to follow the selling patterns of California stores. A materialized view `Calif` can be defined in the data warehouse for this purpose. Figure 2.4 shows both SQL and relational algebra definitions of `Calif`. The materialized view for `Calif` over our sample data is shown in Figure 2.5.

The analyst browses the view table and is interested in the second tuple $\langle$`Target`, `pencil`, `3000`$\rangle$. He would like to see the relevant detailed information and asks question Q1: "Which base data produced tuple $\langle$`Target`, `pencil`, `3000`$\rangle$ in view `Calif`?" Using the algorithm presented in Section 2.4, we obtain the answer in Figure 2.6. The answer tells us that the Target store in Palo Alto sold 3000 pencils at a price of 1 dollar each. That

s_name	i_name	num
Target	binder	1000
Target	pencil	3000
Target	pants	600
Macy's	shirt	1500
Macy's	pants	600

Figure 2.5: Contents of `Calif` view

store

s_id	s_name	city	state
001	Target	Palo Alto	CA

item

i_id	i_name	category
0002	pencil	stationery

sales

s_id	i_id	price	num
001	0002	1	3000

Figure 2.6: `Calif` lineage for `<Target,pencil,3000>`

is, these three tuples join to form the $\langle$`Target`, `pencil`, `3000`$\rangle$ tuple. □

Example 2.1.2 (Lineage of Aggregation View) Now let us consider another warehouse view `Clothing`, for analyzing the total clothing sales of large stores (those that have sold more than 5000 clothing items). The SQL and relational algebra definitions of the view are shown in Figure 2.7. We extend traditional relational algebra with an aggregation operator, denoted $\alpha_{G,aggr(B)}$, where G is a list of grouping attributes, and $aggr(B)$ represents a list of aggregate functions over attributes. (Details are given in Section 2.2.) Assuming the data of Figures 2.1, 2.2, and 2.3, the materialized view contains one tuple, $\langle$`5400`$\rangle$.

The analyst may wish to learn more about the origins of this tuple, and asks question Q2: "Which base data produced tuple `<5400>` in view `Clothing`?" Not surprisingly, due to the more complex view definition, this question is more difficult to answer than Q1. We develop the appropriate algorithms in Section 2.5, and Figure 2.8 presents the answer. It lists all the branches of Macy's, the clothing items they sell (but not other items), and the sales information. All of this information is used to derive the tuple `<5400>` in `Clothing`. □

Questions such as Q1 and Q2 ask about the base tuples that derive a given view tuple. We call these base tuples the *derivation* (or *lineage*) of the view tuple. In Section 2.3, we formally define the concept of view tuple derivation. Sections 2.4–2.7 then present algorithms to compute tuple derivations for different view scenarios.

```
CREATE VIEW Clothing AS
SELECT   sum(num) as total
FROM     item, store, sales
WHERE    sales.s_id = store.s_id AND
         sales.i_id = item.i_id AND
         item.category = "clothing"
GROUP BY s_name
HAVING   total > 5000
```

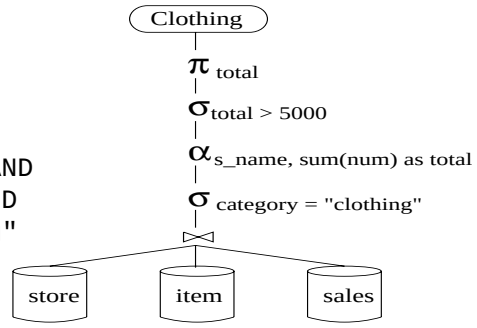


Figure 2.7: View definition for `Clothing`

store

s_id	s_name	city	state
003	Macy's	San Francisco	CA
004	Macy's	New York City	NY

item

i_id	i_name	category
0003	shirt	clothing
0004	pants	clothing

sales

s_id	i_id	price	num
003	0003	45	1500
003	0004	60	600
004	0003	50	2100
004	0004	70	1200

Figure 2.8: `Clothing` lineage for `<5400>`

2.2 Preliminaries

Throughout the thesis, we assume that a table (relation) R with schema $\mathbf{R}$ contains a set of tuples $\{t_1, \ldots, t_n\}$. A database D contains a list of *base* (*source*) *tables* $\langle R_1, \ldots, R_m \rangle$. A *view* V is a virtual or materialized result of a query over the base tables in D. The query (or the mapping from the base tables to the view table) is called the *view definition*, denoted as v. We say that $R_1, \ldots, R_m$ *derives* V if $V = v(R_1, \ldots, R_m)$. We consider *set semantics* (no duplicates) in Sections 2.3–2.6 of this chapter. We adapt our work to *bag semantics* (duplicates permitted) in Section 2.7.

Under set semantics, we consider a class of views defined over base relations using the relational algebra operators *selection* (σ), *projection* (π), *join* ($\bowtie$), *aggregation* (α), *set union* ($\cup$), and *set difference* ($-$). We use the standard relational semantics, included here for completeness. For an attribute list $A = A_1, \ldots, A_n$, we use the shorthand $t.A$ to denote $\langle t.A_1, \ldots, t.A_n \rangle$.

- Base case: $R = \{t \mid t \in R\}$

- *Selection*: $\sigma_C(V_1) = \{t \mid t \in V_1 \text{ and } t \text{ satisfies condition } C\}$

- *Projection*: $\pi_A(V_1) = \{t.A \mid t \in V_1\}$
  Note that we assume that π_A is duplicate-eliminating under set semantics.

- *Join*: $\bowtie_\theta(V_1, \ldots, V_m) = \{\langle t_1, \ldots, t_m\rangle \mid t_i \in V_i \text{ for } i = 1..m, \text{ and the } t_i\text{'s satisfy condition } \theta\}$
  In the remainder of the thesis, we will use the infix notation $\bowtie$ generically to indicate natural join ($\bowtie$), theta join ($\bowtie_\theta$), and cross-product ($\times$).

- *Aggregation*: $\alpha_{G,aggr(B)}(V_1) = \{\langle T.G,\ aggr(T.B)\rangle \mid T \subseteq V_1, \forall t,\ t' \in T, \forall t'' \notin T: t'.G = t.G\ \wedge\ t''.G \neq t.G\}$
  G is a grouping attribute list from V_1, and $aggr(B)$ represents a list of aggregation functions applied to attributes of V_1.[1] The aggregation operator groups the input table V_1 based on the grouping attribute G, such that each pair of tuples in each group have the same G value, and tuples across groups have different G values. For each group T, the answer contains the grouping value G and the aggregate function results.

- *Set Union*: $V_1 \cup \cdots \cup V_m = \{t \mid t \in V_i \text{ for some } i \in 1..m\}$

- *Set Difference*: $V_1 - V_2 = \{t \mid t \in V_1 \text{ and } t \notin V_2\}$

For convenience in formulation, when a view references the same relation more than once, we consider each relation instance as a separate relation. For example, we treat the self-join $R \bowtie R$ as (R **as** R_1) $\bowtie$ (R **as** R_2), and we consider R_1 and R_2 to be two tables in D. This approach allows view definitions to be expressed using an algebra tree instead of a graph, while not limiting the views we can handle.

Any view definition in our language can be expressed using a *query tree*, with base tables as the leaf nodes and operators as inner nodes. Figures 2.4 and 2.7 show examples of query trees. If a view definition contains only selection, projection, and join operators,

[1]For notational purposes, we always assume that G is nonempty. An empty G (indicating aggregation over all tuples) causes no problems in our framework since it only results in dropped projection attributes and dropped selection conditions. However, we do not handle the special-case semantics of SQL aggregation that produces a single tuple (with value 0, for example) in the case of aggregating over an empty set.

we call it an *SPJ* view. If the view contains aggregation operators, we call it an *ASPJ* view. Finally, view definitions that also contain the other relational operators specified above are called *general* views, which encompass all conventional relational views, excluding recursion or SQL-specific constructs.

2.3 View Tuple Derivations

In this section, we define the notion of a *tuple derivation*, which is the set of base relation tuples that produce a given view tuple. To define the concept of derivation, we assume logically that the view contents are computed by evaluating the view definition query tree bottom-up. Each operator in the tree logically generates an intermediate result table based on the results of its children nodes, and passes its result table upwards. We begin by focusing on individual relational operators, defining derivations of the operator's result tuples based on its input tuples. We then define tuple derivations for relational views inductively based on tuple derivations for relational operators.

2.3.1 Tuple Derivations for Operators

According to relational semantics, each operator can generate its result tuple-by-tuple based on its operand tables. Intuitively, given a tuple t in the result of operator *Op*, some subset of the input tuples produced t. We say that tuples in this subset *contribute to* t, and we call the entire subset the *derivation* of t. Input tuples not in t's derivation either contribute to nothing, or only contribute to result tuples other than t. Figure 2.9 illustrates the derivation of a result tuple. In the figure, operator *Op* is applied to tables T_1 and T_2, which may be base tables or temporary results from other operators. (In general, we use R's to denote base tables and T's to denote tables that may be base or derived.) Table T is the operation result. Given tuple t in T, only subsets T_1^* and T_2^* of T_1 and T_2 contribute to t. $\langle T_1^*, T_2^* \rangle$ is called t's derivation. The formal definition of tuple derivation for an operator is given next, followed by additional explanation.

Definition 2.3.1 (Tuple Derivation for an Operator) Let *Op* be any relational operator over tables $T_1, \ldots, T_m$, and let $T = Op(T_1, \ldots, T_m)$ be the table that results from applying

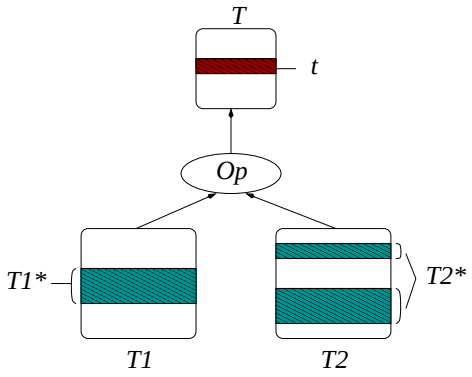


Figure 2.9: Derivation of tuple t

Op to $T_1, \ldots, T_m$. Given a tuple $t \in T$, we define *t's derivation in $T_1, \ldots, T_m$ according to Op* to be $Op^{-1}{}_{\langle T_1,\ldots,T_m\rangle}(t) = \langle T_1^*, \ldots, T_m^*\rangle$, where $T_1^*, \ldots, T_m^*$ are **maximal** subsets of $T_1, \ldots, T_m$ such that:

(a) $Op(T_1^*, \ldots, T_m^*) = \{t\}$

(b) $\forall T_i^*$: $\forall t^* \in T_i^*$: $Op(T_1^*, \ldots, \{t^*\}, \ldots, T_m^*) \neq \varnothing$

Also, we say that $Op^{-1}{}_{T_i}(t) = T_i^*$ is *t's derivation in T_i*, and each tuple t^* in T_i^* *contributes to t*, for $i = 1..m$. □

In Definition 2.3.1, requirement (a) says that the derivation tuple sets (the T_i^*'s) derive exactly t. From relational semantics, we know that for any result tuple t, there must exist such tuple sets. Requirement (b) says that each tuple in the derivation does in fact contribute something to t. For example, with requirement (b) and given $Op = \sigma_C$, base tuples that do not satisfy the selection condition C and therefore make no contribution to any result tuple will not appear in any result tuple's derivation. By defining the T_i^*'s to be the maximal subsets that satisfy requirements (a) and (b), we make sure that the derivation contains exactly all the tuples that contribute to t. Thus, the derivation fully explains why a tuple exists in the result.[2]

[2]By Definition 2.3.1, if $V = R - S$, then t's derivation not only includes t from R, but also includes all tuples $t' \neq t$ in S. We discuss this definition of derivation for set difference in more detail in Section 2.6.

Op^{-1} can be extended to represent the derivation of a set of tuples:

$$Op^{-1}{}_{\langle T_1,\ldots,T_m\rangle}(T) = \bigcup_{t\in T} Op^{-1}{}_{\langle T_1,\ldots,T_m\rangle}(t)$$

where $\bigcup$ represents the multi-way union of relation lists, i.e., $\langle R_1,\ldots,R_m\rangle \cup \langle S_1,\ldots,S_m\rangle = \langle (R_1 \cup S_1),\ldots,(R_m \cup S_m)\rangle$. Theorem 2.3.2 shows that there is a unique derivation for any operator and result tuple.

Theorem 2.3.2 (Derivation Uniqueness) Given $t \in Op(T_1,\ldots,T_m)$ where t is a tuple in the result of applying operator *Op* to tables $T_1,\ldots,T_m$, there exists a unique derivation of t in $T_1,\ldots,T_m$ according to *Op*. □

Proof: Recall that we are assuming set semantics at this point. Suppose that t has two derivations in $T_1,\ldots,T_m$: $D' = \langle T'_1,\ldots,T'_m\rangle$ and $D'' = \langle T''_1,\ldots,T''_m\rangle$. We prove that the two derivations must be the same.

If Op is α: $m = 1$. T'_1 is a maximal set that satisfies $Op(T'_1) = \{t\}$ and $\forall t' \in T'_1$: $Op(\{t'\}) \neq \varnothing$. From the semantics of the aggregation operator, we know that T'_1 contains exactly all tuples in T_1 that have the same grouping attributes as t. So does T''_1. Therefore $T'_1 = T''_1$.

For the remaining operators, σ, π, $\bowtie$, $\cup$, and $-$, we first prove that $D^* = \langle T^*_1,\ldots,T^*_m\rangle = D' \cup D''$ also satisfies requirements (a) $Op(T^*_1,\ldots,T^*_m) = \{t\}$ and (b) $\forall i = 1..m$: $\forall t^* \in T^*_i$: $Op(T^*_1,\ldots,\{t^*\},\ldots,T^*_m) \neq \varnothing$ in Definition 2.3.1.

If Op is σ, π, or $\cup$:

(a) Since D' and D'' satisfy (a), $Op(D^*) = Op(D' \cup D'') = Op(D') \cup Op(D'') = \{t\}$

(b) $\forall t^* \in T^*_i \in D^*$: $t^* \in T'_i$ or $t^* \in T''_i$. Suppose $t^* \in T'_i$. Since D' and D'' satisfy (b) $Op(T^*_1,\{t^*\},T^*_m) \supseteq Op(T'_1,\{t^*\},T'_m) \neq \varnothing$

If Op is $-$:

(a) $\langle T'_1, T'_2\rangle$ satisfies (a): $T'_1 - T'_2 = \{t\}$.

$\Rightarrow$ t is the only tuple that is in T'_1, but not in T'_2.
Also, $\langle T'_1, T'_2\rangle$ satisfies (b): $\forall t^* \in T'_1$: $\{t^*\} - T'_2 \neq \varnothing$, $t^* \notin T'_2$

$\Rightarrow$ $T'_1 = \{t\}$. Similarly, $T''_1 = \{t\}$

$\Rightarrow$ $T^*_1 - T^*_2 = \{t\}$

(b) From (a), we know $\forall t^* \in T_1^*$: $t^* = t \Rightarrow \{t^*\} - T_2^* = \{t\} \neq \varnothing$.
Also, $\forall t^* \in T_2^*$: $t^* \neq t \Rightarrow T_1^* - \{t^*\} = \{t\} \neq \varnothing$

If Op is $\bowtie$:

(a) $\langle T_1', \ldots, T_m' \rangle$ satisfies (a) and Op is monotone
$\Rightarrow$ $Op(T_1^*, \ldots, T_m^*) \supseteq Op(T_1', \ldots, T_m') = \{t\}$
Consider an arbitrary $T_i' \in D'$ and $t' \in T_i'$
$\Rightarrow$ Since $\langle T_1', \ldots, T_m' \rangle$ also satisfies (b), we know that:
$\varnothing \subsetneq Op(T_1', \ldots, \{t'\}, \ldots, T_m') \subseteq Op(T_1', \ldots, T_i', \ldots, T_m') = \{t\}$
$\Rightarrow$ $Op(T_1', \ldots, \{t'\}, \ldots, T_m') = \{t\}$
$\Rightarrow$ $t' = t.\mathbf{T_i}$
Similarly, we can prove $\forall T_i'' \in D''$: $\forall t'' \in T_i''$: $t'' = t.\mathbf{T_i}$
$\Rightarrow$ $\forall T_i^* \in D^*$: $\forall t^* \in T_i^*$: $t^* \in T_i'$ or $t^* \in T_i''$, and therefore $t^* = t.\mathbf{T_i}$
$\Rightarrow$ $Op(T_1^*, \ldots, T_m^*) \subseteq \{t\}$
$\Rightarrow$ $Op(T_1^*, \ldots, T_m^*) = \{t\}$

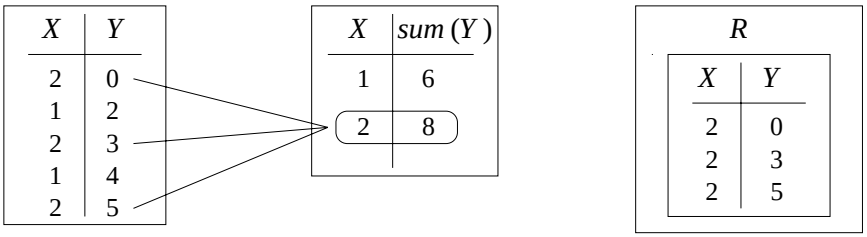
(b) Consider an arbitrary $T_i^* \in D^*$ and $t^* \in T_i^*$
$\Rightarrow$ $t^* \in T_i'$ or $t^* \in T_i''$. Suppose $t^* \in T_i'$
$\Rightarrow$ According to (b), $Op(T_1', \ldots, \{t^*\}, \ldots, T_m') \neq \varnothing$
$\Rightarrow$ According to monotonicity of Op,
$Op(T_1^*, \ldots, \{t^*\}, \ldots, T_m^*) \supseteq Op(T_1', \ldots, \{t^*\}, \ldots, T_m')$
$\Rightarrow$ $Op(T_1^*, \ldots, \{t^*\}, \ldots, T_m^*) \neq \varnothing$

Above, we have shown that $D^* = D' \cup D''$ also satisfies requirements (a) and (b) in Definition 2.3.1. According to the maximality of tuple derivations, D' is a maximal subset of D that satisfies the requirements. Thus, we know that $D' = D^*$. Similarly, we can prove $D'' = D^*$. Therefore $D' = D''$, and there is a unique derivation for every view tuple. □

Example 2.3.3 (Tuple Derivation for Aggregation) Given table R in Figure 2.10(a), and tuple $t = \langle 2, 8 \rangle \in \alpha_{X,sum(Y)}(R)$ in Figure 2.10(b), the derivation of t is

$$\alpha_{X,sum(Y)}{}^{-1}{}_{R}(\langle 2, 8 \rangle) = \{\langle 2, 0 \rangle, \langle 2, 3 \rangle, \langle 2, 5 \rangle\}$$

shown in Figure 2.10(c). Notice that R's subset $\{\langle 2, 3 \rangle, \langle 2, 5 \rangle\}$ also satisfies requirements



(a) R (b) $\alpha_{X,\,sum\,(Y\,)}(R)$ (c) $\alpha^{-1}_{R}(<2, 8>)$

Figure 2.10: Tuple derivation for aggregation

(a) and (b) in Definition 2.3.1, but it is not maximal. Intuitively, $\langle 2, 0\rangle$ also contributes to the result tuple, since $t = \langle 2, 8\rangle \in \alpha_{X,sum(Y)}(R)$ is computed by adding the Y attributes of $\langle 2, 3\rangle$, $\langle 2, 5\rangle$, and $\langle 2, 0\rangle$ in R. □

From Definition 2.3.1 and the semantics of the operators introduced in Section 2.2, we now specify the actual tuple derivations for each operator. We prove that these specific derivations indeed capture the general definition of tuple derivations from Definition 2.3.1. The derivations for selection, projection, join, aggregation, and set union are fairly easy to understand. For the set difference operator, given $t \in T_1 - T_2$, the tuple t from T_1 and all tuples in T_2 contribute to t. Recall that a tuple's derivation *fully* explains why the tuple appears in the operation result. In particular, a tuple t appears in $T_1 - T_2$ not only because $\exists t \in T_1$, but also because $\neg\exists t \in T_2$. Therefore, all tuples in T_2 contribute to $t \in T_1 - T_2$ in the sense that they ensure $\neg\exists t \in T_2$. Further discussion and examples can be found in Section 2.6.

Theorem 2.3.4 (Tuple Derivations for Operators) Let $T, T_1, \ldots, T_m$ be tables. Recall that $\mathbf{T_i}$ denotes the schema of T_i.

$$
\begin{aligned}
{\sigma_C}^{-1}{}_{\langle T\rangle}(t) &= \langle\{t\}\rangle, \text{ for } t \in \sigma_C(T)\\
{\pi_A}^{-1}{}_{\langle T\rangle}(t) &= \langle\sigma_{A=t}(T)\rangle, \text{ for } t \in \pi_A(T)\\
{\bowtie}^{-1}{}_{\langle T_1,\ldots,T_m\rangle}(t) &= \langle\{t.\mathbf{T_1}\},\ldots,\{t.\mathbf{T_m}\}\rangle, \text{ for } t \in T_1 \bowtie \cdots \bowtie T_m\\
{\alpha_{G,aggr(B)}}^{-1}{}_{\langle T\rangle}(t) &= \langle\sigma_{G=t.G}(T)\rangle, \text{ for } t \in \alpha_{G,aggr(B)}(T)\\
{\cup}^{-1}{}_{\langle T_1,\ldots,T_m\rangle}(t) &= \langle\sigma_{\mathbf{T_1}=t}(T_1),\ldots,\sigma_{\mathbf{T_m}=t}(T_m)\rangle, \text{ for } t \in T_1 \cup \cdots \cup T_m\\
{-}^{-1}{}_{\langle T_1,T_2\rangle}(t) &= \langle\{t\}, T_2\rangle, \text{ for } t \in T_1 - T_2
\end{aligned}
$$

Proof: We formally prove that the theorem follows our general lineage definition for relational operators in Definition 2.3.1. To prove ${Op^{-1}}_{\langle T_1,\ldots,T_m\rangle}(t) = \langle {T_1}', \ldots, {T_m}'\rangle$, we need to prove:

1. ${T_i}' \subseteq T_i$, $i = 1..m$.
2. $Op({T_1}', \ldots, {T_m}') = \{t\}$.
3. $\forall t' \in {T_i}' : Op({T_1}', \ldots, \{t'\}, \ldots, {T_m}') \neq \varnothing$.
4. $\forall \langle {T_1}'', \ldots, {T_m}''\rangle$ that satisfy 1, 2, 3: $\langle {T_1}'', \ldots, {T_m}''\rangle \subseteq \langle {T_1}', \ldots, {T_m}'\rangle$.

Here, we show the proofs for $\bowtie$, α, and $-$. Proofs for the other operators are very similar.

(1) Join: ${\bowtie^{-1}}_{\langle T_1,\ldots,T_m\rangle}(t) = \langle\{t.\mathbf{T_1}\}, \ldots, \{t.\mathbf{T_m}\}\rangle$, for $t \in T_1 \bowtie \cdots \bowtie T_m$.

Let ${T_i}' = \{t.\mathbf{T_i}\}$, $i = 1..m$.

1. $t \in T_1 \bowtie \cdots \bowtie T_m \Rightarrow t.\mathbf{T_i} \in T_i$, $i = 1..m \Rightarrow {T_i}' \subseteq T_i$, $i = 1..m$
2. $\{t.\mathbf{T_1}\} \bowtie \cdots \bowtie \{t.\mathbf{T_m}\} = \{t\}$ according to the definition of $\bowtie$
3. Consider an arbitrary t' in an arbitrary ${T_i}'$.
   $t' = t.\mathbf{T_i} \Rightarrow {T_1}' \bowtie \cdots \bowtie \{t'\} \bowtie \ldots {T_m}' = \{t\} \neq \varnothing$
4. Consider an arbitrary $\langle {T_1}'', \ldots, {T_m}''\rangle$ that satisfies 1, 2, 3.

$\Rightarrow$ ${T_1}'' \bowtie \cdots \bowtie {T_m}'' = \{t\}$

$\Rightarrow$ $\forall t'' \in {T_i}''$: ${T_1}' \bowtie \ldots \{t''\} \bowtie {T_m}' \subseteq \{t\}$ because of the monotonicity of $\bowtie$

Since $\forall t'' \in {T_i}''$: ${T_1}'' \bowtie \ldots \{t''\} \bowtie {T_m}'' \neq \varnothing$

$\Rightarrow$ $\forall t'' \in {T_i}''$: ${T_1}' \bowtie \ldots \{t''\} \bowtie {T_m}' = \{t\}$

$\Rightarrow$ $\forall t'' \in {T_i}''$: $t'' = t.\mathbf{T_i} \in {T_i}' \Rightarrow {T_i}'' \subseteq {T_i}'$

(2) Aggregation: ${\alpha_{G,aggr(B)}}^{-1}{}_{\langle T\rangle}(t) = \langle\sigma_{G=t.G}(T)\rangle$, for $t \in \alpha_{G,aggr(B)}(T)$.

Let $T' = \sigma_{G=t.G}(T)$.

1. $T' \subseteq T$ according to the definition of σ
2. $\forall t' \in T' : t'.G = t.G$ and $\forall t''$ such that $t'' \in T$ and $t'' \notin T' : t''.G \neq t.G$

$\Rightarrow$ $t = \langle T'.G, aggr(T'.B)\rangle$

(In other words, T' is the group from which t is computed.)

$\Rightarrow$ $\alpha_{G,aggr(B)}(T') = \{t\}$

3. Consider an arbitrary $t' \in T'$
   $t'.G = t.G \Rightarrow \alpha_{G,aggr(B)}(\{t'\}) = \{\langle t.G, aggr(\{t'.B\})\rangle\} \neq \varnothing$

4. Consider an arbitrary T'' that satisfies 1, 2, 3.
   $\alpha_{G,aggr(B)}(T'') = \{t\}$
   $\Rightarrow$ $\forall t'' \in T''$: $t''.G = t.G$
   $\Rightarrow$ $\forall t'' \in T''$: $t'' \in T' \Rightarrow T'' \subseteq T'$

(3) Difference: $-^{-1}{}_{\langle T_1,T_2\rangle}(t) = \langle\{t\}, T_2\rangle$, for $t \in T_1 - T_2$

Let $T_1' = \{t\}$, $T_2' = T_2$.

1. It is obvious that $T_i' \subseteq T_i$.
2. $t \in T_1 - T_2 \Rightarrow t \notin T_2 \Rightarrow T_1' - T_2' = \{t\} - T_2 = \{t\}$
   Thus, $\langle T_1', T_2'\rangle$ satisfies condition (a)
3. $\forall t' \in T_1', t' = t \Rightarrow \{t'\} - T_2' = \{t\} \neq \varnothing$
   $\forall t' \in T_2', t' \neq t \Rightarrow T_1' - \{t'\} = \{t\} \neq \varnothing$
   Thus, $\langle T_1', T_2'\rangle$ satisfies condition (b)
4. Consider an arbitrary $\langle T_1'', T_2''\rangle$ that satisfies 1, 2, 3.
   $\Rightarrow$ $T_1'' - T_2'' = \{t\}$ and $\forall t'' \in T_1''$: $\{t''\} - T_2'' \neq \varnothing$
   $\Rightarrow$ $\forall t'' \in T_1''$: $t'' = t$, because otherwise
   either $\{t''\} \subseteq T_1'' - T_2'' \neq \{t\}$ given $t'' \notin T_2''$, or $\{t''\} - T_2'' = \varnothing$ given $t'' \in T_2''$
   $\Rightarrow$ $T_1'' \subseteq T_1'$
   Also, $T_2'' \subseteq T_2' = T_2$

. □

2.3.2 Tuple Derivations for Views

As mentioned earlier, a view definition can be thought of as an operator tree that is evaluated bottom-up. Thus, in this section we proceed to define tuple derivations for views inductively based on tuple derivations for the operators comprising the view definition tree. Intuitively, if a base tuple t^* contributes to a tuple t' in the intermediate result of a view evaluation, and t' further contributes to a view tuple t, then t^* contributes to t. We define a view tuple's derivation to be the set of all base tuples that contribute to the view tuple. The specific process through which the view tuple is derived can be illustrated by applying the view definition tree to the derivation tuple sets, and presenting the intermediate results for each operator in the evaluation.

Definition 2.3.5 (Tuple Derivation for a View) Let D be a database with base tables $R_1, \ldots, R_m$, and let $V = v(D)$ be a view over D. Consider a tuple $t \in V$.

1. $v = R_i$, $i = 1..m$: Tuple $t \in R_i$ *contributes to itself* in V.

2. $v = Op(v_1, \ldots, v_k)$, where v_j is a view definition over D, $j = 1..k$: Suppose $t' \in v_j(D)$ contributes to t according to the operator *Op* (by Definition 2.3.1), and $t^* \in R_i$ contributes to t' according to the view v_j (by this definition recursively). Then t^* *contributes to* t *according to* v.

We define t*'s derivation in* D *according to* v to be ${v^{-1}}_D(t) = \langle R_1^*, \ldots, R_m^* \rangle$, where $R_1^*, \ldots, R_m^*$ are subsets of $R_1, \ldots, R_m$ such that $t^* \in R_i^*$ iff $t^* \in R_i$ contributes to t according to v, for $i = 1..m$. Also, we call R_i^* t*'s derivation in* R_i *according to* v, denoted as ${v^{-1}}_{R_i}(t)$. □

The derivation of a view tuple set T contains all base tuples that contribute to any view tuple in the set T:

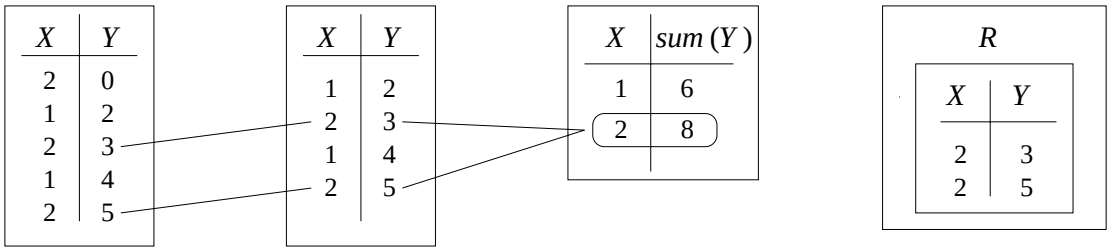
$${v^{-1}}_D(T) = \bigcup_{t \in T} {v^{-1}}_D(t)$$

Theorem 2.3.2 can be applied inductively in the obvious way to show that view tuple derivations are always unique.

Example 2.3.6 (Tuple Derivation for a View) Given base table R in Figure 2.11(a), view $V = \alpha_{X,sum(Y)}(\sigma_{Y \neq 0}(R))$ in Figure 2.11(c), and tuple $t = \langle 2, 8 \rangle \in V$, it is easy to see that tuples $\langle 2, 3 \rangle$ and $\langle 2, 5 \rangle$ in R contribute to $\langle 2, 3 \rangle$ and $\langle 2, 5 \rangle$ in $\sigma_{Y \neq 0}(R)$ in Figure 2.11(b), and further contribute to $\langle 2, 8 \rangle$ in V. The derivation of t is ${v^{-1}}_R(\langle 2, 8 \rangle) = \{\langle 2, 3 \rangle, \langle 2, 5 \rangle\}$ as shown in Figure 2.11(d). □

We now state some properties of view tuple derivations to provide the groundwork for our derivation tracing algorithms.

Lemma 2.3.7 (Contribution Transitivity) Let D be a database with base tables $R_1, \ldots, R_m$. Given view $V = v(D) = v'(V_1, \ldots, V_k)$, where $V_j = v_j(D)$ for $j = 1..k$, $\forall t \in V$: $\forall t^* \in R_i \in D$: t^* contributes to t according to v iff $\exists j \in 1..k$, $t' \in V_j$ such that t^* contributes to t' according to v_j and t' contributes to t according to v'. □



(a) R (b) $\sigma_{Y \neq 0}(R)$ (c) $V = \alpha_{X,\, sum\,(Y)}(\sigma_{Y \neq 0}(R))$ (d) $v^{-1}_{R}(<2, 8>)$

Figure 2.11: Tuple derivation for a view

Proof: We use induction on the height of the query tree for v', denoted $h(v')$, where the v_i's are the leaves.

Base case: For $h(v') = 0$, $v = v_1$. The Lemma holds trivially.
Induction hypothesis: Suppose for $0 \leq h(v') \leq n-1$, the Lemma also holds.
Induction step: For $h(v') = n$, $v' = Op(v_1', \ldots, v_q')$, $V_p' = v_p'(V_1, \ldots, V_k)$, and $h(v_p') \leq n-1$, $p = 1..q$.

1. If t^* contributes to t
$\Rightarrow$ According to Definition 2.3.5, $\exists p \in 1..q, \exists t'' \in V_p'$ such that t^* contributes to t'' and t'' contributes to t
$\Rightarrow$ Since $h(v_p') \leq n-1$, according to the induction hypothesis, $\exists j \in 1..k, t' \in V_j$ such that t^* contributes to t' and t' contributes to t''
$\Rightarrow$ According to Definition 2.3.5, t' contributes to t
$\Rightarrow$ $\exists j \in 1..k, t' \in V_j$, t^* contributes to t' and t' contributes to t.
2. If $\exists j \in 1..k, t' \in V_j$: t^* contributes to t' and t' contributes to t
$\Rightarrow$ According to Definition 2.3.5, $\exists p \in 1..q, \exists t'' \in V_p'$ such that t' contributes to t'' and t'' contributes to t
$\Rightarrow$ Since $h(v_p') \leq n-1$, according to the induction hypothesis, t^* contributes to t''
$\Rightarrow$ According to Definition 2.3.1, t^* contributes to t.

By induction, the Lemma holds for v' of height n, and therefore for any v' and v. □

Theorem 2.3.8 (Derivation Transitivity) Let D be a database with base tables $R_1, \ldots, R_m$,

and let $V = v(D)$ be a view over D. Suppose that v can also be represented as $V = v'(V_1, \ldots, V_k)$, where $V_j = v_j(D)$ is an intermediate view over D, for $j = 1..k$. Given tuple $t \in V$, let $V_j{}^*$ be t's derivation in V_j according to v'. Then t's derivation in D according to v is the concatenation of all $V_j{}^*$s' derivations in D according to v_j, $j = 1..k$:

$$v^{-1}{}_D(t) = \bigodot_{j=1..k} v_j^{-1}{}_D(V_j{}^*)$$

where $\bigodot$ represents the multi-way concatenation of relation lists.[3] □

Proof: We prove Theorem 2.3.8 using Lemma 2.3.7. We need to prove that $\forall i = 1..m$: all tuples in the R_i component[4] of $v^{-1}{}_D(t)$ are also in the R_i component of $\bigodot_{j=1..k} v_j{}^{-1}{}_D(V_j^*)$, and vice-versa.

1. Consider an arbitrary R_i and $t^* \in v^{-1}{}_{R_i}(t)$
⇒ According to Definition 2.3.5, t^* contributes to t
⇒ According to Lemma 2.3.7, $\exists j \in 1..k, t' \in V_j$ such that
t' contributes to t according to v', and t^* contributes to t' according to v_j
⇒ $t' \in V_j^*$ and $t^* \in v_j{}^{-1}{}_{R_i}(t')$
⇒ $t^* \in v_j{}^{-1}{}_{R_i}(V_j^*)$, where $v_j{}^{-1}{}_{R_i}(V_j^*)$ is the R_i component of $\bigodot_{j=1..k} v_j{}^{-1}{}_D(V_j^*)$

2. Consider an arbitrary R_i and t^* in the R_i component of $\bigodot_{j=1..k} v_j{}^{-1}{}_D(V_j^*)$
⇒ $\exists j \in 1..k$, such that v_j is defined over R_i
⇒ $t^* \in v_j{}^{-1}{}_{R_i}(V_j^*)$
⇒ $\exists t' \in V_j^*$, such that $t^* \in v_j{}^{-1}{}_{R_i}(t')$
⇒ t' contributes to t according to v', and t^* contributes to t' according to v_j
⇒ According to Lemma 2.3.7, t^* contributes to t
. ⇒ According to Definition 2.3.5, $t^* \in v^{-1}{}_{R_i}(t)$ □

[3]The concatenation of two relation lists $\langle R_1, \ldots, R_m \rangle \circ \langle S_1, \ldots, S_n \rangle$ is $\langle R_1, \ldots, R_m, S_1, \ldots, S_n \rangle$. Recall that relations are renamed so that the same relation never appears twice.

[4]The derivation of a tuple consists of a list of subsets of base tables. The R_i component of the derivation is the subset of base table R_i in the list. Here specifically, the R_i component of $v^{-1}{}_D(t)$ is $v^{-1}{}_{R_i}(t)$.

Theorem 2.3.8 follows from Definition 2.3.5. It shows that given a view V with a complex definition tree, we can compute a tuple's derivation by recursively tracing through the hierarchy of intermediate views that comprise the tree.

Since we define tuple derivations inductively based on the view query tree, an interesting question arises: Are the derivations of tuples in two equivalent views also equivalent? Two view definitions (or query trees) v_1 and v_2 are equivalent iff $\forall D$: $v_1(D) = v_2(D)$ [Ull89b]. We prove in Theorem 2.3.11 that given any two equivalent Select-Project-Join (SPJ) views, their tuple derivations also are equivalent. In order to prove Theorem 2.3.11, we first give a definition of tuple derivations specific to SPJ views and prove that it is equivalent to Definition 2.3.5 for SPJ views.

Definition 2.3.9 (Tuple Derivation for an SPJ View) Let D be a database with base tables $R_1, \ldots, R_m$, and let $V = v(D)$ be an SPJ view over D. Given tuple $t \in V$, $\forall R_i$, $t^* \in R_i$ *contributes* to t iff $t \in v(R_1, \ldots, R_{i-1}, \{t^*\}, R_{i+1}, \ldots, R_m)$. *$t$'s derivation according to* v is ${v^{-1}}_D(t) = \langle R_1^*, \ldots, R_m^* \rangle$, where each R_i^* contains all the tuples in R_i that contribute to t. □

Definition 2.3.9 says that a base tuple $t^* \in R_i$ contributes to t in the view if it produces t together with other base tables $R_1, \ldots, R_{i-1}, R_{i+1}, \ldots, R_n$. For SPJ views, this definition is equivalent to Definition 2.3.5, as shown in Lemma 2.3.10. However, it does not apply to views with aggregation.

Lemma 2.3.10 (Definition Equivalence) Definition 2.3.9 is equivalent to Definition 2.3.5 for SPJ views. In other words, the derivation of t defined by Definition 2.3.5 is the same as that defined by Definition 2.3.9 for every SPJ view and every traced tuple. □

Proof: We first prove the equivalence of the two definitions for views with a single SPJ operator. For $V = \sigma_C(R)$ or $\pi_A(R)$, $\forall t \in V$, let $\langle R^* \rangle$ and $\langle R^{**} \rangle$ be t's derivations defined by Definition 2.3.5 and Definition 2.3.9.

1. Consider an arbitrary $t^* \in R^*$

$\Rightarrow$ According to Definition 2.3.5 and the monotonicity of V,
$\varnothing \subsetneq v(\{t^*\}) \subseteq v(R^*) = \{t\} \Rightarrow v(\{t^*\}) = \{t\}$

$\Rightarrow$ t^* contributes to t according to Definition 2.3.9

$\Rightarrow$ $t^* \in R^{**}$

2. Consider an arbitrary $t^* \in R^{**}$

$\Rightarrow$ According to Definition 2.3.9, $v(\{t^*\}) = \{t\}$

$\Rightarrow$ $v(R^* \cup \{t^*\}) = v(R^*) \cup v(\{t^*\}) = \{t\}$ and $\forall t_1 \in R^* \cup \{t^*\} :\ v(\{t_1\}) \neq \varnothing$

$\Rightarrow$ $R^* \cup \{t^*\}$ also satisfies (a) and (b) in Definition 2.3.5

$\Rightarrow$ From the maximality of R^*, we know that $t^* \in R^*$

Therefore, $R^* = R^{**}$.

For $V = R \bowtie S$, $\forall t \in V$, let $\langle R^*, S^* \rangle$ and $\langle R^{**}, S^{**} \rangle$ be t's derivations defined by Definition 2.3.5 and Definition 2.3.9.

1. Consider an arbitrary $t^* \in R^*$

$\Rightarrow$ According to Definition 2.3.5 and the monotonicity of V,
$\varnothing \subsetneq \{t^*\} \bowtie S^* \subseteq R^* \bowtie S^* = \{t\} \Rightarrow \{t^*\} \bowtie S^* = \{t\}$

$\Rightarrow$ t^* contributes to t according to Definition 2.3.9

$\Rightarrow$ $t^* \in R^{**}$

2. Consider an arbitrary $t^* \in R^{**}$

$\Rightarrow$ According to Definition 2.3.9, $\{t^*\} \bowtie S = \{t\}$

$\Rightarrow$ $\exists t' \in S$ such that $\{t^*\} \bowtie \{t'\} = \{t\}$

$\Rightarrow$ $(R^* \cup \{t^*\}) \bowtie (S^* \cup \{t'\}) = \{t\}$. Also,
$\forall t_1 \in R^* \cup \{t^*\} :\ \{t_1\} \bowtie (S^* \cup \{t'\}) \neq \varnothing$ and
$\forall t_2 \in S^* \cup \{t'\} :\ (R^* \cup \{t^*\}) \bowtie \{t_2\} \neq \varnothing$

$\Rightarrow$ $\langle R^* \cup \{t^*\}, S^* \cup \{t'\} \rangle$ also satisfies (a) and (b) in Definition 2.3.5.

$\Rightarrow$ From the maximality of $\langle R^*, S^* \rangle$, we know that $t^* \in R^*$.

Therefore, $R^* = R^{**}$. Similarly, we can prove $S^* = S^{**}$.

We already know from Theorem 2.3.8 that tuple derivation as defined by Definition 2.3.5 is transitive. We can also prove this property for derivation as defined by Definition 2.3.9. After proving that Definition 2.3.5 and Definition 2.3.9 are equivalent for views with a single SPJ operator, due to the transitive property of tuple derivation, we can easily prove that Definition 2.3.5 and Definition 2.3.9 are equivalent for any SPJ view by induction. □

Theorem 2.3.11 (Derivation Equivalence for SPJ Views) Tuple derivations of equivalent SPJ views are equivalent. That is, given two equivalent SPJ views v_1 and v_2, $\forall D$: $\forall t \in v_1(D) = v_2(D)$: ${v_1^{-1}}_D(t) = {v_2^{-1}}_D(t)$. □

Proof: Having proved that Definition 2.3.9 is equivalent to Definition 2.3.5 for SPJ views, we now prove Theorem 2.3.11 based on Definition 2.3.9.

Given two equivalent SPJ views v_1 and v_2, we know that $\forall D = \langle T_1, \ldots, T_m \rangle$: $v_1(D) = v_2(D)$. Given $t \in v_1(D) = v_2(D)$, $\forall t^* \in {v_1^{-1}}_D(t)$: According to Definition 2.3.9, $v_1(R_1, \ldots, \{t^*\}, R_m) = \{t\}$. Since $v_1 \equiv v_2$, we know that $v_2(R_1, \ldots, \{t^*\}, R_m) = \{t\}$. Thus, $t^* \in {v_2^{-1}}_D(t)$. Therefore, ${v_1^{-1}}_D(t) \subseteq {v_2^{-1}}_D(t)$. We can similarly prove that ${v_2^{-1}}_D(t) \subseteq {v_1^{-1}}_D(t)$. Therefore, ${v_1^{-1}}_D(t) = {v_2^{-1}}_D(t)$. □

According to Theorem 2.3.11, we can transform any SPJ view to a simple canonical form before tracing tuple derivations, and we will exploit this property in our algorithms. Unfortunately, views with aggregation do not have this nice property, as shown in the following example.

Example 2.3.12 (Derivation Inequivalence for Views with Aggregation) Let $V_1 = v_1(R) = \alpha_{X,sum(Y)}(R))$ and $V_2 = v_2(R) = \alpha_{X,sum(Y)}(\sigma_{Y \neq 0}(R))$. v_1 and v_2 are equivalent, since $\forall R, v_1(R) = v_2(R)$. Given base table R in Figures 2.10(a) and 2.11(a), Figures 2.10(b) and 2.11(c) show that the contents of the two views are the same. However, the derivation of tuple $t = \langle 2, 8 \rangle \in V_1$ according to v_1 (shown in Figure 2.10(c)) is different from that according to v_2 (shown in Figure 2.11(d)). □

Given Definition 2.3.5, a straightforward way to compute a view tuple's derivation is to compute the intermediate results for all operators in the view definition tree, store the results as temporary tables, then trace the tuple's derivation in the temporary tables recursively until reaching the base tables. Obviously, this approach is impractical due to the computation and storage required for all the intermediate results. In the following sections, we separately consider SPJ views, ASPJ views, and more general views with set operators. We will show that one relational query over the base tables suffices to compute tuple derivations for simple views such as SPJ views. Recursive derivation tracing algorithms that require a modest amount of auxiliary information are needed for ASPJ and more general view derivation tracing.

2.4 Derivation Tracing for SPJ Views

Derivations of tuples in SPJ views can be computed using a single relational query over the base data. In this section, we specify *derivation tracing queries* for SPJ views, and briefly discuss some optimization issues for these queries.

2.4.1 Derivation Tracing Queries

Sometimes, we can write a query for a specific view definition v and view tuple t, such that if we apply the query to the database D it returns t's derivation in D (based on Definition 2.3.5). We call such a query a *derivation tracing query (or tracing query) for t and v*. More formally, we have:

Definition 2.4.1 (Derivation Tracing Query) Let D be a database, and let v be a view over D. Given tuple $t \in v(D)$, $TQ_{t,v}$ is a *derivation tracing query for t and v* iff $TQ_{t,v}(D) = {v^{-1}}_D(t)$, where ${v^{-1}}_D(t)$ is t's derivation in D according to v, and $TQ_{t,v}$ is independent of database instance D. We can similarly define the tracing query for a view tuple set T, and denote it as $TQ_{T,v}(D)$. □

2.4.2 Tracing Queries for SPJ Views

All SPJ views can be transformed into the form $\pi_A(\sigma_C(R_1 \bowtie \cdots \bowtie R_m))$ using a sequence of algebraic transformations [Ull89b]. We call this form the *SPJ canonical form*. From Theorem 2.3.11, we know that SPJ transformations do not affect view tuple derivations. Thus, given an SPJ view, we first transform it into SPJ canonical form, then compute its tuple derivations systematically using a single tracing query based on the canonical form. We first introduce an additional operator used in tracing queries for SPJ views.

Definition 2.4.2 (*Split* Operator) Let T be a table with schema $\mathbf{T}$. The operator *Split* breaks T into a list of tables; each table in the list is a projection of T onto a set of attributes $A_i \subseteq \mathbf{T}$, $i = 1..m$.

$$\mathit{Split}_{A_1,\ldots,A_m}(T) = \langle \pi_{A_1}(T), \ldots, \pi_{A_m}(T) \rangle$$

Theorem 2.4.3 (Tracing Query for an SPJ View) Let D be a database with base tables $R_1, \ldots, R_m$, and let $V = v(D) = \pi_A(\sigma_C(R_1 \bowtie \cdots \bowtie R_m))$ be an SPJ view over D. Given tuple $t \in V$, t's derivation in D according to v can be computed by applying the following query to the base tables:

$$TQ_{t,v} = \mathit{Split}_{\mathbf{R_1},\ldots,\mathbf{R_m}}(\sigma_{C \wedge A=t}(R_1 \bowtie \cdots \bowtie R_m))$$

Given a tuple set $T \subseteq V$, T's derivation tracing query is:

$$TQ_{T,v} = \mathit{Split}_{\mathbf{R_1},\ldots,\mathbf{R_m}}(\sigma_C(R_1 \bowtie \cdots \bowtie R_m) \ltimes T)$$

where $\ltimes$ represents the relational semijoin operator [Ull89a]. □

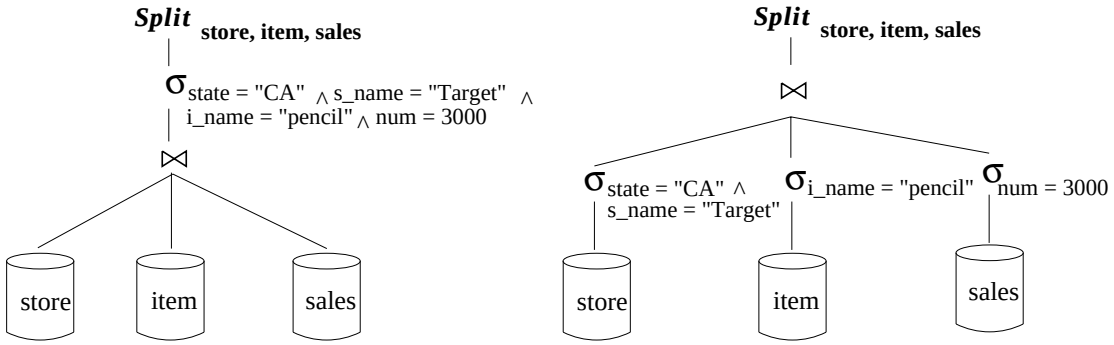
Proof: We need to prove:

$$v^{-1}{}_D(t) = \mathit{Split}_{\mathbf{R_1},\ldots,\mathbf{R_m}}(\sigma_{A=t \wedge C}(R_1 \bowtie \ldots \bowtie R_m))$$
$$v^{-1}{}_D(T) = \mathit{Split}_{\mathbf{R_1},\ldots,\mathbf{R_m}}(\sigma_C(R_1 \bowtie \ldots \bowtie R_m) \ltimes T)$$

Let $V_2 = R_1 \bowtie \cdots \bowtie R_m$, $V_1 = \sigma_C(V_2)$, and $V = \pi_A(V_1)$. According to Theorem 2.3.8:

$$\begin{aligned} v^{-1}{}_D(t) &= {v_1}^{-1}{}_D({\pi_A}^{-1}{}_{V_1}(t)) = {v_1}^{-1}{}_D(\sigma_{A=t}(V_1)) \\ &= {v_2}^{-1}{}_D({\sigma_C}^{-1}{}_{V_2}(\sigma_{A=t}(V_1))) = {v_2}^{-1}{}_D(\sigma_{A=t}(V_1)) \\ &= \mathit{Split}_{\mathbf{R_1},\ldots,\mathbf{R_m}}(\sigma_{A=t}(V_1)) \\ &= \mathit{Split}_{\mathbf{R_1},\ldots,\mathbf{R_m}}(\sigma_{A=t}(\sigma_C(R_1 \bowtie \ldots \bowtie R_m))) \\ &= \mathit{Split}_{\mathbf{R_1},\ldots,\mathbf{R_m}}(\sigma_{A=t \wedge C}(R_1 \bowtie \ldots \bowtie R_m)) \end{aligned}$$

$$\begin{aligned} v^{-1}{}_D(T) &= \bigcup_{t \in T} v^{-1}{}_D(t) = \bigcup_{t \in T} \mathit{Split}_{\mathbf{R_1},\ldots,\mathbf{R_m}}(\sigma_{A=t \wedge C}(R_1 \bowtie \ldots \bowtie R_m)) \\ &= \mathit{Split}_{\mathbf{R_1},\ldots,\mathbf{R_m}}(\bigcup_{t \in T} \sigma_{A=t \wedge C}(R_1 \bowtie \ldots \bowtie R_m) \\ &= \mathit{Split}_{\mathbf{R_1},\ldots,\mathbf{R_m}}(\sigma_{A \in T}(\sigma_C(R_1 \bowtie \ldots \bowtie R_m))) \\ &= \mathit{Split}_{\mathbf{R_1},\ldots,\mathbf{R_m}}(\sigma_C(R_1 \bowtie \ldots \bowtie R_m) \ltimes T) \end{aligned}$$



(a) Unoptimized (b) Optimized

Figure 2.12: Tracing queries for $\langle$`Target`, `pencil`, `3000`$\rangle$ in view `Calif`

.

Example 2.4.4 (Tracing Query for `Calif`) Recall Q1 over view `Calif` in Example 2.1.1, where we asked about the derivation of tuple $\langle$`Target`, `pencil`, `3000`$\rangle$. Figure 2.12(a) shows the tracing query for $\langle$`Target`, `pencil`, `3000`$\rangle$ in `Calif` according to Theorem 2.4.3. The reader may verify that by applying the tracing query to the source tables in Figures 2.1, 2.2, and 2.3, we obtain the derivation result in Figure 2.6. □

2.4.3 Tracing Query Optimizations

The derivation tracing queries in Section 2.4.2 clearly can be optimized for better performance. For example, the simple technique of pushing selection conditions below the join operator is especially applicable in tracing queries, and can significantly reduce query cost. Figure 2.12(b) shows the optimized tracing query for the `Calif` tuple. If the view contains the key attributes of each base table, the tracing query can be even simpler, as shown in the following theorem.

Theorem 2.4.5 (Derivation Tracing using Key Information) Let R_i be a base table with key attributes K_i, $i = 1..m$, and suppose view $V = \pi_A(\sigma_C(R_1 \bowtie \cdots \bowtie R_m))$ include all base table keys (i.e., $K_i \in A$, $i = 1..m$). Then view tuple t's derivation is $\langle \sigma_{K_1=t.K_1}(R_1), \ldots, \sigma_{K_m=t.K_m}(R_m) \rangle$.

Proof: Assuming $\langle R_1^*, \ldots, R_m^* \rangle$ is the derivation of t, we need to prove that $R_i^* = \sigma_{K_i=t.K_i}(R_i)$, for $i = 1..m$. From Theorem 2.5.2, we know that

$$\langle R_1^*, \ldots, R_m^* \rangle = \mathit{Split}_{\mathbf{R_1},\ldots,\mathbf{R_m}}(\sigma_{A=t \wedge C}(R_1 \bowtie \ldots \bowtie R_m)).$$

1. $\forall t^* \in R_i^* = \pi_{\mathbf{R_i}}(\sigma_{A=t \wedge C}(R_1 \bowtie \ldots \bowtie R_m))$: Since $K_i \in A$ and $K_i \in R_i$, $t^*.K_i = t.K_i$. Therefore, $t^* \in \sigma_{K_i=t.K_i}(R_i)$.

2. $\forall t^* \in \sigma_{K_i=t.K_i}(R_i)$: $t^*.K_i = t.K_i$, and therefore $t^* \in R_i^*$. Because otherwise, $\exists t' \in R_i^*$, such that $t'.K_i = t.K_i$. Then there exist two tuples $t^* \neq t'$ in R_i with the same K_i value, which conflicts with the key constraint.

Therefore $R_i^* = \sigma_{K_i=t.K_i}(R_i)$, for $i = 1..m$. □

According to Theorem 2.4.5, we can use key information to fetch the derivation of a tuple directly from the base tables, without performing a join. The worst-case query complexity is reduced from $O(n^m)$ to $O(mn)$, where n is the maximum size of the base tables, and m is the number of base tables on which v is defined.

We have shown that tuple derivations for SPJ views can be traced in a simple manner. For more complex views with aggregations or set operators, we cannot compute tuple derivations using a single query over the base tables. In the next two sections, we present recursive tracing algorithms for these views.

2.5 Derivation Tracing for ASPJ Views

In this section, we consider SPJ views with aggregation (*ASPJ views*). Although we have shown that no intermediate results are required for SPJ view derivation tracing, some ASPJ views are not traceable without storing or recomputing certain intermediate results. For example, Q2 in Example 2.1.2 asks for the derivation of tuple $t = \langle 5400 \rangle$ in the view `Clothing`. It is not possible to compute t's derivation directly from `store`, `item`, and `sales`, because `total` is the only column of view `Clothing`, and it is not contained in the base tables at all. Therefore, we cannot find t's derivation by knowing only that $t.\texttt{total} = 5400$. In order to trace t's derivation in the base tables correctly, we need its

derivation $\langle$Macy's, $5400\rangle$ in the intermediate aggregation result, whose grouping attribute `s_name` serves as a "bridge" that connects the base tables and the final view table.

We introduce a canonical form for ASPJ views in Section 2.5.1. In Section 2.5.2, we specify a derivation tracing query for "one-level" ASPJ views. We then develop a recursive tracing algorithm for complex ASPJ views and justify its correctness in Section 2.5.3. As mentioned above, intermediate (aggregation) results in the view evaluation are needed for derivation tracing. Relevant portions of the intermediate results can be recomputed from the base tables when needed, or the entire results can be stored as materialized *auxiliary views* in the warehouse; this issue is explored in depth in Chapter 3. In the remainder of this chapter we simply assume that all intermediate aggregation results are available.

2.5.1 ASPJ Canonical Form

Unlike SPJ views, ASPJ views do not have a simple canonical form, because in an ASPJ view definition some selection, projection, and join operators cannot be pushed above or below the aggregation operators [GHQ95]. View `Clothing` in Figure 2.7 is such an example, where the selection `total > 5000` cannot be pushed below the aggregation, and the selection `category = "clothing"` cannot be pulled above the aggregation. However, by commuting and combining some SPJ operators [Ull89b], it is possible to transform any ASPJ view query tree into a form composed of α-π-σ-$\bowtie$ operator sequences that we call *ASPJ segments*. We call this segmented form the *ASPJ canonical form*. An ASPJ segment may omit operators, although each segment in the ASPJ canonical form except the outermost must include an aggregation operator (otherwise the segment would be merged with an adjacent segment). For definition purposes, when a unary operator is missing we assume there is a corresponding *trivial* operator to take its place. The trivial aggregation on table T is $\alpha_{\mathbf{T}}$, the trivial projection is $\pi_{\mathbf{T}}$, and the trivial selection is σ_{true}.

Definition 2.5.1 (ASPJ Canonical Form) Let v be an ASPJ view over database D.

1. $v = R$, where R is a base table in D, is in *ASPJ canonical form*.
2. $v = \alpha_{G,aggr(B)}(\pi_A(\sigma_C(v_1 \bowtie \cdots \bowtie v_k))$ is in *ASPJ canonical form* if v_j is an ASPJ view in ASPJ canonical form with a nontrivial topmost aggregation operator, $j = 1..k$. □

Both of our example views `Calif` and `Clothing` as defined in Section 2.1 are in canonical form, while view $V = \sigma_C(R) \bowtie \pi_A(S)$ is not in canonical form.

As mentioned in Section 2.3, although we can trace any view's derivation by tracing through one operator at a time based on the original view definition, that approach requires us to store or recompute the intermediate results of every operator, which can be extremely expensive. Transforming an ASPJ view definition into canonical form allows us to store or recompute fewer intermediate results, by tracing through all operators in each segment together, as we will see in the next section.

2.5.2 Derivation Tracing Queries for One-Level ASPJ Views

A view defined by one ASPJ segment is called a *one-level ASPJ view*. Similar to SPJ views, we can use one query to trace tuple derivations for a one-level ASPJ view.

Theorem 2.5.2 (Tracing Query for a One-Level ASPJ View) Given one-level ASPJ view $V = v(R_1, \ldots, R_m) = \alpha_{G,aggr(B)}(\pi_A(\sigma_C(R_1 \bowtie \cdots \bowtie R_m)))$ and tuple $t \in V$, t's derivation in $R_1, \ldots, R_m$ according to v can be computed using the following tracing query:

$$TQ_{t,v} = \mathit{Split}_{\mathbf{R_1},\ldots,\mathbf{R_m}}(\sigma_{C \wedge G=t.G}(R_1 \bowtie \cdots \bowtie R_m))$$

Given tuple set $T \subseteq V$, T's derivation tracing query is:

$$TQ_{T,v} = \mathit{Split}_{\mathbf{R_1},\ldots,\mathbf{R_m}}(\sigma_C(R_1 \bowtie \cdots \bowtie R_m) \ltimes T)$$

. □

Proof: Let $V_1 = \pi_A(\sigma_C(T_1 \bowtie \cdots \bowtie T_m))$ and let $V = \alpha_{G,aggr(B)}(V_1)$.

$$\begin{aligned} {v^{-1}}_D(t) &= {v_1}^{-1}{}_D({\alpha_{G,aggr(B)}}^{-1}{}_{V_1}(t)) = {v_1}^{-1}{}_D(\sigma_{G=t.G}(V_1)) \\ &= Split_{\mathbf{T_1},\ldots,\mathbf{T_m}}(\sigma_C(T_1 \bowtie \ldots \bowtie T_m) \ltimes \sigma_{G=t.G}(\pi_A(\sigma_C(T_1 \bowtie \cdots \bowtie T_m)))) \\ &= Split_{\mathbf{T_1},\ldots,\mathbf{T_m}}(\sigma_{G=t.G \wedge C}(T_1 \bowtie \ldots \bowtie T_m)) \end{aligned}$$

$$
\begin{aligned}
{v^{-1}}_D(T) &= \bigcup_{t \in T} {v^{-1}}_D(t) = \bigcup_{t \in T} Split_{\mathbf{R_1},\ldots,\mathbf{R_m}}(\sigma_{G=t.G \wedge C}(R_1 \bowtie \ldots \bowtie R_m)) \\
&= Split_{\mathbf{R_1},\ldots,\mathbf{R_m}}(\bigcup_{t \in T} \sigma_{G=t.G \wedge C}(R_1 \bowtie \ldots \bowtie R_m) \\
&= Split_{\mathbf{R_1},\ldots,\mathbf{R_m}}(\sigma_C(R_1 \bowtie \ldots \bowtie R_m) \ltimes T)
\end{aligned}
$$

. □

Here too, evaluation of the tracing query can be optimized in various ways as discussed in Section 2.4.3.

We note that previous work [KLM+97] also defines a notion of tuple derivations, and the views they consider correspond to our one-level ASPJ views. In [KLM+97] the *derivation set* of $t \in V$ is defined as $\langle R'_1, \ldots, R'_m \rangle$, where $R'_i = \sigma_{C_i \wedge G_i = t.G_i}(R_i)$ such that $C_i \subseteq C$ and $G_i \subseteq G$ are the selection conditions and grouping attributes local to R_i. Although this approach localizes derivation tracing to a single base table at a time, derivation sets as defined in [KLM+97] are not as accurate as our tuple derivations: those tuples in R'_i that do not join with the other tables could not have participated in the derivation of t.

2.5.3 Derivation Tracing Algorithm for Multi-Level ASPJ Views

Given a general ASPJ view definition, we first transform the view into ASPJ canonical form, divide it into a set of ASPJ segments, and define an intermediate view for each segment.

Example 2.5.3 (ASPJ Segments and Intermediate Views for `Clothing`) Recall the view `Clothing` in Example 2.1.2. We can rewrite its definition in ASPJ canonical form with two segments, and introduce an intermediate view `AllClothing` as shown in Figure 2.13. □

We then trace a tuple's derivation by recursively tracing through the hierarchy of intermediate views top-down. At each level, we use the tracing query for a one-level ASPJ view to compute derivations for the current tracing tuples with respect to the views or base tables at the next level below.

Figure 2.14 presents our recursive derivation tracing algorithm for a general ASPJ view. Given a view definition v in ASPJ canonical form, and tuple $t \in v(D)$, procedure

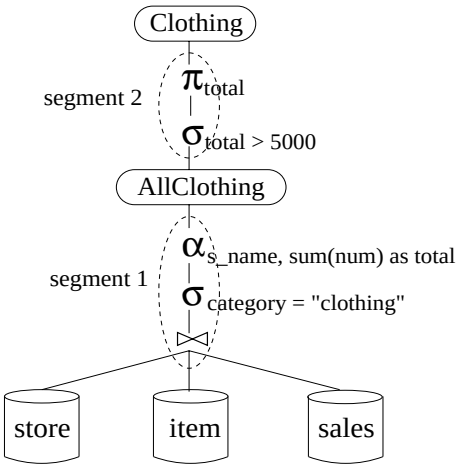


Figure 2.13: ASPJ segments and intermediate view for `Clothing`

`TupleDerivation(`t, v, D`)` computes the derivation of tuple t according to v over D. The main algorithm, procedure `TableDerivation(`T, v, D`)`, computes the derivation of a tuple set $T \subseteq v(D)$ according to v over D. As discussed earlier, we assume that $v = v'(V_1, \ldots, V_k)$ where v' is a one-level ASPJ view, and $V_j = v_j(D)$ is available as a base table or an intermediate view, $j = 1..k$. The procedure first computes T's derivation $\langle V_1^*, \ldots, V_k^* \rangle$ in $\langle V_1, \ldots, V_k \rangle$ using the one-level ASPJ view tracing query `TQ(`$T, v', \langle V_1, \ldots, V_m \rangle$`)` from Theorem 2.5.2. It then calls procedure `TableListDeriv(` $\{V_1^*, \ldots, V_k^*\}, \{v_1, \ldots, v_k\}, D$`)`, which computes (recursively) the derivation of each tuple set V_j^* according to v_j, $j = 1..k$, and concatenates the results to form the derivation of the entire list of view tuple sets.

Example 2.5.4 (Recursive Derivation Tracing) We divided the view `Clothing` into two segments in Example 2.5.3. We assume that the contents of the intermediate view `AllClothing` are available (shown in Figure 2.15). According to our algorithm, we first compute the derivation T_1^* of $\langle$`5400`$\rangle$ in `AllClothing` to obtain $T_1^* = \{\langle$`Macy's, 5400`$\rangle\}$, then trace T_1^*'s derivation to the base tables to obtain the derivation result in Figure 2.8. □

Note that we do not necessarily materialize complete intermediate aggregation views such as `AllClothing`. In fact, there are many choices of what (if anything) to store. The issue of storing versus recomputing the intermediate information needed for derivation tracing is covered in Chapter 3.

procedure TupleDerivation(t, v, D)
in: a tracing tuple t, a view definition v, and a database D
out: t's derivation in D according to v
1 **return** (TableDerivation($\{t\}, v, D$));

procedure TableDerivation(T, v, D)
in: a tracing tuple set T, a view definition v, and a database D
out: T's derivation in D according to v
1 **if** $v = R \in D$ **then return** ($\langle T \rangle$);
2 // otherwise $v = v'(v_1, \ldots, v_k)$ where v' is a one-level ASPJ view
3 // $V_j = v_j(D)$ is an intermediate view or a base table, $j = 1..k$
4 $\langle V_1^*, \ldots, V_k^* \rangle \leftarrow$ TQ($T, v', \{V_1, \ldots, V_k\}$);
5 **return** (TableListDeriv($\langle V_1^*, \ldots, V_k^* \rangle, \{v_1, \ldots, v_k\}, D$));

procedure TableListDeriv($\langle T_1, \ldots, T_k \rangle, \{v_1, \ldots, v_k\}, D$)
in: a list of tuple sets $T_1, \ldots, T_k$ to be traced,
a list of view definitions $v_1 \ldots, v_k$, and a database D
out: the concatenation of T_i's derivations according to v_i, $i = 1..k$
1 $D^* \leftarrow \varnothing$;
2 **for** $j \leftarrow 1$ to k **do**
3 $D^* \leftarrow D^* \circ$ TableDerivation(T_j, v_j, D);
4 **return** (D^*);

Figure 2.14: Derivation tracing algorithm for ASPJ views

To justify the correctness of our algorithm, we claim the following:

1. Transforming a view into ASPJ canonical form does not affect its derivations. We can "canonicalize" an ASPJ view by transforming each segment between adjacent aggregation operators into its SPJ canonical form. The process consists only of SPJ transformations [Ull89b]. Theorem 2.3.11 shows that derivations are unchanged by SPJ transformations.

2. It is correct to trace derivations recursively down the view definition tree. From Theorem 2.3.8, we know that derivations are transitive through levels of the view definition tree. Thus, when tracing tuple derivations for a canonicalized ASPJ view, we can first divide its definition into one-level ASPJ views, and then compute derivations for the intermediate views in a top-down manner.

s_name	total
Target	1400
Macy's	5400

Figure 2.15: `AllClothing` table

We have so far introduced a simple tracing query for SPJ and one-level ASPJ views, and a recursive tracing algorithm for general ASPJ views. In the next section, we consider derivation tracing for even more general views that include the set operators union and difference.

2.6 Derivation Tracing for Views with Set Operators

In this section, we extend our derivation tracing algorithm for general ASPJ views that allow arbitrary use of set union and difference operators in the view definition. In Section 2.6.1, we briefly review tuple derivations for set operators, and provide an example justifying the lineage definition for the difference operator. In Section 2.6.2, we incorporate set operators into our view definition canonical form, and we present a procedure to transform any ASPJ view with set operators (referred to as a *general* view) into canonical form. Finally, in Section 2.6.3 we present a recursive algorithm that traces tuple derivations for general views.

2.6.1 Tuple Derivations for Set Operators

Recall from Theorem 2.3.4 that the tuple derivations for set operators are:

$$\cup^{-1}{}_{\langle T_1,\ldots,T_m\rangle}(t) = \langle \sigma_{\mathbf{T_1}=t}(T_1), \ldots, \sigma_{\mathbf{T_m}=t}(T_m)\rangle, \text{ for } t \in T_1 \cup \cdots \cup T_m$$
$$-^{-1}{}_{\langle T_1,T_2\rangle}(t) = \langle \{t\}, T_2\rangle, \text{ for } t \in T_1 - T_2$$

For the union operator, given $t \in T_1 \cup \cdots \cup T_m$, each tuple t from any input table $T_1, \ldots, T_m$ contributes to t. For the difference operator, given $t \in T_1 - T_2$, the tuple t from T_1 and all tuples in T_2 contribute to t. Recall from Definition 2.3.1 that a tuple's derivation *fully* explains why the tuple appears in the operation result. In particular, a tuple t appears in

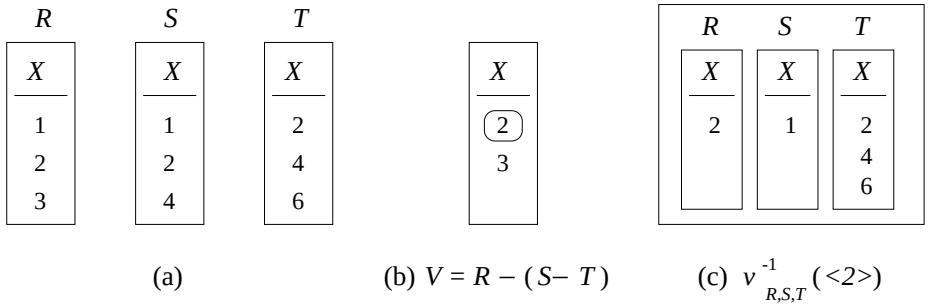


Figure 2.16: Tuple derivation for set difference

$T_1 - T_2$ not only because $\exists t \in T_1$, but also because $\neg\exists t \in T_2$. Therefore, all tuples in T_2 contribute to $t \in T_1 - T_2$ in the sense that they ensure $\neg\exists t \in T_2$. Example 2.6.1 further explains the necessity of including T_2 in the derivation of $t \in T_1 - T_2$.

Example 2.6.1 (Tuple Derivation for Difference) Consider tables R, S, T in Figure 2.16(a), and view $V = R - (S - T)$ in Figure 2.16(b). Although we have not yet given our tracing algorithm for general views, the derivation of tuple $t = \langle 2\rangle \in V$ using our algorithm (and consistent with the tuple derivation definitions given above) is as shown in Figure 2.16(c). Notice that tuple $\langle 2\rangle$ would not have appeared in V without tuple $\langle 2\rangle$ in T. So $\langle 2\rangle \in T$ obviously contributes in a significant way to $\langle 2\rangle \in V$. In general, contributions of this sort are not exhibited in derivation tracing for a tuple t unless we include in t's derivation all tuples in (or contributing to) the second operand of the set difference operator.[5] □

2.6.2 Canonical Form for General Views

We now define an extended canonical form that accommodates views with union and difference operators, as well as aggregation, selection, projection, and join operators. The extended canonical form is composed of two types of segments: *AUSPJ-segments*, which are operator sequences in the form α-$\cup$-π-σ-$\bowtie$, and *D-segments*, which contain a single

[5]By including all tuples in the second operand, we also include tuples that do not contribute directly to the view tuple, e.g., tuples $\langle 4\rangle$ and $\langle 6\rangle$ in T of Figure 2.16(c). However, no intuitive lineage definition we know of can include, for example, tuple $\langle 2\rangle$ in T without tuples $\langle 4\rangle$ and $\langle 6\rangle$, in the general case.

difference operator. An AUSPJ-segment may omit any of the operators in the operator sequence α-$\cup$-π-σ-$\bowtie$, but it must satisfy one of the following:

1. It has an aggregation operator on the top.
2. It is directly below a D-segment.
3. It is below a join operator, and has a union operator on the top.
4. It is the top segment in the view definition, which means that no more segments lie above this segment.

When a α, π, or σ operator is missing from an AUSPJ-segment, for definition purposes we assume a trivial operator is present, as in Section 2.5.1. In Figure 2.17, we provide a procedure `Canonicalize` that transforms a general view definition tree into canonical form, divides the transformed view definition into segments, and associates an intermediate view with each segment. We perform canonicalizing transformations based on the following equivalences:

1. pulling up $\cup$: $\pi_A(R \cup S) \equiv \pi_A(R) \cup \pi_A(S)$ and $\sigma_C(R \cup S) \equiv \sigma_C(R) \cup \sigma_C(S)$
2. pulling up π: $\sigma_C(\pi_A(R)) \equiv \pi_A(\sigma_C(R))$ and $\pi_A(R) \bowtie_\theta \pi_B(S) \equiv \pi_{A \cup B}(R \bowtie_\theta S)$[6]
3. pulling up σ: $\sigma_C(R) \bowtie \sigma_{C'}(S) \equiv \sigma_{C \wedge C'}(R \bowtie S)$

As in Figure 2.17, we first pull up the union operators in the view definition tree as far as possible. We then pull up projection and selection operators, and finally merge the same algebra operators when they appear adjacent to each other. Notice that in the canonicalizing procedure, we pull projection and selection operators above join operators, so that we can compress the operator tree into as few segments as possible. This helps to reduce the total tracing cost, and possibly the number of intermediate views that are stored. However, when we actually apply the tracing query for each segment to obtain a derivation, we can

[6]Recall that we use $\bowtie$ throughout the thesis to represent natural join, theta join, and cross-product (Section 2.2). For this equivalence to hold in general, it is necessary to make the theta explicit: it does not apply for natural join.

procedure Canonicalize(v_0)
in: a general view definition tree v_0
out: a canonicalized view definition v with an intermediate view associated with each segment in v
1 copy v_0 to v;
2 pull union operators above selections and projections in v;
3 pull projection operators above joins and selections in v;
4 pull selection operators above joins in v;
5 merge the same algebra operators adjacent to each other in v;
6 divide v into segments;
7 define an intermediate view over each segment in v;
8 **return** (v);

Figure 2.17: Canonicalizing general view definitions

sometimes push selection and projection operators back below the join to reduce the cost of the tracing query, as discussed in Section 2.5.2.

Having obtained the canonicalized definition tree, we divide it into AUSPJ-segments and D-segments, based on the following rules:

1. Every aggregation and difference operator begins a segment.
2. Every non-leaf child of a difference or join operator begins a segment.
3. The topmost node begins a segment.

Finally, we define an intermediate view for each segment. As with multi-level ASPJ views, when tracing tuple derivations for a general view, each segment (intermediate view) becomes a "tracing unit". That is, we recursively trace tuple derivations down the view definition tree, through one segment at a time, until we reach the base tables.

Theorem 2.6.2 (Extended Canonical Form) Procedure `Canonicalize` in Figure 2.17 returns a view v in canonical form. v is equivalent to the given view v_0, and the two views have equivalent tuple derivations. □

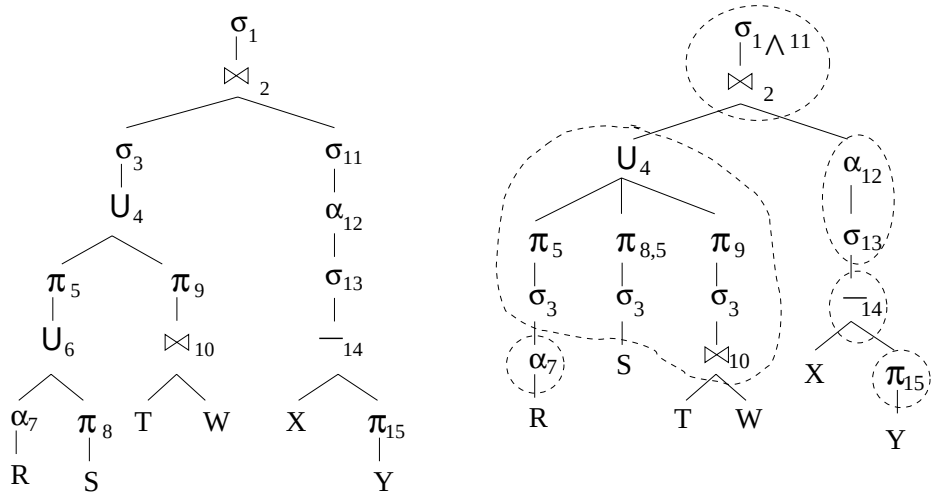
Proof: If an operator o is the parent of another operator o' in a view definition tree, we say $o \rightarrow o'$. We first prove that procedure `Canonicalize(`v_0`)` returns the canonical form. In `Canonicalize`, after we pull up the union operators in a top-down manner, there are

only the following possibilities for a union node ($\cup$) and the node right above it: $- \rightarrow \cup$, $\alpha \rightarrow \cup$, $\bowtie \rightarrow \cup$, $\cup \rightarrow \cup$, or nothing above $\cup$. For any other possibility, we can always pull the union operator further up. Pulling up the projections afterwards does not affect the above patterns. At the same time, there are only the following patterns for projections: $- \rightarrow \pi$, $\alpha \rightarrow \pi$, $\cup \rightarrow \pi$, $\pi \rightarrow \pi$, or nothing above π. Similarly, pulling up selections does not affect any of the above patterns, and there are only the following patterns for selections: $- \rightarrow \sigma$, $\alpha \rightarrow \sigma$, $\cup \rightarrow \sigma$, $\pi \rightarrow \sigma$, $\sigma \rightarrow \sigma$, or nothing above σ. For join, any operator could be its parent: $- \rightarrow \bowtie$, $\alpha \rightarrow \bowtie$, $\cup \rightarrow \bowtie$, $\pi \rightarrow \bowtie$, $\sigma \rightarrow \bowtie$, $\bowtie \rightarrow \bowtie$, or nothing above $\bowtie$. After merging the same-type operators, we are left with 18 possible two-operator sequences out of 36 total possibilities in the definition tree after applying `Canonicalize`. With the given 18 sequences, any two adjacent operators either belong to different segments, or are in the order consistent with the sequence order in an AUSPJ-segment. Therefore, procedure `Canonicalize(`v_0`)` returns a canonical form.

We now prove the view equivalence after view transformations in `Canonicalize`. We know from [Ull89b] that projection can be pushed above selections and joins while keeping a query (or view) equivalent. Here, we prove that union operators can be also pushed above selections and projections while keeping the transformed view equivalent to the original view. In other words, we need to prove (1) $\pi_A(R \cup S) = \pi_A(R) \cup \pi_A(S)$; (2) $\sigma_A(R \cup S) = \sigma_A(R) \cup \sigma_A(S)$. For (1), $\forall t$: $t \in \pi_A(R \cup S) \Leftrightarrow \exists t' \in R \cup S$ such that $t'.A = t$ $\Leftrightarrow \exists t' \in R$ or $\in S$ such that $t'.A = t \Leftrightarrow t \in \pi_A(R)$ or $\pi_A(S) \Leftrightarrow t \in \pi_A(R) \cup \pi_A(S)$. We can similarly prove (2).

Finally, we can easily extend Lemma 2.3.10 to prove that equivalent SPJ views with set unions have equivalent derivations, which proves the derivation equivalence after applying `Canonicalize`. □

Example 2.6.3 (Canonical Form for a View with Set Operators) Consider the general view definition in Figure 2.18(a). Figure 2.18(b) shows the canonical form and its segments. Notice that σ_{11} is pulled up and merged with σ_1 to obtain $\sigma_{1 \wedge 11}$, π_8 is subsumed by π_5, and $\cup_6$ is merged into $\cup_4$. □



(a) original view definition tree (b) canonicalized view definitior

Figure 2.18: Canonical form for a view with set operators

2.6.3 Derivation Tracing Algorithm for General Views

Once we have obtained a canonicalized view definition, we can trace its tuple derivations through one segment at a time. Theorem 2.6.4 presents the derivation tracing query for a one-level AUSPJ view (or an AUSPJ segment). Tuple derivations for a D-segment are traced based on Theorem 2.3.4.

Theorem 2.6.4 (Tracing Query for a One-level AUSPJ View) Consider a one-level AUSPJ view $V = v(R_1^1, \ldots, R_{l_k}^k) = \alpha_{G,aggr(B)}(\bigcup_{j=1..k} \pi_A(\sigma_{C_j}(R_1^j \bowtie \cdots \bowtie R_{l_j}^j))$. Given tuple $t \in V$, t's derivation according to v can be computed by applying the following query to the base tables:

$$TQ_{t,v} = \bigodot_{j=1..k} \mathit{Split}_{\mathbf{R_1^j},\ldots,\mathbf{R_{l_j}^j}}(\sigma_{C \wedge G=t.G}(R_1^j \bowtie \cdots \bowtie R_{l_j}^j))$$

Given tuple set $T \subseteq V$, T's derivation tracing query is:

$$TQ_{T,v} = \bigodot_{j=1..k} \mathit{Split}_{\mathbf{R_1^j},\ldots,\mathbf{R_{l_j}^j}}(\sigma_C(R_1^j \bowtie \cdots \bowtie R_{l_j}^j) \ltimes T)$$

For the special case where the traced tuple set is the entire view table V (which will appear

later in our recursive tracing algorithm for general views), we use a flag "**ALL**" to specify that the entire view table is to be traced, and the tracing query can be simplified by removing the semijoin:

. $$TQ_{\mathbf{ALL},v} = \bigodot_{j=1..k} Split_{\mathbf{R_1^j},\ldots,\mathbf{R_{l_j}^j}}(\sigma_C(R_1^j \bowtie \cdots \bowtie R_{l_j}^j))$$ □

Proof: Let:

$$V_j = \pi_A(\sigma_C(T_{j,1} \bowtie \cdots \bowtie T_{j,l_j}))$$
$$V' = \bigcup_{j=1..k} (V_j)$$
$$V = \alpha_{G,aggr(B)}(V')$$

According to Theorem 2.3.8 and the derivation for set union operator, we have:

$$\begin{aligned} {v^{-1}}_D(t) &= {v'^{-1}}_D({\alpha_{G,aggr(B)}}^{-1}{}_{V'}(t)) \\ &= {v'^{-1}}_D(\sigma_{G=t.G}(V')) \\ &= \bigodot_{j=1..k} {v_j}^{-1}{}_D(\sigma_{G=t.G}(V'))) \\ &= \bigodot_{j=1..k} Split_{\mathbf{T_{j,1}},\ldots,\mathbf{T_{j,l_j}}}(\sigma_{G=t.G \wedge C}(T_{j,1} \bowtie \ldots \bowtie T_{j,l_j})) \end{aligned}$$

${v^{-1}}_D(T)$ can be computed from ${v^{-1}}_D(t)$ similarly as in the proof of Theorem 2.5.2. We now prove ${v^{-1}}_D(V)$ based on ${v^{-1}}_D(T)$.

$$\begin{aligned} {v^{-1}}_D(V) &= \bigodot_{j=1..k} Split_{\mathbf{T_{j,1}},\ldots,\mathbf{T_{j,l_j}}}(\sigma_C(T_{j,1} \bowtie \ldots \bowtie T_{j,l_j}) \ltimes V) \\ &= \bigodot_{j=1..k} Split_{\mathbf{T_{j,1}},\ldots,\mathbf{T_{j,l_j}}}(\sigma_C(T_{j,1} \bowtie \ldots \bowtie T_{j,l_j})) \end{aligned}$$

. □

The recursive tracing algorithm for ASPJ views (Figure 2.14) can now be extended to handle general views by modifying the procedure `TableDerivation`. The new specification for `TableDerivation(`T, v, D`)` is given in Figure 2.19. As we recursively trace down a canonicalized view definition tree, at each level we are tracing a view (or a base table) v that has one of three forms: (1) $v = R$ where R is a base table; (2) $v = v'(v_1, \ldots, v_k)$ where v' is an AUSPJ-segment and v_i is an intermediate view or a base table, $i = 1..k$;

procedure TableDerivation(T, v, D)
in: a tracing tuple set T (or the special symbol **ALL**),
a view definition v, and a database D
out: T's derivation in D according to v
1 **case** $v = R \in D$:
2 **if** $T =$ **ALL then return** ($\langle R \rangle$); **else return** ($\langle T \rangle$);
3 **case** $v = v'(v_1, \ldots, v_k)$:
4 // v' is a one-level ASPJ view
5 // $V_j = v_j(D)$ is an intermediate view or a base table, $j = 1..k$
6 $\langle V_1^*, \ldots, V_k^* \rangle \leftarrow$ TQ($T, v', \{V_1, \ldots, V_k\}$);
7 **return** (TableListDeriv($\langle V_1^*, \ldots, V_k^* \rangle, \{v_1, \ldots, v_k\}, D$));
8 **case** $v = v_1 - v_2$:
9 // $V = v(D)$
10 // $V_j = v_j(D)$ is an intermediate view or a base table, $j = 1, 2$
11 **if** $T =$ **ALL then** $T \leftarrow V$;
12 **return** (TableListDeriv($\langle T, \mathbf{ALL} \rangle, \langle v_1, v_2 \rangle, D$));

Figure 2.19: Derivation tracing algorithm for general views

(3) $v = v_1 - v_2$ where v_i is an intermediate view or a base table, $i = 1, 2$. These three cases map to the case statement in Figure 2.19. For derivation tracing, we may need to store or recompute the contents of some intermediate views, including the contents of the v_i's in case (2) and v in case (3). For case (2), to trace the derivation of T according to v, we first apply the AUSPJ tracing query TQ from Theorem 2.6.4 to $V_1, \ldots, V_k$ in order to obtain t's derivation V_i^* in V_i, $i = 1..k$. We then recursively trace the derivations of the V_i^*'s through lower segments. For case (3), we trace the derivation of T according to v_1 and the derivation of all tuples in $v_2(D)$ according to v_2, as discussed in Section 2.6.1. Recall from Theorem 2.6.4 that we do not need to store or recompute the actual view table $v_2(D)$. Instead, we use a flag "**ALL**" to specify that the entire view table is to be traced.

2.7 Derivation Tracing with Bag Semantics

So far we have addressed the derivation tracing problem using set semantics: the base tables are assumed to be sets, and the view is defined using operators with set semantics so that all operation results, including the final view table, are also sets. In this section, we extend our

		base tables: no duplicates	base tables: duplicates possible
view	no duplicates	(1)	(2)
	duplicates possible	(3)	(4)

Table 2.1: Base table and view scenarios

definition for tuple derivations as well as our tracing algorithms to consider bag semantics. That is, we treat base tables and the results of σ, π, and $\bowtie$ operations as bags, and we use bag union ($\uplus$) and bag difference ($\dot{-}$).[7] Duplicate elimination, if desired, can be achieved using our aggregation (α) operator.

Duplicates can occur in base tables and/or in views. Consider the four scenarios in Table 2.1. Case (1) represents the scenario we have considered so far. Case (4) represents the most general scenario to be addressed in this section, and case (2) will use the same algorithms as (4). Case (3), in which views may have duplicates but base tables are known to have keys, allows us to perform certain optimizations, related to the key-based example we gave earlier in Section 2.5.2. In fact, we may choose to transform a case (4) scenario into case (3) by attaching system-generated tuple IDs to the base data, in order to apply a more efficient tracing procedure.

In general, the derivation tracing problem for set semantics considered so far is subsumed by the problem we are about to address for bag semantics. However, as we will see, the solutions we gave for set semantics are simpler and more efficient than those for bag semantics, so the separate treatment is worthwhile.

2.7.1 Tuple Derivations for Bag Semantics

Consider an operator *Op* with bag semantics. Given N copies of a tuple t in $Op(T_1, \ldots, T_m)$, there are at least N ways to derive t from the T_i's. Consider the following examples.

1. Let $R = \{\langle a\rangle, \langle a\rangle\}$, $S = \{\langle b\rangle\}$, and $V = R \times S = \{\langle a, b\rangle, \langle a, b\rangle\}$. A tuple $\langle a, b\rangle$ in

[7] Bag operators do not eliminate duplicates. For example, $\{a, b\} \uplus \{a, c\} = \{a, a, b, c\}$, and $\{a, a, b\} \dot{-} \{a, b, c\} = \{a\}$. Bag operator $\uplus$ corresponds to SQL's `UNION ALL` (as opposed to `UNION`); bag operator $\dot{-}$ is not supported by SQL.

V may be derived from the first $\langle a \rangle$ in R and the $\langle b \rangle$ in S, or from the second $\langle a \rangle$ in R and the $\langle b \rangle$ in S.

2. Let $R(A, B) = \{\langle a, b \rangle, \langle a, c \rangle\}$ and $V = \pi_A(R) = \{\langle a \rangle, \langle a \rangle\}$ where π preserves duplicates. A tuple $\langle a \rangle$ in V may be derived from $\langle a, b \rangle$ or $\langle a, c \rangle$ in R.

3. Let $R = \{\langle a \rangle\}$, $S = \{\langle a \rangle\}$, and $V = R \uplus S = \{\langle a \rangle, \langle a \rangle\}$. A tuple $\langle a \rangle$ in V may be derived from the $\langle a \rangle$ in R or the $\langle a \rangle$ in S.

4. Let $R = \{\langle a \rangle, \langle a \rangle\}$, $S = \{\langle a \rangle\}$, and $V = R \dot{-} S = \{\langle a \rangle\}$. The (single) tuple $\langle a \rangle$ in V may be derived from the first or the second $\langle a \rangle$ in R.

Thus, often there is no longer a unique derivation of t according to Definition 2.3.1. Given any operator other than bag difference, if there are N copies of t in the result then there are exactly N derivations of t, each of which derives one copy of t as in examples 1–3 above. Given a bag difference operator, the number of derivations of t may exceed the number of t's in the result, as in example 4 above.

More generally, given a view v, a tuple $t \in v(D)$ may have multiple derivations according to v. Sometimes, we may want to retrieve the derivations of a tuple one by one. Other times, we may prefer a complete set of contributing tuples, without distinguishing individual derivations. We introduce the concepts of *derivation set* and *derivation pool* in Definitions 2.7.2 and 2.7.3 to cover both cases. First, we redefine tuple derivations for a view under bag semantics, because our previous definition (Definition 2.3.5), although clearer, produces only derivation pools under bag semantics and not individual derivations. Definition 2.7.1 subsumes Definition 2.3.5.

Definition 2.7.1 (Tuple Derivations for a View) Let D be a database with base tables $R_1, \ldots, R_m$, and let $V = v(D)$ be a view over D. Consider a tuple $t \in V$.

1. $v = R_i$: $\{t\}$ is a *derivation of t according to v*.

2. $v = Op(v_1, \ldots, v_k)$, where v_j is a view definition over D, $j = 1..k$: Suppose $\langle T_1^*, \ldots, T_k^* \rangle$ is a derivation of t according to *Op* (by Definition 2.3.1), and $\langle {R_1^j}^*, \ldots, {R_{l_j}^j}^* \rangle$ is a derivation of T_j^* according to v_j over D (by this definition recursively), $j = 1..k$. Then $\langle {R_1^1}^*, \ldots, {R_{l_1}^1}^*, \ldots, {R_1^k}^*, \ldots, {R_{l_k}^k}^* \rangle$ is a *derivation of t according to v*.

Derivations for a tuple set T are constructed from all possible combinations of derivations for the tuples in T such that the derivations selected for any $t_1 = t_2 \in T$ are different. □

As we can see, the number of derivations for a tuple set T may be exponential in the number of tuples in T: if T contains n tuples, and each tuple has up to m derivations, then T may have as many as m^n derivations. Thus, enumerating all derivations for a tuple set (Definition 2.7.2) can be impractical, and we may prefer to trace the tuple set's derivation pool (Definition 2.7.3).

Definition 2.7.2 (Derivation Set) Let D be a database with base tables $R_1, \ldots, R_m$, and let $V = v(D)$ be a view over D. Given a tuple $t \in V$, the set of all of t's derivations according to v is called the *derivation set of t according to v*, denoted ${v^{-S}}_D(t)$.[8] The derivation set of a tuple set $T \subseteq Op(T_1, \ldots, T_m)$ is the set of all derivations of T, denoted ${v^{-S}}_D(T)$. □

Definition 2.7.3 (Derivation Pool) Let D be a database with base tables $R_1, \ldots, R_m$, and let $V = v(D)$ be a view over D. Given a tuple $t \in V$, let $\langle {R_1}^*, \ldots, {R_m}^* \rangle$ contain all tuples in any of t's derivations (based on Definition 2.3.5). $\langle {R_1}^*, \ldots, {R_m}^* \rangle$ is called the *derivation pool of t according to v*, denoted ${v^{-P}}_D(t)$. The derivation pool of a tuple set $T \subseteq V$ contains all tuples in the derivation pool of any tuple in T, and is denoted by ${v^{-P}}_D(T)$. □

Notice that a view tuple's derivation set and derivation pool contain exactly the same collection of base tuples, but organized in different ways. As we will see, their tracing procedures will differ considerably.

Example 2.7.4 (Multiple Derivations, Derivation Set, Derivation Pool) Consider a view `Stationery` defined over our original tables `store`, `item`, and `sales` from Figures 2.1–2.3: $\texttt{Stationery} = \pi_{\texttt{s\_name, i\_name}}(\sigma_{\texttt{category=“stationery”}}(\texttt{store} \bowtie \texttt{item} \bowtie \texttt{sales}))$, where π now does not eliminate duplicates. Figure 2.20 shows `Stationery`'s contents. A single tuple $t = \langle \texttt{Target}, \texttt{pencil} \rangle$ in `Stationery` has two

[8] Although called a *set*, a derivation set may well have duplicates; that is, two derivations of t may have the same value.

s_name	i_name
Target	binder
Target	pencil
Target	binder
Target	pencil

Figure 2.20: `Stationery` contents

D^*_1

store

s_id	s_name	city	state
001	Target	Palo Alto	CA

item

i_id	i_name	category
0002	pencil	stationery

sales

s_id	i_id	price	num
001	0002	1	3000

D^*_2

store

s_id	s_name	city	state
002	Target	Albany	NY

item

i_id	i_name	category
0002	pencil	stationery

sales

s_id	i_id	price	num
002	0002	2	2000

Figure 2.21: Multiple derivations of $\langle$`Target, pencil`$\rangle$

store

s_id	s_name	city	state
001	Target	Palo Alto	CA
002	Target	Albany	NY

item

i_id	i_name	category
0002	pencil	stationery

sales

s_id	i_id	price	num
001	0002	1	3000
002	0002	2	2000

Figure 2.22: Derivation pool for $\langle$`Target, pencil`$\rangle$

derivations, D^*_1 and D^*_2, as shown in Figure 2.21, so t's derivation set is $\{D^*_1, D^*_2\}$. Figure 2.22 shows t's derivation pool. □

We are now considering two modes of derivation tracing: tracing individual derivations, and tracing derivation pools. We can prove that the property of view and derivation equivalence after our canonicalizing transformations (Theorem 2.6.2) still holds for bag semantics, derivation sets, and derivation pools. Thus, we can still transform a view into canonical form before tracing derivation sets or pools. The transitive property (Theorem 2.3.8) also applies to derivation sets and pools under bag semantics, which allows us to trace derivations down the view definition tree recursively. In addition, we prove the following theorems, which provide the groundwork for our later tracing algorithms.

Theorem 2.7.5 (Derivation Uniqueness for Unique View Tuples) Let v be a general view over database D with bag semantics and no difference operators. If a tuple $t \in v(D)$ has no duplicates, then t has a unique derivation in v. As a result, in this case ${v^{-1}}_D(t) = {v^{-P}}_D(t)$. □

Proof: According to Definition 2.7.1, given a tuple t in view $v(D)$, a derivation of t is a group of base tuples that can produce t by themselves. Given a monotonic view, each of t's derivations derives a tuple with value t in the view table. These derived tuples cannot be removed by other base tuples due to monotonicity. Therefore, if tuple $t \in v(D)$ has no duplicates, it has a unique derivation ${v^{-1}}_D(t)$. (Otherwise, if t has multiple derivations, there must be multiple tuples in $v(D)$ with the same value as t. This conflicts with the fact that t has no duplicates in $v(D)$.) Furthermore, t's derivation pool contains exactly all tuples in t's unique derivation; in other words, ${v^{-P}}_D(t) = {v^{-1}}_D(t)$. □

Theorem 2.7.6 (Derivation Pool of a Set of Same-Valued Tuples) Let v be any general view over database D. If all tuples in a tuple set $T \subseteq v(D)$ have the same value t, then ${v^{-P}}_D(T) = {v^{-P}}_D(t)$. This also implies that ${v^{-P}}_D(\sigma_{\mathbf{V}=t}(v(D))) = {v^{-P}}_D(t)$. □

Proof: $\forall t^* \in {v^{-P}}_D(T)$: $\exists t' \in T$ such that $t^* \in {v^{-P}}_D(t')$. Because T contains only tuples with value t, $t' = t$. So, $t^* \in {v^{-P}}_D(t)$. Therefore, ${v^{-P}}_D(T) \subseteq {v^{-P}}_D(t)$. Also, since $t \in T$, ${v^{-P}}_D(T) \supseteq {v^{-P}}_D(t)$. Therefore, ${v^{-P}}_D(T) = {v^{-P}}_D(t)$. □

Theorem 2.7.7 (Derivation Pool of Selected Portion in View Table) Let v be any general view over database D. To trace the derivation pool of a selected portion $\sigma_C(v(D))$ of the view table, we can define another view $v' = \sigma_C(v)$, and trace the derivation of the entire view table $v'(D)$ according to v'. In other words:

$$ {v^{-P}}_D(\sigma_C(v(D))) = {v'^{-P}}_D(v'(D)) = {v'^{-P}}_D(\mathbf{ALL}) $$

where $v' = \sigma_C(v)$. □

Proof: Let $v' = \sigma_C(v)$, ${v'^{-1}}_D(v'(D)) = {v^{-1}}_D({\sigma_C}^{-1}{}_{\langle V \rangle}(\sigma_C(v(D)))) = {v^{-1}}_D(V \ltimes \sigma_C(v(D))) = {v^{-1}}_D(\sigma_C(v(D)))$. □

In Sections 2.7.2, 2.7.3, and 2.7.4, we develop techniques for: (1) tracing derivation pools, (2) tracing derivation sets, and (3) associating a unique derivation with each view tuple using existing or system-generated base table keys.

2.7.2 Tracing Derivation Pools

Tracing derivation pools for views with bag semantics is similar to tracing derivations for views with set semantics as specified in Sections 2.5 and 2.6. We begin by specifying the derivation pools for all of the relational operators, using bag semantics.

Theorem 2.7.8 (Derivation Pools for Operators with Bag Semantics) Let T and $T_1, \ldots, T_m$ be tables. Recall that $\mathbf{T_i}$ denotes the schema of T_i. The derivation pools Op^{-P} for σ, π, $\bowtie$, and α are the same as their derivations Op^{-1} as specified in Theorem 2.3.4, except that $\{t\}$ is replaced by $\sigma_{\mathbf{T}=t}(T)$, and $t.\mathbf{T}_i$ is replaced by $\sigma_{\mathbf{T_i}=t.\mathbf{T_i}}(T_i)$.

$$\uplus^{-P}{}_{\langle T_1,\ldots,T_m\rangle}(t) = \langle \sigma_{\mathbf{T_1}=t}(T_1), \ldots, \sigma_{\mathbf{T_m}=t}(T_m)\rangle, \text{ for } t \in T_1 \uplus \cdots \uplus T_m$$

$$\dot{-}^{-P}{}_{\langle T_1,T_2\rangle}(t) = \langle \sigma_{\mathbf{T_1}=t}(T_1), \sigma_{\mathbf{T_2}\neq t}(T_2)\rangle, \text{ for } t \in T_1 \dot{-} T_2$$

. □

Proof: We prove the theorem for $\bowtie$ and $\dot{-}$. Others can be proved similarly.

- $\bowtie^{-1}{}_{\langle T_1,\ldots,T_m\rangle}(t) = \langle \sigma_{\mathbf{T_1}=t.\mathbf{T_1}}(T_1), \ldots, \sigma_{\mathbf{T_m}=t.\mathbf{T_m}}(T_m)\rangle$, for $t \in T_1 \bowtie \cdots \bowtie T_m$.

  D^* is a derivation of $t \in T_1 \bowtie \cdots \bowtie T_m$ iff $D^* = \langle \{t.\mathbf{T_1}\}, \ldots, \{t.\mathbf{T_m}\}\rangle$, since D^* satisfies Theorem 2.3.4. Therefore, in T_i, only tuples with value $t.\mathbf{T_i}$ contributes to $t \in T_1 \bowtie \cdots \bowtie T_m$, for $i = 1..m$. According to the definition of derivation pool, $\bowtie^{-1}{}_{T_i}(t)$ contains all tuples that contribute to t. Therefore, $\bowtie^{-1}{}_{T_i}(t) = \sigma_{\mathbf{T_i}=t}(T_i)$, for $i = 1..m$.

- $\dot{-}^{-1}{}_{\langle T_1,T_2\rangle}(t) = \langle \sigma_{\mathbf{T_1}=t}(T_1), \sigma_{\mathbf{T_2}\neq t}(T_2)\rangle$, for $t \in T_1 - T_2$.

  D^* is a derivation of $t \in T_1 - T_2$ iff $D^* = \langle \{t\}, \sigma_{T_2\neq t}(T_2)\rangle$, since D^* satisfies Theorem 2.3.4. Therefore, tuples in T_1 with value t and tuples in T_2 with value different from t contribute to $t \in T_1 - T_2$. Therefore, $\dot{-}^{-1}{}_{\langle T_1,T_2\rangle}(t) = \langle \sigma_{\mathbf{T_1}=t}(T_1), \sigma_{\mathbf{T_2}\neq t}(T_2)\rangle$, for $t \in T_1 - T_2$. □

procedure TableDerivation(T, v, D)
in: a tracing tuple set T, a view definition v, and a database D
out: T's derivation pool in D according to v
1 **case** $v = R \in D$:
2 **if** $T = \mathbf{ALL}$ **then return** ($\langle R \rangle$);
3 **else return** ($\underline{\langle R \ltimes T \rangle}$);
4 **case** $v = v'(v_1, \ldots, v_k)$:
5 // v' is a one-level AUSPJ view
6 // $V_j = v_j(D)$ is an intermediate view or a base table, $j = 1..k$
7 $\langle V_1^*, \ldots, V_k^* \rangle \leftarrow \underline{\text{TQ}}(T, v', \{V_1, \ldots, V_k\})$;
8 **return** (TableListDeriv($\langle V_1^*, \ldots, V_k^* \rangle, \{v_1, \ldots, v_k\}, D$));
9 **case** $v = v_1 \dot{-} v_2$:
10 // $V = v(D)$
11 // $V_j = v_j(D)$ is an intermediate view or a base table, $j = 1, 2$
12 **if** $T = \mathbf{ALL}$ **then** $T \leftarrow V$;
13 **if** T contains only tuples with value t
14 **then return** (TableListDeriv($\langle T, \mathbf{ALL} \rangle, \langle v_1, \sigma_{\mathbf{V_2} \neq t}(v_2) \rangle, D$));
15 **else return** (TableListDeriv($\langle T, \mathbf{ALL} \rangle, \langle v_1, v_2 \rangle, D$));

Figure 2.23: Derivation pool tracing algorithm

The derivation pool tracing query for one-level AUSPJ views with bag semantics is obtained by replacing the *Split* operator in Theorem 2.6.4 with a *targeted split (TSplit)* operator.

Definition 2.7.9 (*TSplit* Operator) Let T be a table with schema $\mathbf{T}$, and let T_i be a table with schema $\mathbf{T_i} \subseteq \mathbf{T}$, $i = 1..m$. We define the *targeted split of* T *by* $T_1, \ldots, T_m$ to be:

. $$\mathit{TSplit}_{T_1,\ldots,T_m}(T) = \langle T_1 \ltimes T, \ldots, T_m \ltimes T \rangle \qquad \square$$

Note that we need to use semijoins with the data in $T_1, \ldots, T_n$ instead of simple projections based on schema (which we used in the original *Split* operator), because semijoin is the only way to produce the correct number of duplicates in the derivation result. However, since *TSplit* is less efficient than *Split*, we recommend using the original tracing query when the base tables are known to have no duplicates.

The general derivation pool tracing procedure is obtained by making only small modifications to procedure `TableDerivation` from Figure 2.19, shown as the underlined portions in Figure 2.23. We replace T with $R \ltimes T$ in the return value for the case $v = R$

to retrieve all tuples in R that are also in T, including duplicates. The tracing query `TQ` is now using *TSplit* in Definition 2.7.9 instead of using *Split* from Theorem 2.6.4. Finally, for D-segments ($v = v_1 \dot{-} v_2$), we separate the case where T contains only tuples with the same value t. When all tuples have the same value, the recursive procedure call directly follows the derivation pool equation in Theorem 2.7.8. However, if two tuples in T have distinct values, t_1 and t_2 say, then we still need to include any copies of t_1 that might appear in v_2 (or derivations of such tuples), since any t_1's in v_2 are part of t_2's derivation pool in v. A similar argument holds with t_1 and t_2 reversed.

2.7.3 Tracing Derivation Sets

Unlike the derivation pool for a tuple t, t's derivation set separates its distinct derivations. In this section we present a *derivation enumeration* technique that produces t's derivations one by one, thus generating t's entire derivation set.

Procedure `DerivationEnum(`t, v, D`)` in Figure 2.24 traces the derivation set of tuple t according to view v. We assume that v is already in canonical form (Section 2.6.2). However, when recursively tracing down the view definition tree, we treat AUSPJ segments with an omitted (i.e., trivial) aggregation operator separately. We also separate the bag union node from the SPJ subtree in these segments for presentation clarity. Thus, each (intermediate) view v we trace through during the derivation enumeration procedure has one of the following five forms: (1) $v = R$; (2) $v = \pi_A(\sigma_C(v_1 \bowtie \cdots \bowtie v_k))$; (3) $v = v_1 \uplus \cdots \uplus v_k$; (4) $v = v'(v_1, \ldots, v_k)$; (5) $v = v_1 \dot{-} v_2$, where R is a base table in database D, v_i is an intermediate view or a base table, $i = 1..k$, and v' is an AUSPJ segment with a nontrivial aggregation node. These cases map to the five cases in Figure 2.24, in order.

For case (1), according to Definition 2.3.5 each copy of t in R forms a derivation $\{t\}$ of t in V. For case (2), we first compute t's derivations in $v_1(D), \ldots, v_k(D)$ based on Theorem 2.7.10:

Theorem 2.7.10 (Derivation Set for an SPJ View) Given SPJ view $V = v(T_1, \ldots, T_m) = \pi_A(\sigma_C(T_1 \bowtie \cdots \bowtie T_m))$ and tuple $t \in V$, t's derivation set according to v is

$$v^{-S}{}_{T_1,\ldots,T_m}(t) = \{\langle\{t'.\mathbf{T_1}\}, \ldots, \{t'.\mathbf{T_m}\}\rangle \mid t' \in \sigma_{C \wedge A=t}(T_1 \bowtie \cdots \bowtie T_m)\}. \qquad \square$$

procedure DerivationEnum(t, v, D)
in: a tracing tuple t, a view definition v, and a database D
out: the set of all t's derivations in D according to v

$DS \leftarrow \varnothing$;
case $v = R \in D$:
  for each tuple in $\sigma_{\mathbf{R}=t}(R)$ **do** insert $\langle\{t\}\rangle$ into DS;
case $v = \pi_A(\sigma_C(v_1 \bowtie \cdots \bowtie v_k))$:
  $Pending \leftarrow \sigma_{C \wedge A=t}(v_1(D) \bowtie \cdots \bowtie v_k(D))$;
  while $Pending \neq \varnothing$ **do**
  begin
    pick a tuple t' in $Pending$;
    for $j \leftarrow 1$ to k **do** $DS_j \leftarrow$ DerivationEnum($t'.\mathbf{V_j}, v_j, D$);
    insert all elements of $DS_1 \times \cdots \times DS_k$ into DS;
    remove t' and all its duplicates from $Pending$;
  end
case $v = v_1 \uplus \cdots \uplus v_k$:
  for $j \leftarrow 1$ to k **do** insert all elements of DerivationEnum(t, v_j, D) into DS;
case $v = v'(v_1, \ldots v_k)$: // v' is an AUSPJ view with aggregation:
  $\langle V_1^*, \ldots, V_k^* \rangle =$ TQ($\{t\}, v', \langle V_1, \ldots, V_k \rangle$);
  for $j \leftarrow 1$ to k **do** $DS_j \leftarrow$ TableDerivEnum(V_j^*, v_j, D);
  $DS \leftarrow DS_1 \times \cdots \times DS_k$;
case $v = v_1 \dot{-} v_2$:
  $DS \leftarrow$ DerivationEnum(t, v_1, D) $\times$ TableDerivEnum($\mathbf{ALL}, \sigma_{\mathbf{V_2} \neq t}(v_2), D$);
return (DS);

procedure TableDerivEnum(T, v, D)
in: a tracing tuple set T, a view definition v, and a database D
out: the set of all T's derivations in D according to v

let $T = \{t_1, \ldots, t_n\}$;
for $i \leftarrow 1$ to n **do** $DS_i \leftarrow$ DerivationEnum(t_i, v, D);
$DS \leftarrow \varnothing$;
for each element $D_1, \ldots, D_n$ in $DS_1 \times \cdots \times DS_n$ **do**
  if $\forall i \neq j : D_i \neq D_j$ **then** insert $D_1 \cup \cdots \cup D_n$ into DS;
return (DS);

Figure 2.24: Derivation enumeration algorithm

Proof:

D^* is a derivation of $t \in V$ according to v

$\Leftrightarrow$ $\exists T' \subseteq V' = \sigma_C(R_1 \bowtie \ldots R_m)$ such that T' is a derivation of t according to $v_1 = \pi_A(V')$, and D^* is a derivation of T' according to v'

$\Leftrightarrow$ $\exists t' \in \sigma_{A=t \wedge C}(R_1 \bowtie \ldots R_m)$ such that $T' = \{t'\}$

$\Leftrightarrow$ $\exists T'' \subseteq V'' = R_1 \bowtie \ldots R_m$ such that T'' is a derivation of $\{t'\}$ according to $v_2 = \sigma_C(V'')$, and D^* is a derivation of T'' according to v''

$\Leftrightarrow$ $T'' = t'$, and $D^* = \langle \{t'.\mathbf{T_1}\}, \ldots, \{t'.\mathbf{T_m}\} \rangle$

$\cdot \Leftrightarrow$ $D^* \in DS_v(t)$ □

We initialize a $Pending$ set to be $\sigma_{C \wedge A=t}(v_1(D) \bowtie \cdots \bowtie v_k(D))$. According to Theorem 2.7.10, $\forall t' \in Pending$: $D^* = \langle \{t'.\mathbf{V_1}\}, \ldots, \{t'.\mathbf{V_k}\} \rangle$ forms a derivation of t in $v_1(D), \ldots, v_k(D)$. We then further trace the derivation set DS_j of $t'.\mathbf{V_j}$ according to v_j, for $j = 1..k$. Based on Definition 2.3.5, we then append the cross-product of the DS_j's to t's derivation set, where $DS_1 \times DS_2 = \{D_1^* \circ D_2^* \mid D_1^* \in DS_1,\ D_2^* \in DS_2\}$. We remove t' and its duplicates from the $Pending$ set, and repeat the above process until the $Pending$ set becomes empty.

For case (3), where v is a bag union of $v_1, \ldots, v_k$, each t in any $v_i(D)$ is a derivation of $t \in v(D)$ according to the bag union operator, so each derivation of t according to v_i forms a derivation of t according to v. Therefore, we simply enumerate through t's derivations according to each v_i, adding them to the derivation set.

Consider case (4), where v is defined by an AUSPJ segment v' over $v_1, \ldots, v_k$ with a nontrivial aggregation operator. Since v has no duplicates due to the aggregation, by Theorem 2.7.5 t has a unique derivation in $V_1, \ldots, V_k$ according to v'. This derivation $\langle V_1^*, \ldots, V_k^* \rangle$ is obtained by tracing t in $V_1, \ldots, V_k$ according to v'. Then we trace the derivation set DS_j for each V_j^* according to v_j. (More on this step below.) The cross-product of the DS_j's forms the derivation set of t according to v.

For case (5), where v is a bag difference of v_1 and v_2, each derivation of t according to v_1 and each derivation of $T_2 = \sigma_{\mathbf{V_2} \neq t}(v_2(D))$ according to v_2 together form a derivation of t according to v. Thus, t's derivation set is the cross-product of t's derivation set according to v_1 and T_2's derivation set according to v_2.

Procedure `TableDerivEnum(`T, v, D`)` in Figure 2.24 is called to trace the derivation

set of a tuple set T according to a view v. Recall from Definition 2.7.1 that derivations for a tuple set T are constructed from all possible combinations of derivations for the tuples in T such that the derivations selected for any $t_1 = t_2 \in T$ are different. Let T have n tuples. We first trace the derivation set DS_i for each tuple $t_i \in T$, $i = 1..n$. Then, for every combination of distinct derivations $D_1, \ldots, D_n$ from $DS_1, \ldots, DS_n$, we add $D_1 \cup \cdots \cup D_n$ to the derivation set for T. Note that this procedure can be extremely expensive if T is large. While calling `TableDerivEnum` is necessary in the general case, in many cases we can replace it with the much cheaper `TableDerivation` (Figure 2.23). In case (4), if v does not contain any difference operators, then by Theorem 2.7.5 t has a unique derivation in D according to v. Hence, we can use t's derivation pool obtained by calling `TableDerivation`, since in this case the derivation pool is the same as the derivation set. Likewise, in case (5), if v_2 does not contain any difference operators, we can replace `TableDerivEnum` with `TableDerivation` to obtain the single derivation for **ALL**.

2.7.4 Using Key Information

We now propose an alternative approach to tracing view tuple derivations in the presence of duplicates for multilevel ASPJ views. Suppose for now that view v's base tables all have keys, i.e., case (3) in Figure 2.1. We can extend v's definition using the key information to obtain a *supporting view* v' such that v', as well as all of its intermediate results, has no duplicates. Then, after mapping each tuple $t \in v(D)$ to a distinct tuple $t' \in v'(D)$, we can retrieve a unique derivation for t by tracing the derivation of t' according to v' using the tracing algorithm for set semantics. In other words, we use base table keys to assign a unique derivation to each copy of each view tuple. If we are not interested in individual unique derivations, this technique is still useful for tracing derivation sets and pools as in previous sections in a more efficient manner.

Theorem 2.7.11 (SPJ View Definition with Keys) Let D be a database with tables $R_1, \ldots, R_m$, and let $V = v(D) = \pi_A(\sigma_C(R_1 \bowtie \cdots \bowtie R_m))$ be an SPJ view over D. Suppose each R_i has a set of attributes K_i that are a key for R_i, $i = 1..m$. Then we can define a view $V' = v'(D) = \pi_{A \cup K_1 \cup \cdots \cup K_m}(\sigma_C(R_1 \bowtie \cdots \bowtie R_m))$ that contains no duplicates. If V contains n copies of a tuple t, then there are n tuples $t_1, \ldots, t_n$ in V' such

s_name	i_name	s_id	i_id
Target	binder	001	0001
Target	pencil	001	0002
Target	binder	002	0001
Target	pencil	002	0002

Figure 2.25: `ExtStationery` contents

that $t_j.A = t$, $j = 1..n$. The derivation of each t_j according to v' is also a derivation of t according to v, and ${v^{-S}}_D(t) = \{{v'^{-1}}_D(t_j),\ j = 1..n\}$. □

Proof: We first prove that V' contains no duplicates. Suppose there exist two tuples $t = t' \in V'$, then $t.K_i = t'.K_i$ for $i = 1..m$. Since K_i is the key of table R_i, there exists a unique tuple $t_i \in R_i$ such that $t_i.K_i = t.K_i = t'.K_i$. According to Definition 2.7.1, we know that t has a unique derivation $D^* = \langle\{\langle t_1\rangle\}, \ldots, \{\langle t_m\rangle\}\rangle$. So has t'. Both t and t' are derived uniquely from D^*, which conflicts with the fact that $v'(D^*)$ contains a single tuple.

According to Theorem 2.3.11, we can rewrite v as $v = \pi_A(v')$ while not affecting its tuple derivations. Given n copies of a tuple t in V, there are n derivations of t in V' according to π_A: $\langle\{\langle t_j\rangle\}\rangle$ such that $t_j.A = t$, for $j = 1..n$. Since V' contains no duplicates, each t_j has a unique derivation ${v'^{-1}}_D(t_j)$ in D according to v'. According to Theorem 2.3.8, ${v'^{-1}}_D(t_j)$ is a derivation of t according to v, and $\{{v'^{-1}}_D(t_j) \mid j = 1..n\}$ is the set of all derivations (the derivation set) of t according to v. □

According to Theorem 2.7.11, we can map each copy of a tuple $t \in V$ to a distinct tuple $t_j \in V'$ such that $t_j.A = t$. Then we can associate a unique derivation with each copy of t by tracing the corresponding t_j's derivation. We can also retrieve the entire derivation set (or pool) for t by tracing the derivations for all the t_j's. Recall from Theorem 2.4.5 that derivation tracing for views that include base table keys (e.g., view v') takes time at most linear in the size of $R_1, \ldots, R_m$, which is generally much more efficient than the derivation enumeration algorithm in Section 2.7.3.

Example 2.7.12 (SPJ View Extended with Keys) Consider again the problem of tracing the derivation of $\langle$`Target,pencil`$\rangle \in$ `Stationery` in Example 2.7.4. Suppose table `store` has key `s_id`, `item` has key `i_id`, and `sales` has key $\langle$`s_id`, `i_id`$\rangle$. We first define a view `ExtStationery` $= \pi_{\texttt{s\_name,i\_name,s\_id,i\_id}}(\sigma_{\texttt{category=“stationery”}}($

store

s_id	s_name	city	state
001	Target	Palo Alto	CA

item

i_id	i_name	category
0002	pencil	stationery

sales

s_id	i_id	price	num
001	0002	1	3000

Figure 2.26: Derivation of t'

store

s_id	s_name	city	state
002	Target	Albany	NY

item

i_id	i_name	category
0002	pencil	stationery

sales

s_id	i_id	price	num
002	0002	2	2000

Figure 2.27: Derivation of t''

$\texttt{store} \bowtie \texttt{item} \bowtie \texttt{sales}$)). Figure 2.25 shows the view contents. Let us map the traced tuple $t = \langle \texttt{Target,pencil} \rangle$ in `Stationery` to $t' = \langle \texttt{Target,pencil,001,0002} \rangle$ in `ExtStationery`. We retrieve t''s derivation according to `ExtStationery` and obtain the result in Figure 2.26. We can similarly map another copy of $\langle \texttt{Target,pencil} \rangle$ in `Stationery` to $t'' = \langle \texttt{Target,pencil,002,0002} \rangle$ in `ExtStationery`, and retrieve the second derivation, shown in Figure 2.27. Notice that Figures 2.26 and 2.27 are identical to the two derivations in Figures 2.21, as desired. □

Given a multilevel ASPJ view $V = v(D)$, if v's top segment contains an aggregation operator, then V has no duplicates, and each tuple in V has a unique derivation (Theorem 2.7.5). If v's top segment contains no aggregation operator, i.e., $V = \pi_A(\sigma_C(T_1 \bowtie \cdots \bowtie T_m))$ where T_i is a base table or an intermediate aggregation view, we can first rewrite the top segment to include T_i's key attributes, as in Theorem 2.7.11. (For the cases where T_i is an intermediate aggregation view, its key is the grouping attributes. Otherwise we use base table keys.) Then, each tuple in the extended view has a unique derivation which can be traced using the recursive algorithm in Figure 2.14.

For base tables without keys, we can still apply the techniques introduced in this section by first attaching a system-generated key (e.g., the tuple ID or a *surrogate*) to each base tuple. We then extend the view definition to include these keys as described earlier. After tracing tuple derivations for the extended view, we project out the system-generated key attributes from the derivation results to obtain the correct derivation according to the

original view.

Extending the techniques of this section to general views including the bag operators union and difference is somewhat complex, due in part to the requirement of uniform schemas in operands, and is beyond the scope of this thesis.

2.7.5 Algorithms Summary

We have presented four major derivation tracing algorithms:

1. The *basic tracing algorithm* (Sections 2.4–2.6) traces view tuple derivations under set semantics for general views.

2. The *pool tracing algorithm* (Section 2.7.2) traces view tuple derivation pools under bag semantics for general views. It retrieves all contributing tuples for the traced tuple without distinguishing individual derivations. The algorithm itself is similar to the basic tracing algorithm, although it incurs some overhead for handling duplicates in base tables and intermediate views during the tracing process.

3. The *enumerating algorithm* (Section 2.7.3) traces view tuple derivation sets under bag semantics for general views. It lists each derivation of the traced tuple separately. This algorithm is the most expensive of the four algorithms we propose.

4. The *key-based algorithm* (Section 2.7.4) traces view tuple derivations in the presence of duplicates in the view by using key information to associate a unique derivation with each view tuple. In general, this algorithm is more efficient than the other three algorithms, but it can only be used easily for multilevel ASPJ views (not general views with union and difference operators) and may require additional machinery to generate tuple IDs.

2.8 Revisiting Introductory Example

For continuity, let us revisit our original motivating example from Chapter 1, presented in Section 1.2. Recall that we considered a university data warehouse constructed from four

source tables, Program, Student, Course, and Enrollment, depicted with sample data in Figure 1.2. From these sources, we defined a materialized view IncomeByField. The view was not specified formally in Section 1.2, but it can be written in relational algebra as: $\alpha_{\texttt{field},\texttt{sum(unit\_tuition*units)}}(\texttt{Program} \bowtie \texttt{Student} \bowtie \texttt{Course} \bowtie \texttt{Enrollment})$. The view and its contents for our sample source data are depicted in Figure 1.3.

Assuming set semantics, we are dealing with a one-level ASPJ view as covered in Section 2.5 of this chapter. Thus, our general tracing query for a tuple t is: $TQ_{t,v} = Split_{\mathbf{Program},\mathbf{Student},\mathbf{Course},\mathbf{Enrollment}}(\sigma_{\texttt{field}=t.\texttt{field}}(\texttt{Program} \bowtie \texttt{Student} \bowtie \texttt{Course} \bowtie \texttt{Enrollment}))$. When tracing the lineage of example view tuple $\langle \texttt{Databases}, \$84\texttt{K} \rangle$, the instantiated tracing query is $Split_{\mathbf{Program},\mathbf{Student},\mathbf{Course},\mathbf{Enrollment}}(\sigma_{\texttt{field}='\texttt{Databases}'}(\texttt{Program} \bowtie \texttt{Student} \bowtie \texttt{Course} \bowtie \texttt{Enrollment}))$. The result of this query over the sample data is exactly the lineage result provided for the original example in Figure 1.4.

2.9 Related Work

Most previous work related to this chapter was covered in the general related work section of Chapter 1 (Section 1.5). One reference, [KLM+97], also introduces *derivation sets*, in the context of concurrency control for materialized views. Their definition of a derivation set and our lineage definition are related, in the sense that both definitions attempt to isolate the base tuples that affect specific view tuples. However, [KLM+97] defines derivation sets in a database-independent way, so their definition is much looser than ours. Furthermore, [KLM+97] considers a more restricted class of views than we consider. The two approaches were compared in more detail in Section 2.5.2.

2.10 Chapter Summary

In this chapter, we formalized the data lineage problem for warehouses defined as relational views over relational sources. We presented lineage tracing algorithms for a very general class of views under both set and bag semantics. Using our algorithms, the system converts the view definition into a segmented normal form when the view is defined. At tracing time, the system recursively traces data lineage through each segment in the view definition,

usually using one tracing query per segment, until reaching the source tables. Our tracing queries are generated automatically based on the view definition and are parameterized by the traced tuples, and they can be optimized easily by a conventional database management system (DBMS). Implementation of several of the algorithms in this chapter in the context of the WHIPS data warehousing prototype will be described in Section 3.8 of the next chapter.

Chapter 3

Storing Auxiliary Views for Efficient Lineage Tracing

In this chapter, we introduce techniques for storing *auxiliary views* to improve the performance of lineage tracing in relational data warehouses. Using our approach presented in Chapter 2, to compute the lineage of a warehouse data item, we need the definition of the view containing the item, the original source data, and depending on the view's complexity possibly some auxiliary information representing intermediate results in the view definition (recall Section 2.5). In a distributed multi-source data warehousing environment, querying the sources during lineage tracing may be difficult or impossible: sources may periodically be inaccessible, they may be expensive to access, expensive or slow to transfer data from, and/or inconsistent with the views at the warehouse. By storing additional auxiliary information (auxiliary views) in the warehouse, we can reduce or entirely avoid querying the sources. Furthermore, auxiliary views may reduce or eliminate the cost of recomputing intermediate results for lineage tracing. However, computing and maintaining auxiliary views may significantly increase warehouse maintenance cost.

There are numerous options for which auxiliary information to store, with significant performance tradeoffs. For Select-Project-Join (SPJ) views, we propose a variety of auxiliary view schemes and study overall warehouse tracing and maintenance performance for the different schemes through simulations. For arbitrary views with aggregation (the ASPJ

views introduced in Section 2.2), we suggest an initial search space of possible beneficial auxiliary views based on the *ASPJ view normal form* from Section 2.5.1, we propose several algorithms for selecting auxiliary views within this search space, and we compare empirically the running time of our selection algorithms and the optimality of the auxiliary view sets they select. We use the TPC-D benchmark [TPC96], as well as a suite of synthetic view definitions and statistics.

Section 3.1 introduces several running examples and motivates the technique of storing auxiliary views for improved lineage tracing performance. Section 3.2 introduces a variety of auxiliary view schemes for the restricted case of SPJ views. Section 3.3 presents a performance study of the proposed auxiliary view schemes, considering lineage tracing cost as well as overall warehouse view maintenance and storage costs. Section 3.4 specifies the auxiliary view selection problem for ASPJ views, and Section 3.5 describes the detailed cost model we use to estimate lineage tracing and view maintenance costs for a given ASPJ view and a set of auxiliary views for it. Section 3.6 then presents our auxiliary view selection algorithms based on the cost model, and Section 3.7 compares their performance using experiments. Section 3.8 introduces a lineage tracing system prototype we implemented based on the results presented thus far in the thesis. Finally, Section 3.9 surveys related work, and Section 3.10 concludes the chapter. The work presented in this chapter appeared originally in [CW00a, CW00b, CW00c].

3.1 Running Examples

For this chapter's running examples, let us again consider the retail data warehouse from Section 2.1, where views `Calif` (Figure 2.4) and `Clothing` (Figure 2.7) are defined over three source tables `store`, `item`, and `sales` (Figures 2.1–2.3). Again, suppose we select a tuple in `Calif` (or `Clothing`) and want to trace its data lineage. Recall that we used the terms *derivation set* and *derivation pool* in Chapter 2 to formalize the concept of data lineage under set and bag semantics, respectively. In this chapter and all later chapters, we consider only set semantics, where there is always a unique derivation for any given view data item (Theorem 2.3.2). Thus, we use the term *lineage* generically throughout the rest of the thesis to indicate the unique *derivations* of Theorem 2.3.2.

To trace the lineage of view tuple $t = \langle \texttt{Target}, \texttt{pencil}, 3000 \rangle$ in `Calif`, using our tracing algorithms presented in Section 2.4, we can apply either of the tracing queries in Figure 2.12 to the source tables. However, as mentioned at the outset of this chapter, querying the source tables during lineage tracing may be inefficient, difficult, or impossible. To reduce or avoid source accesses during lineage tracing, we may choose to store at the warehouse *auxiliary materialized views* over the source tables. For the simple view `Calif`, we could store a copy of all source tables in the warehouse and apply the tracing query directly to the local copies to avoid remote source queries altogether during lineage tracing. There are several alternative auxiliary views we could store to reduce source accesses during lineage tracing, which we will introduce later in Section 3.2.

Recall that for lineage tracing in a complex ASPJ view like `Clothing` in Figure 2.7, we first divide the view definition tree into two ASPJ segments and define an intermediate view `AllClothing` over the bottom segment, as shown in Figure 2.13. As discussed in Section 2.5.3, the contents of the intermediate view `AllClothing` are needed in the tracing query for the top segment of view `Clothing`. We can recompute the relevant portion of the aggregate when tracing a tuple's lineage, or we can materialize `AllClothing` as an auxiliary view at the warehouse specifically for lineage tracing.

All materialized views at the warehouse, including the *primary view* being traced and any auxiliary views to support lineage tracing, must be *maintained* when the sources change, as discussed in Section 1.1. Throughout this chapter, we assume a standard *incremental view maintenance* approach is used, e.g., [GMS93, ZGMHW95]. In this approach, changes to each source table are recorded in a *delta table*. During view maintenance, changes to the view are computed using a predefined query called a *maintenance expression*. In our example, when tuples are inserted into `sales`, the insertions are recorded in a delta table `sales-ins`. Insertions to the view `Calif` can then be computed using a query (maintenance expression) that is the same as the view definition in Figure 2.4, except that `sales` is replaced by `sales-ins`. Deletions to the view are computed similarly. We then *refresh* the view table by applying the changes, and the view becomes up-to-date. To evaluate the maintenance expressions, it is usually necessary to query the source tables, which can be problematic as discussed earlier. Prior work has addressed this problem by adding auxiliary views to ensure *self-maintainability*: a set of views is self-maintainable if

it can be maintained using only the source changes and the view data, without querying the sources [QGMW96].

As we introduce auxiliary views in the warehouse to support lineage tracing, overall warehouse maintenance cost may increase since more views need to be maintained. However, the same auxiliary views that help lineage tracing often can help maintain the primary view, sometimes even making the entire set of views self-maintainable. For example, the auxiliary view `AllClothing` that we defined for lineage tracing can also help the maintenance of primary view `Clothing`, because we can avoid recomputing the intermediate aggregation results during each view maintenance process.

To take the interaction of auxiliary views for lineage tracing and view maintenance into consideration, we decided to study auxiliary view schemes for lineage tracing and view maintenance together. We also consider warehouse storage cost, because some auxiliary view schemes that can improve both lineage tracing and view maintenance performance may incur very hight storage space requirements. (Storing an entire copy of all source data might be such an example.)

In general, given an arbitrary ASPJ view, finding a set of auxiliary views that minimizes the overall lineage tracing and view maintenance cost is a hard optimization problem. To illustrate the general auxiliary view selection problem, we will use a somewhat more complicated example. First, we extend our source tables by adding an extra attribute `cost` to the source table `item`, and we add a fourth source table `employee`(`e_id`, `s_id`, `salary`). The complete source schema is now:

- `store`(`s_id`, `s_name`, `city`, `state`), as earlier
- `item`(`i_id`, `i_name`, `category`, `cost`), which includes the wholesale cost of each item
- `sales`(`s_id`, `i_id`, `price`, `num`), where `num` now represents the expected number sold each month
- `employee`(`e_id`, `s_id`, `salary`), which includes the monthly salary of each employee at each store

```
CREATE VIEW HighProfit AS
SELECT s_name
FROM store as S,
  (SELECT s_id, SUM(num*(price−cost)) AS profit
   FROM sales, item
   WHERE sales.i_id = item.i_id
   GROUP BY s_id) AS P,
  (SELECT s_id, SUM(salary) AS salaries
   FROM employee
   GROUP BY s_id) AS E
WHERE S.s_id = E.s_id
  AND E.s_id = P.s_id
  AND P.profit−E.salaries > 100000
```

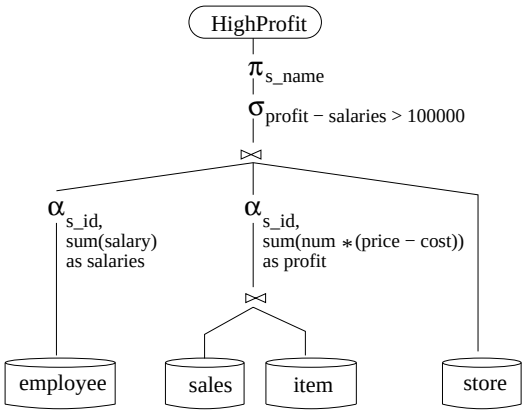


Figure 3.1: View definition for `HighProfit`

Consider a materialized view `HighProfit` that keeps track of stores with high profits, i.e., stores whose expected monthly sales income exceeds its total salary expenses by at least $100,000. An SQL definition for `HighProfit` and its view definition tree in segmented ASPJ normal form (Section 2.5) are shown in Figure 3.1.

For view `HighProfit`, there are numerous ways of storing auxiliary views to help lineage tracing or view maintenance. For example, we could store the result of the two lower-level aggregations, we could store the join result, we could store copies of most or all of the source tables, etc. In Sections 3.4–3.6, we will introduce techniques for selecting a good set of auxiliary views without exploring the entire combinatorial space.

3.2 Auxiliary Views for Tracing SPJ Views

In this section, we introduce a variety of auxiliary view schemes for the restricted case of SPJ views, and we study their relative performance in Section 3.3. In Sections 3.4–3.7, we will extend our auxiliary view schemes to consider arbitrary multilevel ASPJ views.

Given an SPJ view $V = v(D) = \pi_A(\sigma_C(T_1 \bowtie \cdots \bowtie T_m))$ and a tuple set $T \subseteq V$ to be traced, Figure 3.2(a) shows the generic form of its tracing query from Theorem 2.5.2.[1]

[1]We consider tracing a tuple set T rather than a single tuple t for generality: it sets the stage for generalizing our results to multilevel views, and in practice we expect that a warehouse tracing package might permit multiple tuples to be traced together for convenience and efficiency.

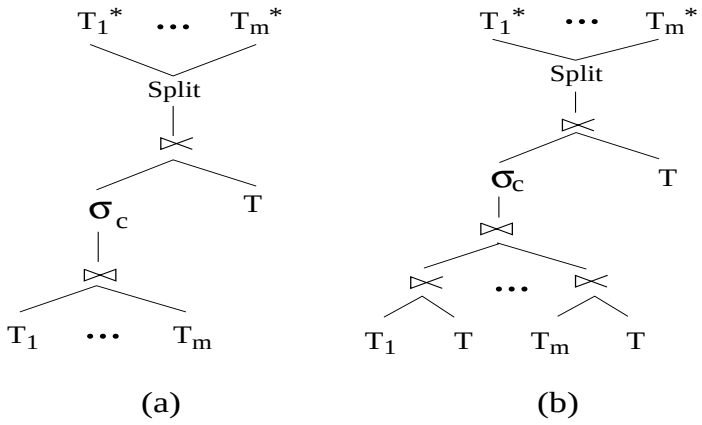


Figure 3.2: Tracing queries

We assume that all local selection conditions in the view—conditions that involve a single base table—are pushed down to the T_i's, so σ_C contains join conditions only. Since the size of T tends to be small, in some cases we also push down the semijoin and rewrite the tracing query as in Figure 3.2(b). The auxiliary views we consider are based on the forms of these two query trees. (Of course since the traced tuple set T is not available until tracing time, we cannot define or maintain auxiliary views on subqueries involving T.) We propose seven schemes for storing auxiliary views to support tracing the lineage of T according to v. For each scheme we specify the new lineage tracing procedure that takes advantage of auxiliary views, as well as the maintenance procedures for the auxiliary views and the original view, since they are all factors in overall performance.

In the maintenance procedures, we use δ to denote the delta tables (as discussed in Section 3.1), but with insertions and deletions combined, and we use $\uplus$ to denote application of the delta tables [GMS93].

3.2.1 Store Nothing ($\varnothing$)

The extreme case is to store no auxiliary views for lineage tracing.

1. Auxiliary views: None
2. Lineage tracing: $TQ_{T,v} = Split_{\mathbf{T_1},\ldots,\mathbf{T_m}}(\sigma_C((T_1 \ltimes T) \bowtie \cdots \bowtie (T_m \ltimes T)) \ltimes T)$
3. Maintenance of auxiliary views: None

4. Maintenance of v [GMS93]: $\delta V = \pi_A(\sigma_C(\delta T_1 \bowtie (T_2 \uplus \delta T_2) \bowtie \cdots \bowtie (T_m \uplus \delta T_m) \uplus T_1 \bowtie \delta T_2 \bowtie (T_3 \uplus \delta T_3) \bowtie \cdots \bowtie (T_m \uplus \delta T_m) \uplus \cdots \uplus T_1 \bowtie \cdots \bowtie T_{m-1} \bowtie \delta T_m))$

This scheme retrieves all necessary information from source tables every time a user poses a lineage tracing query. It incurs no extra storage or maintenance cost, but leads to poor tracing performance. This scheme is included primarily as a baseline to compare with other, more attractive, schemes.

3.2.2 Store Base Tables (BT)

If we can trace the lineage of any tuple in a view without querying the sources, then we say that the view is *self-traceable*. Self-traceable views can be traced correctly even if source tables are inaccessible or inconsistent with the warehouse views. One easy way to make a view self-traceable is to store in the warehouse a copy of each source table that the view is defined on (after local selections), and issue the tracing queries to the local copies instead of to the source tables during lineage tracing. We refer to these source copies as the *Base Tables (BTs)* for v.

1. Auxiliary views: $BT_i = T_i$, $i = 1..m$
2. Lineage tracing: $TQ_{T,v} = Split_{\mathbf{T_1},\ldots,\mathbf{T_m}}(\sigma_C((BT_1 \ltimes T) \bowtie \cdots \bowtie (BT_m \ltimes T)) \ltimes T)$
3. Maintenance of *BT*s: $\delta BT_i = \delta T_i$, $i = 1..m$
4. Maintenance of v: Same as scheme $\varnothing$ replacing T_i with BT_i, $i = 1..m$

Storing base tables can improve primary view maintenance as well as lineage tracing, and maintaining them is fairly easy. However, base tables can be large, even after applying local selections, and much of the source data may be irrelevant to any view tuple's lineage if joins are selective. For the SPJ view `Calif` from our running example, the base tables are simply copies of tables `store`, `item`, and `sales`.

3.2.3 Store Lineage Views (LV)

An alternative way of improving tracing query performance is to store an auxiliary view based on the left subtree in Figure 3.2(a), which we call the *Lineage View (LV)* for v, since it contains all lineage information for all tuples in the primary view.

s_id	s_name	city	state	i_id	i_name	category	price	num
001	Target	Palo Alto	CA	0001	binder	stationery	$4	1000
001	Target	Palo Alto	CA	0002	pencil	stationery	$1	3000
001	Target	Palo Alto	CA	0004	pants	clothing	$30	600
003	Macy's	San Francisco	CA	0003	shirt	clothing	$45	1500
003	Macy's	San Francisco	CA	0004	pants	clothing	$60	600

Figure 3.3: Lineage View (LV)

1. Auxiliary views: $LV = \sigma_C(T_1 \bowtie \cdots \bowtie T_m)$
2. Lineage tracing: $TQ_{t,v} = Split_{\mathbf{T_1},\ldots,\mathbf{T_m}}(LV \ltimes T)$
3. Maintenance of *LV*: $\delta LV = \sigma_C(\delta T_1 \bowtie (T_2 \uplus \delta T_2) \bowtie \cdots \bowtie (T_m \uplus \delta T_m) \uplus T_1 \bowtie \delta T_2 \bowtie (T_3 \uplus \delta T_3) \bowtie \cdots \bowtie (T_m \uplus \delta T_m) \uplus \cdots \uplus T_1 \bowtie \cdots \bowtie T_{m-1} \bowtie \delta T_m)$
4. Maintenance of v: $\delta V = \pi_A(\delta LV)$

The LV scheme significantly simplifies the tracing query and thus reduces tracing query cost. In addition, like base tables, lineage views can be helpful in maintaining the primary view. However, lineage views can be large and are usually expensive to maintain themselves. The contents of the lineage view for `Calif` are shown in Figure 3.3.

3.2.4 Store Split Lineage Tables (SLT)

For views whose joins are many-to-many, lineage views as defined in Section 3.2.3 can be very large, and thus not efficient when performing the semijoin with T during lineage tracing. One solution is to split the lineage view and store a set of tables instead, which we call the *Split Lineage Tables (SLTs)*. Note that we use lineage view *LV* as defined in Section 3.2.3 in the following definitions.

1. Auxiliary views: $SLT_i = \pi_{\mathbf{T_i}}(LV), i = 1..m$
2. Lineage tracing: $TQ_{T,v} = Split_{\mathbf{T_1},\ldots,\mathbf{T_m}}(\sigma_C((SLT_1 \ltimes T) \bowtie \cdots \bowtie (SLT_m \ltimes T)) \ltimes T)$
3. Maintenance of *SLT*s: $\delta SLT_i = \pi_{\mathbf{T_i}}(\delta LV), i = 1..m$
4. Maintenance of v: $\delta V = \pi_A(\delta LV)$

Split lineage tables contain no irrelevant source data, since every tuple in SLT_i, $i = 1..m$, contributes to some view tuples. Furthermore, the size of the split lineage tables can be

SLT-store

s_id	s_name	city	state
001	Target	Palo Alto	CA
003	Macy's	San Francisco	CA

SLT-item

i_id	i_name	category
0001	binder	stationery
0002	pencil	stationery
0003	shirt	clothing
0004	pants	clothing

SLT-sales

s_id	i_id	price	num
001	0001	4	1000
001	0002	1	3000
001	0004	30	600
003	0003	45	1500
003	0004	60	600

Figure 3.4: Split Lineage Tables (SLTs)

much smaller than the lineage view. Their maintenance cost is similar to that of the lineage view. Note that although we do not materialize the lineage view *LV* in the SLT scheme, we still compute δ*LV*, in order to maintain the primary view V and auxiliary views SLT_i, $i = 1..m$. The disadvantage of SLT is that lineage tracing queries may be more expensive. The contents of the split lineage tables for view `Calif` are shown in Figure 3.4.

3.2.5 Store Partial Base Tables (PBT)

Reconsidering the BT scheme (Section 3.2.2), another way to reduce the size of the base tables is to materialize the semijoin of each source table T_i with the primary view V; we call this semijoin result the *Partial Base Table (PBT)* for T_i according to v.

1. Auxiliary views: $PBT_i = T_i \ltimes V$, $i = 1..m$
2. Lineage tracing: $TQ_{T,v} = Split_{\mathbf{T_1},\ldots,\mathbf{T_m}}(\sigma_C((PBT_1 \ltimes T) \bowtie \cdots \bowtie (PBT_m \ltimes T)) \ltimes T)$
3. Maintenance of *PBT*s: $\delta PBT_i = \delta T_i \ltimes (V \uplus \delta V) \uplus T_i \ltimes \delta V$, $i = 1..m$
4. Maintenance of v: Same as scheme $\varnothing$

For views with selective join conditions, the PBT scheme replicates much less source data than the BT scheme, with several benefits: It reduces the storage requirement, as well as the cost of refreshing the auxiliary views. It also reduces the tracing cost, because the tracing query operates on a much smaller table. However, partial base tables do not help with the maintenance of the primary view. Instead, the primary view needs to be maintained first. The partial base tables are then relatively cheap to maintain based on the primary view's contents and changes. The contents of the partial base tables for `Calif` are shown in Figure 3.5.

PBT-store

s_id	s_name	city	state
001	Target	Palo Alto	CA
002	Target	Albany	NY
003	Macy's	San Francisco	CA
004	Macy's	New York City	NY

PBT-item

i_id	i_name	category
0001	binder	stationery
0002	pencil	stationery
0003	shirt	clothing
0004	pants	clothing

PBT-sales

s_id	i_id	price	num
001	0001	4	1000
001	0002	1	3000
001	0004	30	600
003	0003	45	1500
003	0004	60	600

Figure 3.5: Partial Base Tables (PBTs)

3.2.6 Store Base Table Projections (BP)

When source tables have known keys, we can store in our auxiliary views key attributes from the source tables together with other necessary attributes, which we call the *Base Table Projections (BPs)*. This scheme improves tracing query performance (over storing nothing) while reducing view maintenance and storage costs (over storing full source replicas).

1. Auxiliary views: $BP_i = \pi_{A_i}(T_i)$, where A_i includes the key attributes K_i, attributes that are projected into V ($\mathbf{T_i} \cap \mathbf{V}$), and attributes involved in v's join conditions ($\mathbf{T_i} \cap \mathbf{C}$)

2. Lineage tracing: $T_i^* = T_i \ltimes (\sigma_C((BP_1 \ltimes T) \bowtie \cdots \bowtie (BP_m \ltimes T)) \ltimes T)$, ${v^{-1}}_D(T) = \langle {T_1}^*, \ldots, {T_m}^* \rangle$

3. Maintenance of *BP*s: $\delta BP_i = \pi_{A_i}(\delta T_i)$, $i = 1..m$

4. Maintenance of v: Same as scheme $\varnothing$ replacing T_i with BP_i, $i = 1..m$

Note that the semijoins in the tracing procedure are key-based. This scheme can be especially useful when a source table has wide tuples but the view projects only a small fraction. During lineage tracing, the stored information identifies by key which source tuples really contribute to a given view tuple, then the detailed source information is fetched from the source using the key information. Maintenance of the primary view is easy. However, in the BP scheme we do need to query the sources, which has its drawbacks as discussed earlier. The contents of the base table projections for `Calif` are shown in Figure 3.6

BP-store

s_id	s_name	state
001	Target	CA
002	Target	NY
003	Macy's	CA
004	Macy's	NY
005	Safeway	CA
006	Safeway	CO

BP-item

i_id	i_name
0001	binder
0002	pencil
0003	shirt
0004	pants
0005	pot
0006	plate

BP-sales

s_id	i_id	num
001	0001	1000
001	0002	3000
001	0004	600
002	0001	800
002	0002	2000
002	0004	800
003	0003	1500
003	0004	600
004	0003	2100
004	0004	1200
004	0005	200

Figure 3.6: Base Table Projections (BPs)

3.2.7 Store Lineage View Projections (LP)

Again assuming source tables with known keys, we can store a projection over the lineage view (Section 3.2.3) that includes only source table keys and primary view attributes. We call this view the *Lineage View Projection (LP)*. Note that we use lineage view *LV* as defined in Section 3.2.3 in the following definitions.

1. Auxiliary views: $LP = \pi_{A \cup K_1 \cup \cdots \cup K_m}(LV)$, where A is the set of attributes in V, and K_i is the set of key attributes of table T_i, $i = 1..m$

2. Lineage tracing: $T_i^* = T_i \ltimes (LP \ltimes T)$, ${v^{-1}}_D(T) = \langle {T_1}^*, \ldots, {T_m}^* \rangle$

3. Maintenance of *LP*: $\delta LP = \pi_{A \cup K_1 \cup \cdots \cup K_m}(\delta LV)$, $i = 1..m$

4. Maintenance of v: $\delta V = \pi_A(\delta LP)$

Compared with the BP scheme, the LP scheme further simplifies the tracing query and improves tracing performance. However, the maintenance cost for the lineage view projection is higher than for the base table projections. LP also requires a source query as the last step of the tracing process, with the disadvantages previously discussed. The contents of the lineage view projection for `Calif` are shown in Figure 3.7.

s_id	s_name	i_id	i_name	num
001	Target	0001	binder	1000
001	Target	0002	pencil	3000
001	Target	0004	pants	600
003	Macy's	0003	shirt	1500
003	Macy's	0004	pants	600

Figure 3.7: Lineage View Projection (LP)

scheme	∅	BT	LV	SLT	PBT	BP	LP	LV-S	SLT-S	PBT-S
self-traceable?	no	yes	yes	yes	yes	no	no	yes	yes	yes
self-maintainable?	no	yes	no	no	no	yes	no	yes	yes	yes

Table 3.1: Scheme self-traceability and self-maintainability

3.2.8 Self-Maintainability and Self-Traceability

As mentioned in Section 3.2.2, self-traceable views can be traced correctly even if the sources are inaccessible or inconsistent with the warehouse views. Analogously, view self-maintainability as introduced in Section 3.1 ensures that views can be maintained without querying the sources [QGMW96]. In cases where the sources are inaccessible, we must ensure that the primary view together with our auxiliary views are both self-traceable and self-maintainable. Table 3.1 summarizes these properties with respect to the seven schemes introduced so far. We also consider self-maintainable extensions of three of the schemes, LV, SLT, and PBT, calling the extensions LV-S, SLT-S, and PBT-S.

3.3 Performance of Auxiliary View Schemes for SPJ Views

This section presents a simulation-based performance evaluation of our proposed auxiliary view schemes in Section 3.2. We address several questions, including: What is the tracing/maintenance cost distribution and the storage requirement using each scheme? What is the impact of parameters such as the source table size, the number of source tables, the view selectivity, and the tracing-query/update ratio? Finally, which scheme performs the best, in terms of tracing time and maintenance cost, in different settings?

Parameter	Description	Base value
view query load statistics		
query_rate	# of tracing queries per unit time	80
query_size	# of tuples traced per query	10
source update load statistics		
update_ratio	# of source table updates per unit time	20
update_size	# of changed tuples per source table update	10
view (segment) data statistics		
rel_num (fan out)	# of tables in the view (segment)	3
join_ratio	# of joining tuples / cross-product size	0.000125
select_ratio	# of tuples after selection / before selection	0.1
proj_ratio	# of bytes in projection / tuple size in bytes	0.2
aggr_ratio	# of tuples in aggregate / # of tuples before aggregation	1
source data statistics		
tuple_num	# of tuples in source tables	10,000
tuple_size	tuple size of source tables (in bytes)	1000
system statistics		
block_size	block size in bytes at warehouse and source	8000
disk_cost	cost to read/write a disk block in ms/block	10
trans_cost	network transmission cost (in ms/byte)	0.2
msg_cost	network setup cost (in ms/message)	100

Table 3.2: System and data statistics

3.3.1 System Model and Cost Metrics

We consider a simple warehousing system as depicted in Figure 1.1 of Chapter 1. Table 3.2 summarizes the configuration parameters. Recall that we study only SPJ views in this section, and we consider two types of operations on the view: lineage tracing queries and view maintenance. Most entries in the table are self-explanatory. We assume all local selection conditions are pushed down to corresponding source tables and have been incorporated in the source table size. For simplicity, we do not model cache behavior, and we assume that all source tables in the view have the same statistics. We also assume that the warehouse and the source databases have the same data block size and the same disk access cost. Finally, we assume that all sources can perform simple SPJ operations, and all join operations are nested-loop index joins. We consider table scans as well as key-based index lookups. We use the base values in Table 3.2 as the baseline setting for our experiments, varying relevant parameters one at a time.

In our performance analysis we consider several performance metrics. The first is lineage tracing query performance, where we use the average tuple lineage tracing time as the metric. The second metric measures view maintenance cost, including total time spent maintaining auxiliary views as well as the primary view. We consider total view maintenance cost since, as discussed earlier, certain auxiliary view schemes for lineage tracing also improve the performance of primary view maintenance. For both lineage query and view maintenance costs, we consider database access and network cost. More specifically:

lineage tracing (resp. view maintenance) cost
$=$ disk_cost $*$ # of disk I/Os in lineage tracing (resp. in view maintenance)
$+$ trans_cost $*$ # of bytes transmitted in lineage tracing (resp. in view maintenance)
$+$ msg_cost $*$ # of network messages in lineage tracing (resp. in view maintenance)

To estimate the number of disk I/Os, bytes transmitted, and network messages in lineage tracing and view maintenance, we use a fairly conventional cost model for relational queries in a distributed database setting, similar to, e.g., [LQA97, Ull89b, ZGMHW95]. The cost formulas rely on all of the statistics from Table 3.2 and the assumptions we made earlier. We also consider total tracing query and view maintenance time based on the query/update ratio. Finally, we consider the size of the primary view and auxiliary views, which compares the schemes' storage requirements.

3.3.2 Experiments and Results

We present a sample of five experiments addressing the questions raised at the beginning of this section. For each experiment we simulated a total of 1000 operations and estimate the tracing query and view maintenance costs based on the system and cost model introduced in Section 3.3.1. Each operation is either a lineage tracing query (tracing a set of view tuples) or view maintenance, which computes and applies changes to the primary view and auxiliary views based on a set of source changes. We look at the overall performance of our schemes using the base settings of Table 3.2, investigating their storage requirement and how their cost distributes among the relevant cost components. We then study the impact

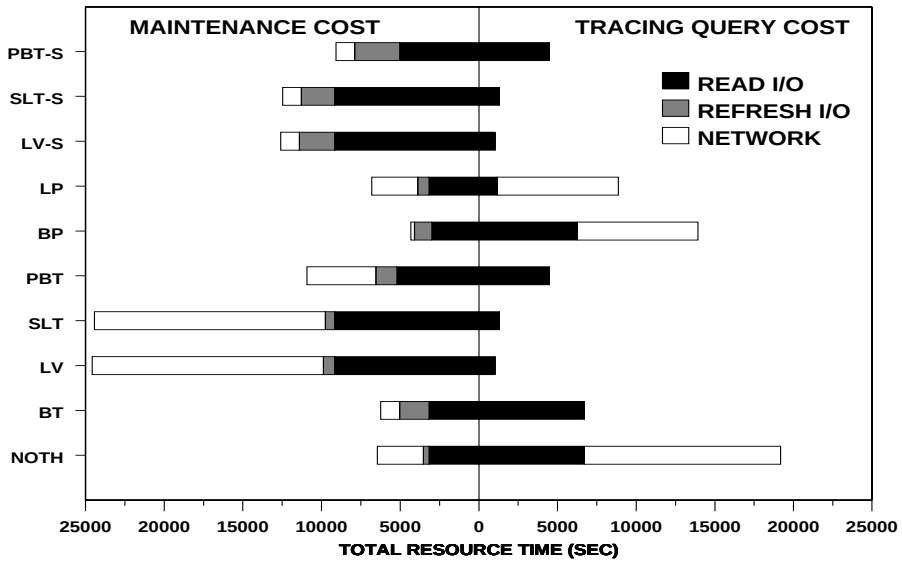


Figure 3.8: Cost distribution under base settings

on the various schemes of source table size, number of source tables, view selectivity, and query/update ratio. Because we measure all ten schemes in our experiments, some graphs are admittedly difficult to decipher, so we highlight the most important aspects of our results in text.

Cost Distribution and Storage. Our first experiment compares the performance of our ten proposed schemes under the base settings. Figure 3.8 shows the cost distributions divided into two parts: the tracing query cost on the right and the view maintenance cost on the left. Figure 3.9 shows the storage requirement of each scheme, including the primary and auxiliary views. (The primary view is fairly selective under our base settings, resulting in the high variance in storage requirement.) From Figure 3.8 we see that under our base settings, LV-S and SLT-S achieve low lineage tracing cost as well as fairly low total cost. $\varnothing$ (NOTH) has the highest tracing cost but low maintenance cost as expected, while conversely LV and SLT have low tracing cost but high maintenance cost. BT and PBT are reasonable compromises between the two extremes. The self-maintainable extensions (LV-S, SLT-S, and PBT-S) significantly reduce the network cost for maintenance, at the expense of higher refresh I/O cost and higher storage requirements (Figure 3.9). The projection-based schemes

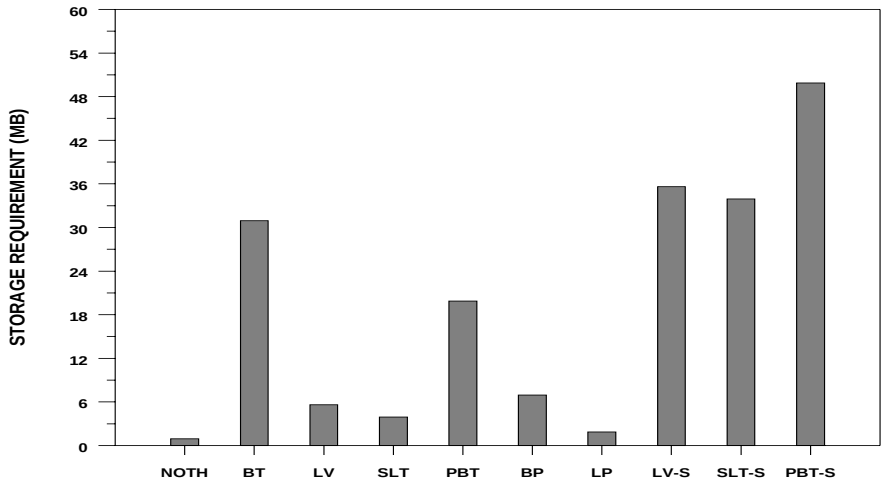


Figure 3.9: Storage cost

(BP and LP) achieve low tracing and maintenance I/O cost, but since they require queries to the sources the network cost remains high. Figure 3.9 shows that $\varnothing$, LV, SLT, BP, and LP have the lowest storage requirements.

Impact of Source Table Size. Next, we look at how our schemes are affected by source table size scale-up. In Figure 3.10, we vary the size of each source table from 1000 to $50,000$ tuples and study the impact on tracing query cost (per traced tuple) and maintenance cost (per updated tuple). The results tell us that as the source table size increases, SLT and SLT-S provide the lowest tracing cost (the cost is identical for the two schemes), because they filter out all base tuples that are irrelevant to the lineage of any view tuple. BP and BT, on the other hand, incur the lowest maintenance costs, because of the simplicity of their definition.

Impact of Source Table Number. We now consider scale-up in the number of source tables. From the results in Figure 3.11, we observe that LV and LV-S, which performed poorly in lineage tracing when scaling up source table size, present the best tracing performance in source number scale-up: tracing queries using lineage views do not involve any join operations. BP, BT, and PBT-S incur the lowest maintenance cost as the number of source tables increases, because maintaining these auxiliary views does not require joining multiple source tables.

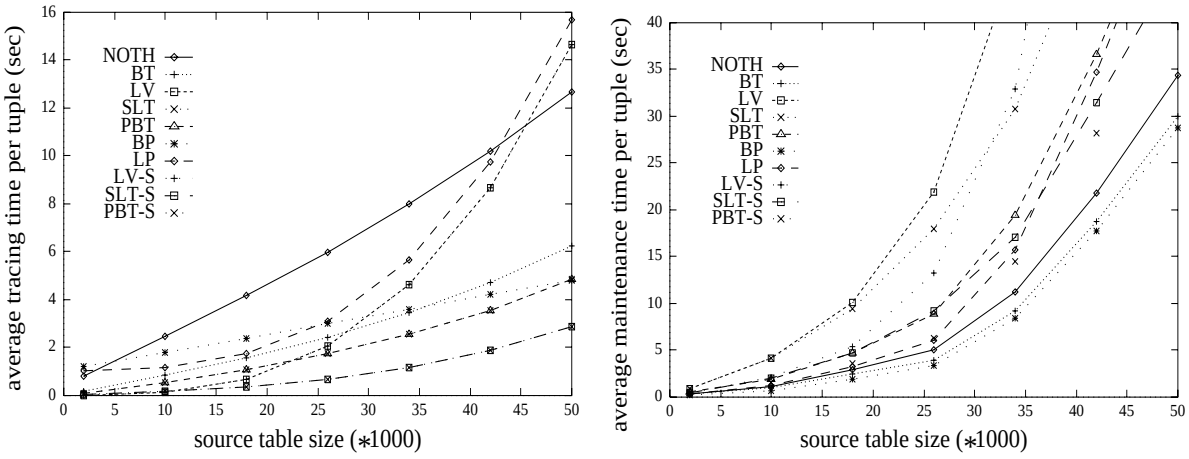


Figure 3.10: Impact of source table size

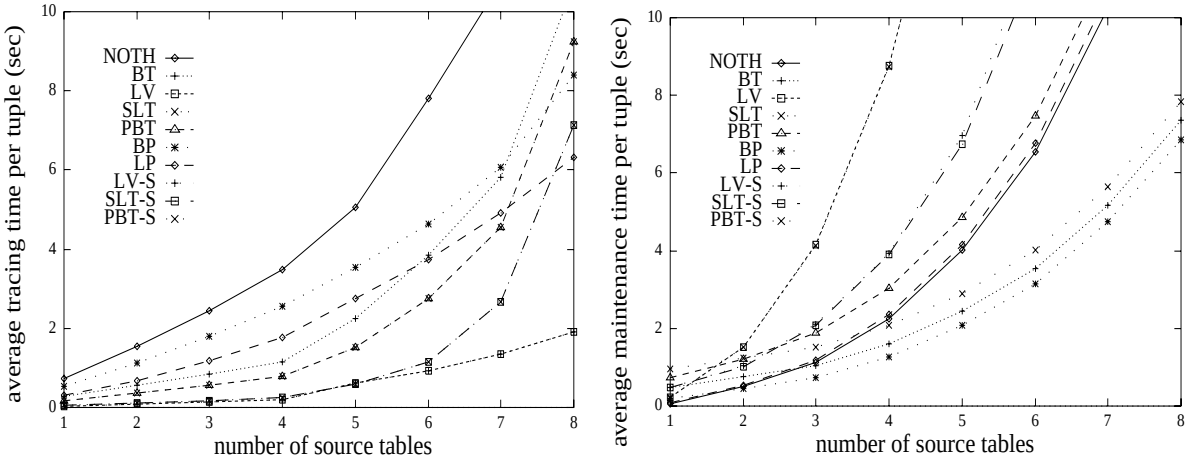


Figure 3.11: Impact of number of source tables

Impact of View Join Selectivity. This experiment studies how the join selectivity of the primary view affects the schemes. Figure 3.12 shows the results, where we vary the view join ratio from 0.0001 to 0.0005. We can see that the tracing performance of LV, LV-S, and LP degrades substantially as the view join ratio increases, while other schemes are much less sensitive. This behavior is because the size of the lineage view grows exponentially with the join ratio, unlike the other auxiliary views. In summary, SLT and SLT-S provide the lowest tracing cost, while BP and BT again incur the lowest maintenance cost.

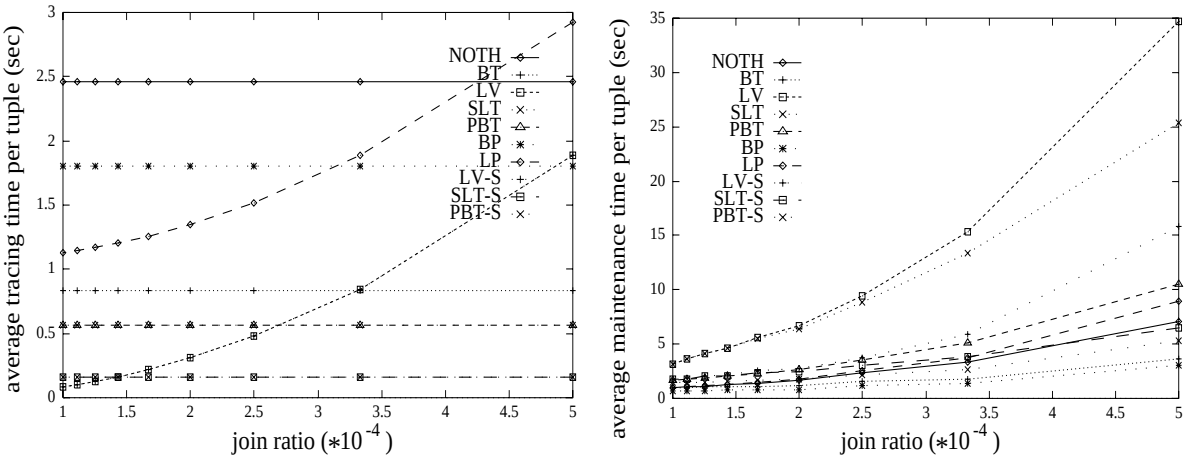


Figure 3.12: Impact of view join selectivity

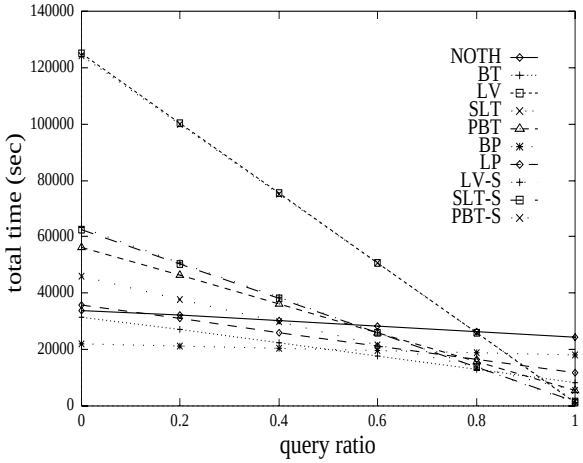


Figure 3.13: Impact of workload pattern

Impact of Workload Pattern. Our final experiment studies the impact of the query ratio on total tracing and maintenance cost. In Figure 3.13, the x-axis varies the ratio from one extreme where the query ratio = 0 (no tracing queries) to the other extreme where the query ratio = 1 (no view maintenance). According to the total cost shown in the figure, auxiliary view schemes that lead to low tracing cost, such as LV-S and SLT-S, are preferred for high query ratios. BT, a simple scheme that benefits both lineage tracing and view maintenance, is preferred for medium query ratios. Finally, BP, which incurs the lowest overall view maintenance cost, is preferred for low query ratios.

3.4 Auxiliary View Selection for General ASPJ Views

So far we have considered auxiliary view schemes for improved lineage tracing and view maintenance performance for SPJ views only. In this section, we consider the problem of selecting auxiliary views for arbitrary complex multilevel ASPJ primary views. We first specify several types of potentially useful auxiliary views for an arbitrary ASPJ primary view, based on our SPJ auxiliary views from Section 3.2, and discuss how view maintenance procedures and lineage tracing queries can take advantage of these auxiliary views in the more general setting of this section. We then formally define the auxiliary view selection problem and estimate the size of our search space. In Sections 3.5 and 3.6, we will discuss our detailed cost model and view selection algorithms. We use our example view `HighProfit` from Section 3.1 throughout these sections.

3.4.1 The Auxiliary Views We Consider

Given an ASPJ view, we divide its normalized view definition into three types of components, and for each type of component we consider certain choices of possible auxiliary views to materialize:

1. **Topmost Segment**: the segment at the root of the view definition tree. Recall from the definition of ASPJ normal form (Section 2.5.1) that the α, π, σ, and/or $\bowtie$ operators (but not all of them) may be omitted in this segment. Also note that the topmost node corresponds to the primary view itself, so its contents are always materialized. For topmost segments we also consider materializing the lineage view (LV) or the split lineage tables (SLTs) for this segment as defined in Section 3.2, but not both. (If we store one, then storing the other will not further reduce the lineage tracing or overall maintenance cost.)

2. **Intermediate Segment**: a non-root segment that is defined over the source tables and/or other segments. Recall again from the definition of ASPJ normal form that the π, σ, and/or $\bowtie$ operators may be omitted in this segment, but the α operator is always present. For an intermediate segment, we consider materializing the following auxiliary views:

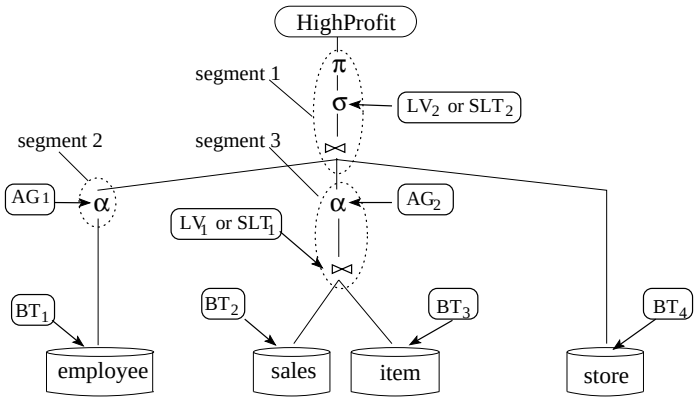


Figure 3.14: Possible auxiliary views for `HighProfit`

(a) The contents of the α node, which we refer to as AG

(b) The lineage view (LV) or the split lineage tables (SLTs), but not both

3. **Source Table**: We continue to assume that all local selection conditions in the view are pushed down to the source tables. For each source table R, we decide whether to store a base table (BT) copy of R, as defined in Section 3.2. If BT is not materialized, we may need to issue queries directly to source table R for view maintenance and lineage tracing.[2]

Example 3.4.1 (Auxiliary Views) Recall view `HighProfit` from Figure 3.1. Figure 3.14 shows the three ASPJ segments in the view definition, among which segment 1 is the top segment, segments 2 and 3 are intermediate segments, and there are four source tables. Figure 3.14 also shows all of the possible auxiliary views we consider materializing for `HighProfit`. □

Notice that we are not considering all of the auxiliary view schemes introduced for SPJ views earlier in this chapter. Instead, we restrict our search space to only three auxiliary view choices for each ASPJ segment, BT, LV, and SLT, because these three auxiliary view choices are the most representative and interesting ones for the multilevel ASPJ view scenario. (Note that an intermediate aggregate view (AG) for the top of an ASPJ segment

[2]Most existing production data warehousing systems automatically store a copy of each source table in the warehouse. However, as we will see in Section 3.6, sometimes it is not beneficial to store a copy.

serves as a base table (BT) for the next higher segment.) We could extend our auxiliary view search space to include all 11 choices for each ASPJ segment as proposed in Section 3.2, dramatically increasing an already very large search space. Also notice that we are not considering different join combinations in the case of a many-way join. This special case is considered in detail in previous work [LQA97], and we could extend our search space accordingly.

3.4.2 Lineage Tracing and View Maintenance Using Auxiliary Views

In Section 3.2 we specified how to rewrite queries for lineage tracing and view maintenance using the different auxiliary view schemes suggested for SPJ views. In general, when we have a set of auxiliary views available, there may be more than one way to rewrite a query to take advantage of auxiliary views. We assume that the "best" rewriting is selected, and this assumption is reflected in the cost model to be presented in Section 3.5. For example, we can rewrite the tracing query TQ for a tuple t according to the topmost segment in the definition of `HighProfit` using different auxiliary views in Figure 3.14 as follows.

$$TQ_1 = Split_{\mathbf{R1},\mathbf{R2},\mathbf{R3}}(\sigma_{\texttt{profit}-\texttt{expenses}>100000 \wedge \texttt{s\_name}=t.\texttt{s\_name}}(AG_1 \bowtie AG_2 \bowtie BT_4))$$
$$TQ_2 = Split_{\mathbf{R1},\mathbf{R2},\mathbf{R3}}(\sigma_{\texttt{s\_name}=t.\texttt{s\_name}}(LV_2))$$
$$TQ_3 = Split_{\mathbf{R1},\mathbf{R2},\mathbf{R3}}(\sigma_{\texttt{s\_name}=t.\texttt{s\_name}}(SLT_2^{\mathbf{R_1}} \bowtie SLT_2^{\mathbf{R_2}} \bowtie SLT_2^{\mathbf{R_3}}))$$

$\mathbf{R1}, \mathbf{R2}, \mathbf{R3}$ in the tracing queries above represent the schemas of the three leaf nodes of the topmost segment. Suppose that LV_2, AG_1, and BT_4 are materialized. Then we could use query TQ_2, or (among other options) we could use a tracing query similar to TQ_1 that recomputes the contents of AG_2. In this case it is likely that TQ_2 would be chosen as the best query rewriting based on the available auxiliary views.

3.4.3 The View Selection Problem and the Search Space

We have shown various auxiliary views that can be used in the view maintenance and lineage tracing processes for general ASPJ views, and obviously they will lead to different

warehouse performance. Our goal is to select among the choices of auxiliary views described in Section 3.4.1 a set that minimizes overall cost: the cost of lineage tracing plus the cost of maintaining the primary and auxiliary views. Our detailed cost model is described in Section 3.5. Here let us consider the size of our search space.

Given an ASPJ view v, we say that v is an n-level ASPJ view if traversing from the root to any leaf in its normalized definition tree crosses at most n segments. The *fan-out* of a segment is the number of operands of the segment's join operator, or 1 if there is no join. For instance, our example view `HighProfit` from Section 3.1 is a 2-level ASPJ view containing three segments: the topmost π-σ-$\bowtie$ segment with fan-out 3, the leftmost α segment with fan-out 1, and the middle α-$\bowtie$ segment with fan-out 2.

Suppose we have an n-level ASPJ view in normal form, and consider a balanced view definition tree[3] with a fan-out of m in each segment. There is one topmost segment, and for that segment we have 3 auxiliary view options: LV, SLTs, or nothing (case 1 in Section 3.4.1). There are $m^1 + m^2 + \cdots + m^{n-1} = O(m^{n-1})$ intermediate segments, each having 2 options for case 2(a) in Section 3.4.1 (AG or nothing) and 3 options for case 2(b) (LV, SLTs, or nothing). Finally, there are m^n source tables, each having 2 options: BT or nothing. Therefore, the size of the entire search space is

$$3^1 \cdot (2 \cdot 3)^{O(m^{n-1})} \cdot 2^{(m^n)} = O(2^{m^n})$$

If k is the total number of components in the view definition, where a component is a segment or a source table, then $k = O(m^n)$ and the search space size is $O(2^k)$.

Example 3.4.2 (Search Space Size) Consider our example view `HighProfit` (Figure 3.14). The number of possible auxiliary view sets for `HighProfit` is $2^6 \cdot 3^2 = 384$. □

The number of choices for `HighProfit` is quite manageable. However, real warehouse views tend to have much higher fan-outs, as well as possibly more levels. As we will see in Section 3.7, even for a view with only 2 levels and average fan-out of 5, we cannot consider all possible auxiliary view sets due to the large search space.

[3] A tree is balanced if each leaf node in the tree has the same depth. We consider this view definition shape since it represents the largest search space size for an n-level view.

3.5 Cost Model

In this section we extend the cost model we introduced in Section 3.3 for SPJ views to estimate the total view maintenance and lineage tracing costs for an arbitrary ASPJ primary view given a set of auxiliary views. We also define a metric for measuring how close a given auxiliary view set is to the optimal one.

Let $cost(Q, s)$ denote the estimated cost of evaluating a query Q at the warehouse given a set of statistics s. Q could be a lineage tracing query, or a query or update in a view maintenance procedure. $cost(Q, s)$ is computed based on the statistics from Table 3.2 as described in Section 3.3.1.

Suppose we have a primary ASPJ view v and a set of materialized auxiliary views $\mathcal{A} = \{v_1, ..., v_n\}$. To trace the lineage of tuples in the primary view given the auxiliary views in $\mathcal{A}$, there are various possible rewritings of the lineage tracing queries using the auxiliary views (recall Section 3.4.2). Our cost model selects the sequence of lineage tracing queries with the lowest estimated cost. Let $q(v, \mathcal{A}, s)$ denote the estimated lineage tracing cost for primary view v, auxiliary views $\mathcal{A}$, and statistics s:

$$q(v, \mathcal{A}, s) = \sum_{1..m} cost(Q_i, s)$$

where $Q_1, \ldots, Q_m$ is the set of tracing queries selected for v given auxiliary view set $\mathcal{A}$. We assume that the lineage query rate and the average number of tuples traced in a lineage query (part of our usage statistics in Table 3.2) are included in the input statistics s, and thus are incorporated into the lineage cost estimated by $q(v, \mathcal{A}, s)$.

Maintenance costs are incurred both for the primary view v and for the auxiliary views in $\mathcal{A} = \{v_1, ..., v_n\}$. As with lineage tracing, when there are multiple possible rewritings for the view maintenance queries and updates using the auxiliary views in $\mathcal{A}$, our cost model selects the ones with the lowest estimated cost. Let $m(v, \mathcal{A}, s)$ denote the estimated maintenance cost for primary view v, auxiliary views $\mathcal{A}$, and statistics s:

$$m(v, \mathcal{A}, s) = \sum_{1..n} cost(M_i, s)$$

where $M_1, \ldots, M_n$ is the set of maintenance queries and updates selected to maintain primary view v and the auxiliary views in $\mathcal{A}$. We assume that the source table update rate and average number of source tuples changed in each update (part of our usage statistics in Table 3.2) are included in the input statistics s, and thus are incorporated into the maintenance cost estimated by $m(v, \mathcal{A}, s)$.

Finally, the total cost is the combination of lineage tracing cost and view maintenance cost:

$$total\_cost(v, \mathcal{A}, s) = q(v, \mathcal{A}, s) + m(v, \mathcal{A}, s)$$

The *optimality* of a set of auxiliary views represents how close the set is to the auxiliary view set that yields the lowest estimated cost. For a given primary view v and statistics s, let $\mathcal{A}_{opt}$ denote the set of auxiliary views within our search space (Section 3.4) with the lowest total cost $total\_cost(v, \mathcal{A}_{opt}, s)$. For a set of auxiliary views $\mathcal{A}$, we define the optimality of $\mathcal{A}$ as:

$$optimality(\mathcal{A}) = \frac{total\_cost(v, \mathcal{A}_{opt}, s)}{total\_cost(v, \mathcal{A}, s)}$$

For example, if the total cost for auxiliary view set $\mathcal{A}$ is three times the total cost for $\mathcal{A}_{opt}$, then the optimality of $\mathcal{A}$ is $1/3$, or 33%. The optimality of $\mathcal{A}_{opt}$ is 1, or 100%.

3.6 Algorithms for Selecting Auxiliary Views

Having defined our search space for the optimization problem and the cost model that we use, we now introduce four different algorithms for selecting a set of auxiliary views within the search space. The input to each algorithm is the primary view definition v in ASPJ normal form, and a set of statistics s as specified in Table 3.2. The output is a set of auxiliary views $\mathcal{A}$.

Parameter	Values							
	HighProfit	employee	sales	item	store	segment 1	segment 2	segment 3
query_rate	100							
query_size	1							
update_rate		10	10	10	0			
update_size		1	100	1	0			
tuple_num		10000	1000000	100000	100			
tuple_size		1000	500	500	400			
fan-out						1	2	3
join_ratio							0.0002	0.0001
select_ratio								0.1
proj_ratio						0.1	0.1	0.1
aggr_ratio						0.01	0.001	0.2
block_size	8K	8K	8K	8K	8K			
disk_cost	1	1	1	1	1			
net_cost	0	0.00001	0.0001	0.0001	0.0001			

Table 3.3: Sample statistics for view HighProfit

3.6.1 Exhaustive Algorithm

The exhaustive algorithm enumerates all choices in the search space, estimates the cost of each choice, and picks the cheapest one. For our example view `HighProfit`, the exhaustive algorithm considers all 384 possible combinations of auxiliary views (recall Example 3.4.2). We set a sample set of statistics s as shown in Table 3.3, including statistics for view `HighProfit`, source tables `employee`, `sales`, `item`, and `store`, as well as each ASPJ segment in the view definition (Figure 3.14). Over this set of statistics, the exhaustive algorithm selects $\mathcal{A} = \{BT_4, AG_1, AG_2, SLT_1, LV_2\}$. The exhaustive algorithm always finds the optimal auxiliary view set according to our cost model. However, the complexity of the algorithm is the same as the search space size: $O(2^k)$ where k is the number of components in the view definition (recall Section 3.4.3).

3.6.2 Naive Algorithm

At the other end of the spectrum, we consider a naive algorithm that selects a fixed set of auxiliary views: LVs for the topmost and all intermediate segments, AGs for all intermediate segments, and all BTs. For example view `HighProfit`, the naive algorithm selects $\mathcal{A} = \{BT_1, BT_2, BT_3, BT_4, AG_1, AG_2, LV_1, LV_2\}$ (Figure 3.14). Even though this naive

fixed set of auxiliary views may not be optimal—in fact it can be arbitrarily bad compared to the optimal set—our experimental results in Section 3.7 show that the naive algorithm selects reasonably good view sets in many cases, especially considering its simplicity. The complexity of the naive algorithm is $O(1)$.

3.6.3 Greedy Algorithm

We also consider a conventional greedy algorithm. This algorithm initializes the auxiliary view set $\mathcal{A}$ to be empty. In each iteration, it adds into $\mathcal{A}$ the auxiliary view (not yet in $\mathcal{A}$) that brings the most benefit, i.e., reduces the total cost the most given the current set of views in $\mathcal{A}$. Iteration continues until there are no more auxiliary views outside $\mathcal{A}$ that can further reduce the total cost. For our example view `HighProfit` and the same sample statistics used in the exhaustive algorithm (Section 3.6.1), the greedy algorithm selects the optimal set of auxiliary views in the following order: AG_2, SLT_1, AG_1, LV_2, BT_4.

The greedy algorithm has complexity $O(k^2)$ instead of $O(2^k)$ as in the exhaustive algorithm, and it selects the optimal auxiliary view set in most cases (see Section 3.7). However, the greedy algorithm cannot guarantee an optimal answer, nor even an answer within some percentage of optimal. In Section 3.7.3, we will see a scenario where the greedy algorithm performs poorly.

3.6.4 Three-Step Algorithm

Our last algorithm divides the auxiliary view selection process into three phases. See Figure 3.15. In the first phase, we use a greedy approach to add auxiliary views of the AG and BT types only. In the second phase, we decide for the topmost and each intermediate segment whether to add LV or SLTs. At this point, it may turn out that some of the AG or BT views selected in the first phase are no longer beneficial given the LV or SLT views selected in the second phase, and they incur maintenance cost. Thus, in third phase we remove AG and BT views that are not beneficial, and we do so in a greedy manner.

For example view `HighProfit` and the same sample statistics from Table 3.3, the three-step algorithm also selects the optimal set of auxiliary views. In phase 1, it selects AG and BT views in the following order: AG_2, AG_1, BT_2, BT_3, BT_4. In phase 2, it selects SLT_1

input: primary view v, statistics s
output: auxiliary view set $\mathcal{A}$
begin
  $\mathcal{A} \leftarrow \varnothing$;

  // phase 1: use greedy algorithm on AG and BT nodes
  $\mathcal{A}_{all} \leftarrow$ all possible auxiliary views for v;
  while true **do**
    for each $v_i \in \mathcal{A}_{all}$ of type AG or BT such that $v_i \notin \mathcal{A}$ **do**
      benefit$_i \leftarrow total\_cost(v, \mathcal{A}, s) - total\_cost(v, \mathcal{A} \cup \{v_i\}, s)$;
    pick v_i with the highest *benefit*$_i$;
    if *benefit*$_i \leq 0$ **then** break **else** $\mathcal{A} \leftarrow \mathcal{A} \cup \{v_i\}$;
  endwhile;

  // phase 2: decide LV and SLTs
  for the topmost and each intermediate segment **do**
    // Let LV and SLT be the Lineage View and Split Lineage Tables for the segment
    $cost_1 \leftarrow cost(v, \mathcal{A} \cup \{LV\}, s)$;
    $cost_2 \leftarrow cost(v, \mathcal{A} \cup \{SLT\}, s)$;
    if $cost_1 \geq cost_2 > cost(v, \mathcal{A}, s)$ **then** $\mathcal{A} \leftarrow \mathcal{A} \cup \{LV\}$
    else if $cost_2 > cost_1 > cost(v, \mathcal{A}, s)$ **then** $\mathcal{A} \leftarrow \mathcal{A} \cup \{SLT\}$;
  endfor;

  // phase 3: remove useless AGs and BTs
  while true **do**
    for each $v_i \in \mathcal{A}$ of type AG or BT **do**
      benefit$_i \leftarrow total\_cost(v, \mathcal{A} - \{v_i\}, s) - total\_cost(v, \mathcal{A}, s)$;
    pick v_i with the lowest *benefit*$_i$;
    if *benefit*$_i > 0$ **then** break **else** $\mathcal{A} \leftarrow \mathcal{A} - \{v_i\}$;
  endwhile;
  return $\mathcal{A}$;
end

Figure 3.15: The three-step algorithm

and LV_2. In phase 3, it removes BT_2 and BT_3 (in that order) because they are no longer beneficial given the views selected in phase 2.

The three-step algorithm has complexity $O(k^2)$, which is the same as the greedy algorithm, but its actual running time is less than the greedy algorithm by a linear factor.

The first phase of the three-step algorithm is faster than the greedy algorithm since it only selects from the AG and BT views, instead of from all auxiliary views. The second phase is linear in the number of segments. The third phase only examines the AG and BT views selected in the first phase, which is a small number in most cases.

Like the greedy algorithm, the three-step algorithm usually selects the optimal set of auxiliary views (see Section 3.7). However, also like the greedy algorithm, the three-step algorithm cannot make any guarantees about the optimality of its answers. In Section 3.7.3 we will see a scenario where the three-step algorithm performs poorly. Interestingly, in the case we show where the three-step algorithm performs poorly, the greedy algorithm performs well, and vice-versa. Thus, one practical option is to combine the two algorithms: run both algorithms and select whichever answer has lower estimated cost. The running time of the combined algorithm remains $O(k^2)$.

3.7 Performance of Auxiliary View Selection Algorithms

In this section, we study the performance of our four auxiliary view selection algorithms specified in Section 3.6, comparing their running times and the optimality of the answers they produce. We also compare the cost of the answers produced by these algorithms against the cost of storing no auxiliary views. In Section 3.7.1, we present results of experiments using the schema, statistics, and some views from the TPC-D benchmark. In Section 3.7.2, we present results of experiments using more complex synthetic view definitions. Since the greedy and three-step algorithms perform quite well in all of the experiments in Sections 3.7.1 and 3.7.2, in Section 3.7.3 we show experiments illustrating that greedy and three-step can perform poorly.

3.7.1 TPC-D Experiments

Our first set of experiments is based on the TPC-D benchmark [TPC96]. We use the schema of tables `Customer`, `Order`, `Lineitem`, `Supplier`, `Nation`, `Region`, `PartSupp`, and `Part` from the benchmark for our experiments. The table statistics we use correspond to a scaling factor of 1. The remaining statistics from Table 3.2 are set according to the benchmark and

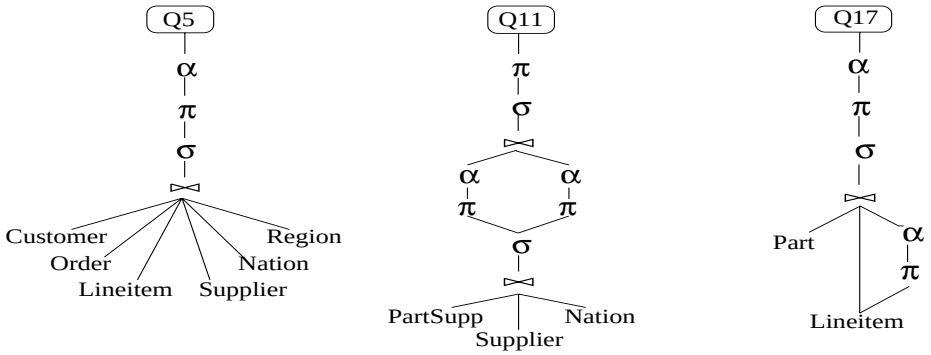


Figure 3.16: Materialized views for TPC-D experiments

Parameter	**Values**							
	Q_5	Customer	Order	LineItem	Supplier	Nation	Region	segment 1
query_rate	100							
query_size	1							
update_rate		1	10	40	1	0	0	
update_size		1	10	10	1	0	0	
tuple_num		150000	1500000	6000000	10000	25	5	
tuple_size		300	100	300	200	100	100	
fan-out								6
join_ratio								0.00001
select_ratio								0.01
proj_ratio								0.1
aggr_ratio								0.001
block_size	8K	8K	8K	8K	8K	8K	8K	
disk_cost	1	1	1	1	1	1	1	
net_cost	0	0.0001	0.00001	0.00001	0.0001	0.0001	0.0001	

Table 3.4: Statistics for Q_5

commonly used database and network system settings. For example, we set the update rate for the `Lineitem` and `Order` tables to be much higher than other tables, since `Lineitem` and `Order` are the *fact tables* according to the benchmark specification. For views, we select queries Q_5, Q_{11}, and Q_{17} from the benchmark, since they are relatively complex and differ somewhat from each other. In each case, we treat the benchmark query as the definition of our primary materialized view to be stored at the warehouse. The general structure of each of the three views is shown in Figure 3.16. The complete list of statistical settings (recall Table 3.2) for our three TPC-D experiments is shown in Tables 3.4–3.6.

Recall that we are comparing five algorithms—the four algorithms from Section 3.6,

Parameter	Values						
	Q_{11}	PartSupp	Supplier	Nation	segment 1	segment 2	segment 3
query_rate	100						
query_size	1						
update_rate		10	1	0			
update_size		1	1	0			
tuple_num		800000	10000	25			
tuple_size		100	200	100			
fan-out					1	3	3
join_ratio						0.0006	0.0006
select_ratio					0.1	0.04	0.04
proj_ratio					0.6	0.1	0.1
aggr_ratio						0.005	0.005
block_size	8K	8K	8K	8K			
disk_cost	1	1	1	1			
net_cost	0	0.0001	0.0001	0.0001			

Table 3.5: Statistics for Q_{11}

Parameter	Values				
	Q_{17}	Part	Lineitem	segment 1	segment 2
query_rate	100				
query_size	1				
update_rate		10	10		
update_size		1	10		
tuple_num		200000	6000000		
tuple_size		200	300		
fan-out				3	1
join_ratio				0.0002	
select_ratio				0.002	
proj_ratio				0.1	0.2
aggr_ratio				0.001	0.03
block_size	8K	8K	8K		
disk_cost	1	1	1		
net_cost	0	0.0001	0.00001		

Table 3.6: Statistics for Q_{17}

as well as the "algorithm" that selects no auxiliary views (which we call algorithm *none*). Figure 3.17 plots the optimality of the five algorithms for each of the TPC-D views we consider. Recall from Section 3.5 that optimality is defined as the cost of the optimal auxiliary view set divided by the cost of the chosen view set. The rightmost bar for Q17 in Figure 3.17 is barely visible, meaning that the scheme of storing no auxiliary views has extremely low optimality (i.e., high cost). Figure 3.18 plots the running time of the

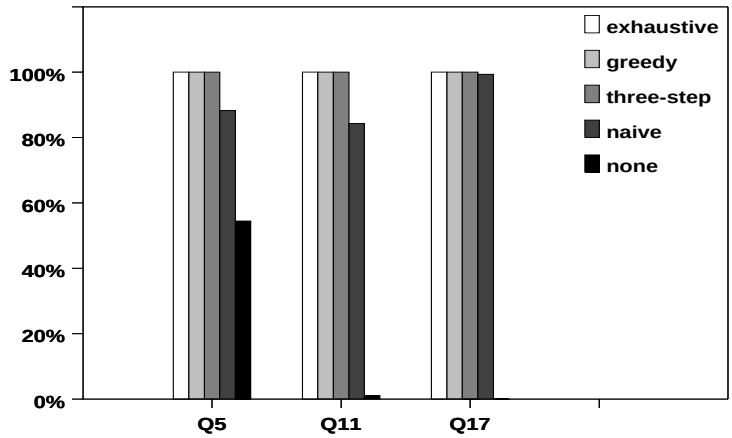


Figure 3.17: Optimality for TPC-D views

	exhaustive	greedy	three-step	naive	none
Q_5	11.15	4.79	2.38	0.08	0.09
Q_{11}	14.07	0.94	0.38	0.03	0.04
Q_{17}	0.62	0.29	0.10	0.02	0.03

Figure 3.18: Running time (seconds) for TPC-D

algorithms. Note that algorithms *naive* and *none* do incur a small running time, which is the time required to compute the cost of their one solution.

We can see from Figure 3.17 that storing no auxiliary views can be dramatically worse than storing some, even those selected by the naive algorithm. We also see from Figure 3.17 that the greedy and three-step algorithms select the optimal auxiliary view set for all three views, and from Figure 3.18 we see that they do so in a small fraction of the running time required by the exhaustive algorithm. We also note that the three-step algorithm runs considerably faster than the greedy algorithm.

3.7.2 Synthetic Experiments

Our next set of experiments is conducted using synthetic views and data statistics. The views we consider all have a regular tree definition, as illustrated by view v_1 in Figure 3.19. We consider seven different views. Figure 3.20 summarizes the "shape" of each view definition tree (number of levels and fan-out of each segment), along with the query/update ratio, which represents the ratio of the average number of tracing queries per unit time to

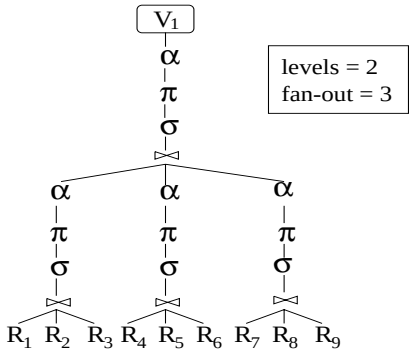


Figure 3.19: Structure of view v_1

	levels	fan-out	query/update ratio
v_1	2	3	100
v_2	2	3	10
v_3	2	3	1
v_4	2	3	0.1
v_5	2	3	0
v_6	6	1	10
v_7	2	5	10

Figure 3.20: Synthetic configurations

Parameter	**Values**						
	v_1	v_2	v_3	v_4	v_5	each source table	each segment
query_rate	1000	100	10	1	0		
query_size	1	1	1	1	1		
update_rate						10	
update_size						1	
tuple_num						10000	
tuple_size						100	
levels	2	2	2	2	2		
fan-out							3
join_ratio							0.001
select_ratio							0.2
proj_ratio							0.4
aggr_ratio							0.1
block_size	8K	8K	8K	8K	8K	8K	
disk_cost	1	1	1	1	1	1	
net_cost	0	0	0	0	0	0.0001	

Table 3.7: Statistics for synthetic views v_1–v_5

the average number of source updates (recall Table 3.2). The complete set of statistical settings for the seven experiments is summarized in Tables 3.7–3.9.

Figure 3.21 plots the optimality of our five algorithms for each of the seven synthetic views we consider. Figure 3.22 plots the running time of the algorithms. The greedy and three-step algorithms always select the optimal auxiliary view set or, in the one case of the three-step algorithm on v_1, very near to optimal. (Actually, for view v_7 the exhaustive algorithm never finished, so optimality is measured against the auxiliary view set selected by the greedy algorithm.) The greedy and three-step algorithms find their answer in a small

Parameter	Values		
	v_6	source table	segment
query_rate	100		
query_size	1		
update_rate		10	
update_size		1	
tuple_num		1000000	
tuple_size		1000	
levels	6		
fan-out			1
join_ratio			
select_ratio			0.5
proj_ratio			0.8
aggr_ratio			0.2
block_size	8K	8K	
disk_cost	1	1	
net_cost	0	0.0001	

Table 3.8: Statistics for v_6

Parameter	Values		
	v_7	source table	segment
query_rate	100		
query_size	1		
update_rate		10	
update_size		1	
tuple_num		10000	
tuple_size		100	
levels	2		
fan-out			5
join_ratio			0.001
select_ratio			0.2
proj_ratio			0.4
aggr_ratio			0.1
block_size	8K	8K	
disk_cost	1	1	
net_cost	0	0.0001	

Table 3.9: Statistics for v_7

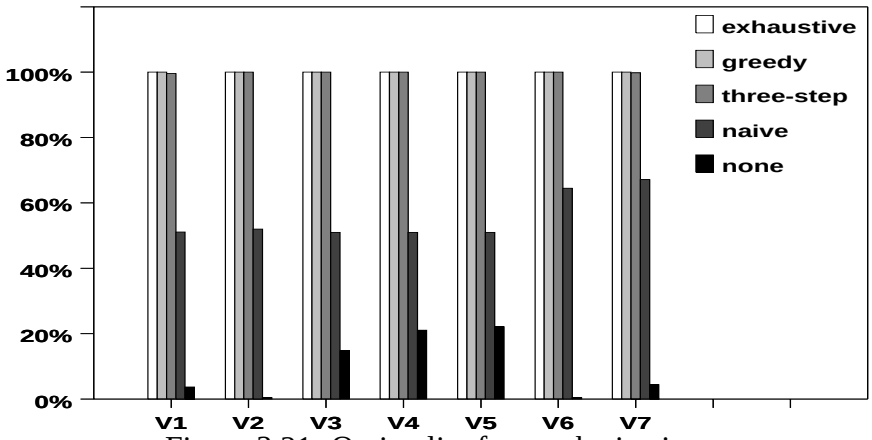


Figure 3.21: Optimality for synthetic views

fraction of the running time required by the exhaustive algorithm, and three-step is much faster than greedy. Another interesting result is that algorithm *none* performs better when the query/update ratio is lower (experiments v_3–v_5). However, we should not infer that the benefit of auxiliary views is primarily for lineage tracing. In fact, in experiment v_5 the query/update ratio is set to 0 (indicating view maintenance only), and we still see significant benefit to using auxiliary views.

Next, we consider in more detail how the running times of the exhaustive, greedy, and

	exhaustive	greedy	three-step	naive	none
v_1	2411.68	3.18	0.67	0.05	0.07
v_2	2414.86	3.18	0.99	0.03	0.03
v_3	2455.74	3.28	0.71	0.08	0.03
v_4	2493.06	3.17	0.63	0.03	0.05
v_5	2376.53	3.93	0.64	0.03	0.03
v_6	757.49	0.99	0.17	0.02	0.02
v_7		156.36	35.29	0.33	0.21

Figure 3.22: Running time for synthetic views (sec)

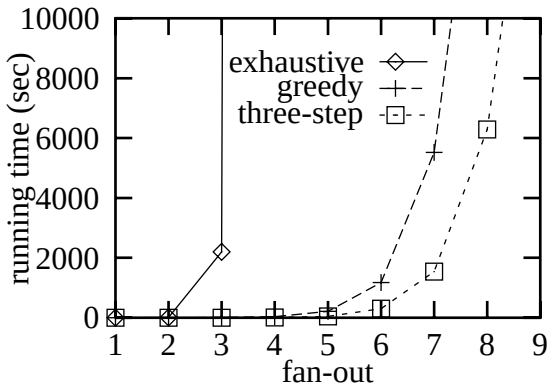


Figure 3.23: Running time vs. fan-out

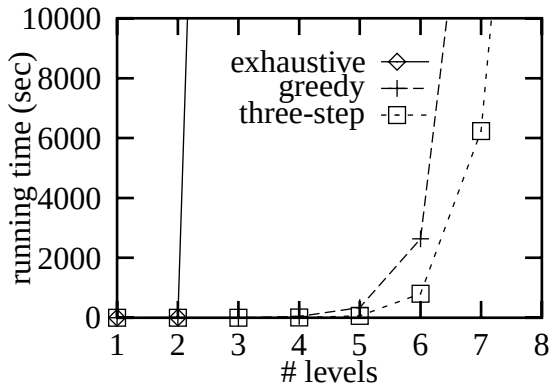


Figure 3.24: Running time vs. # levels

three-step algorithms are affected by view complexity. In Figure 3.23, we consider views where we fix the number of levels at 2 and increase the fan-out from 1 to 9. The exhaustive, greedy, and three-step algorithms become prohibitive when the fan-out exceeds 3, 7, and 8, respectively. We see similar behavior in Figure 3.24, where we fix the fan-out at 2 and increase the number of levels from 1 to 8.

3.7.3 When Greedy and Three-Step Fail

The greedy and three-step algorithms select the optimal (or in one case very near to optimal) auxiliary view set in all of the experiments reported in Sections 3.7.1 and 3.7.2. However, there are cases in which these algorithms fail to pick an optimal or even near-optimal answer.

Consider a simple view definition v and the auxiliary views that are considered for v in

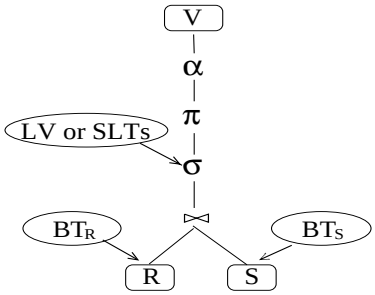


Figure 3.25: View structure of v_8 and v_9

Parameter	Values			
	v_8	R	S	segment 1
query_rate	100			
query_size	1			
update_rate		0	0	
update_size		0	0	
tuple_num		100000	100000	
tuple_size		1000	1000	
fan-out				2
join_ratio				0.001
select_ratio				0.2
proj_ratio				0.6
aggr_ratio				0.2
block_size	8K	8K	8K	
disk_cost	1	1	1	
net_cost	0	0.01	0.01	

Table 3.10: Statistics for v_8

Parameter	Values			
	v_9	R	S	segment 1
query_rate	100			
query_size	1			
update_rate		0	10	
update_size		0	20	
tuple_num		10000	1000000	
tuple_size		1000	1000	
fan-out				2
join_ratio				0.00003
select_ratio				0.2
proj_ratio				0.6
aggr_ratio				0.2
block_size	8K	8K	8K	
disk_cost	1	1	1	
net_cost	0	0.0001	0.001	

Table 3.11: Statistics for v_9

Figure 3.25. By setting the data statistics (Table 3.2) to different values, we can vary the costs and benefits of the four different auxiliary views. We have set two different configurations, which we call v_8 and v_9, both based on the view in Figure 3.25. The complete set of statistical settings for these two experiments is summarized in Tables 3.10 and 3.11. In particular, we set the source table network costs for v_8 to be much higher than for v_9, and we set the join ratio of v_8 higher (less selective) than v_9. The optimal set of auxiliary views for v_8 is $\{BT_R, BT_S\}$, and the optimal set for v_9 is $\{LV\}$.

Figure 3.26 plots the optimality of all five algorithms on views v_8 and v_9. In particular, the greedy algorithm performs extremely poorly on v_8 (selecting $\{LV\}$ instead of $\{BT_R, BT_S\}$), while the three-step algorithm misses the optimal solution for v_9 (selecting $\{BT_R, BT_S\}$ instead of $\{LV\}$). However, as suggested in Section 3.6, if we use a

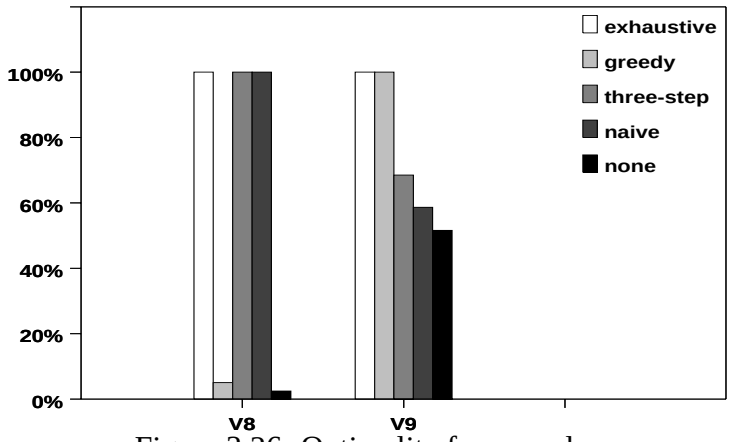


Figure 3.26: Optimality for v_8 and v_9

combined algorithm that runs both greedy and three-step and then selects the lower-cost solution, we will select the optimal view set for both v_8 and v_9.

3.8 Lineage Tracing System Implementation

Based on the techniques introduced so far in this thesis, we implemented a prototype lineage tracing system within the WHIPS [HGMW+95] data warehousing prototype at Stanford. In this section, we first describe the architecture of our lineage tracing system, then introduce the system's Web user interface by walking through a lineage tracing demonstration.

3.8.1 System Architecture

We have shown the basic architecture of WHIPS in Figure 1.1 of Chapter 1. Figure 3.27 illustrates how our lineage tracing system extends this architecture. When a view is defined through the *View Specifier*, the view definition is transformed into its normal form (Section 2.5.1). If the view specification indicates that the view should be traceable, the *Auxiliary View Generator* (*AVGen*) automatically generates auxiliary view definitions based on the user view definition. In our implementation, we store a lineage view (LV) for each ASPJ segment directly defined on a source table, and an aggregate view (AG) for each intermediate aggregate node in the view definition tree. The *Maintenance Procedure Generator*

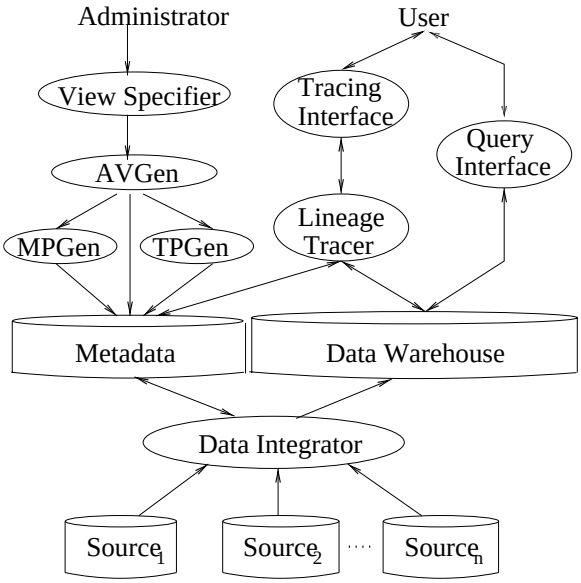


Figure 3.27: The lineage tracing system

(*MPGen*) and the *Tracing Procedure Generator* (*TPGen*) then generate the maintenance procedures (see [HGMW+95]) and lineage tracing queries (Section 3.2) for the user view as well as its auxiliary views, and store them as part of the *Metadata*.

When the warehouse is loaded, the *Data Integrator* also populates the auxiliary view tables in the warehouse based on their definitions in the metadata store. In addition, each time a source table is changed, the data integrator calls the corresponding maintenance procedures to compute the changes to the user view and the auxiliary views. As mentioned earlier in the Chapter, the auxiliary views we store for lineage tracing can also be used in the maintenance procedure for the user view as shown in Section 3.2.

Finally, when a user issues a lineage request through the *Tracing Interface*, the *Lineage Tracer* is activated and calls the appropriate sequence of tracing queries recursively (Section 2.5). Because the auxiliary views we store ensure self-tracability, we never need to query the source tables for lineage tracing. The lineage results are returned to the user as a list of tables. If the user further requests to see the *derivation process*, the lineage tracer combines the lineage results and the view definition to generate a *derivation tree* for the user, showing how the traced view tuple is derived from its lineage tuples step-by-step along the view definition tree. We discuss and illustrate the tracing interface in detail in Section 3.8.2.

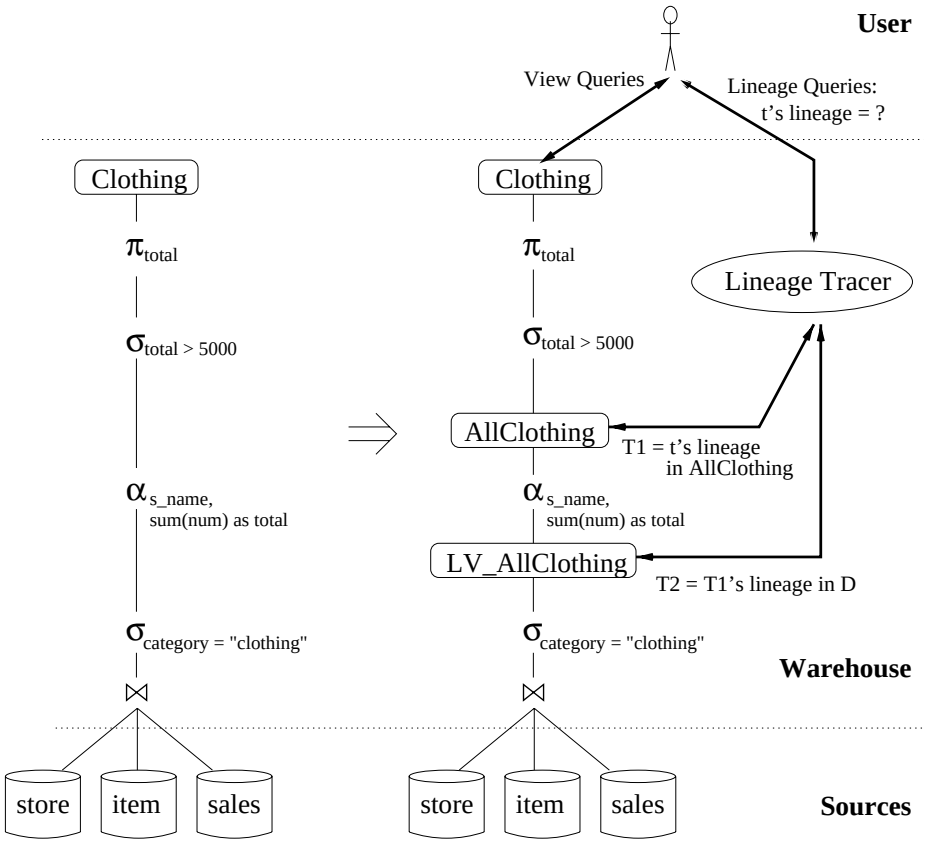


Figure 3.28: Tracing process

Figure 3.28 shows the auxiliary views and the tracing process for our example view `Clothing` from Section 3.1. The query tree on the left side of Figure 3.28 is the original definition of view `Clothing`. One auxiliary view `AllClothing` maintains intermediate aggregation results. Furthermore, to trace tuples in `AllClothing`, a second auxiliary view, the lineage view `LV_AllClothing`, is maintained. The final set of materialized views are:

$$\texttt{Clothing} = \pi_{\texttt{total}}(\sigma_{\texttt{total}>5000}(\texttt{AllClothing}))$$

$$\texttt{AllClothing} = \alpha_{\texttt{s\_name,sum(num) as total}}(\texttt{LV\_AllClothing})$$

$$\texttt{LV\_AllClothing} = \sigma_{\texttt{category}=\texttt{'clothing'}}(\texttt{store} \bowtie \texttt{item} \bowtie \texttt{sales})$$

Each view is computed and maintained based on the views (or source tables) directly

beneath it using pregenerated incremental view maintenance procedures. Bold arrows on the right side of Figure 3.28 show the query and answer data flows. Ordinary view queries are sent to the view `Clothing`, while lineage queries are sent to the *Lineage Tracer*. When receiving a request for the lineage of a tuple t in `Clothing`, the Lineage Tracer first issues a lineage tracing query to auxiliary view `AllClothing` to obtain t's derivation T_1 in `AllClothing` as specified in Theorem 2.4.3. It then queries `LV_AllClothing` for the lineage T_2 of T_1 in D as specified in Section 3.2. T_2 is t's lineage in D, and is returned to the user through the Lineage Tracer.

3.8.2 Lineage Tracing User Interface

In addition to core support for lineage tracing, we developed a Web-based user interface to our lineage tracing system on top of WHIPS. We demonstrated our lineage tracing system in [CW00a]. The demonstration was based on a synthetic financial data warehouse derived from three tables at three different sources:

```
s1.portfolio(tickerS1,date1,price,bought,owned)
s2.daily(tickerS2,date2,high,low,closep,volume)
s3.earnings(tickerS3,earnings)
```

Table `portfolio` at source 1 contains all customer stock purchases including the buying date, price, shares bought, and shares owned (in the case of a split). Table `daily` at source 2 contains the daily price information of each stock, including the high, low, closing prices. Table `earnings` at source 3 contains the latest monthly earnings per share of each stock. Figure 3.29 shows a screen-shot with the three source tables *reported* to the data warehouse. (Once a source table is reported to the data warehouse, we can define materialized views over that source table at the warehouse. WHIPS will populate the warehouse view contents based on the source table contents, and will maintain the warehouse views when the source tables change.) Figure 3.30 shows sample contents of source table `s1.portfolio`. Note that we use a small data set for illustration purposes.

Over the three source tables, we define three warehouse views as indicated in Figure 3.31. View `Gaining` is defined over source tables `s1.portfolio` and `s2.daily`, returning all the stocks that are gaining money for the customer. View `PriceEarnings` computes the

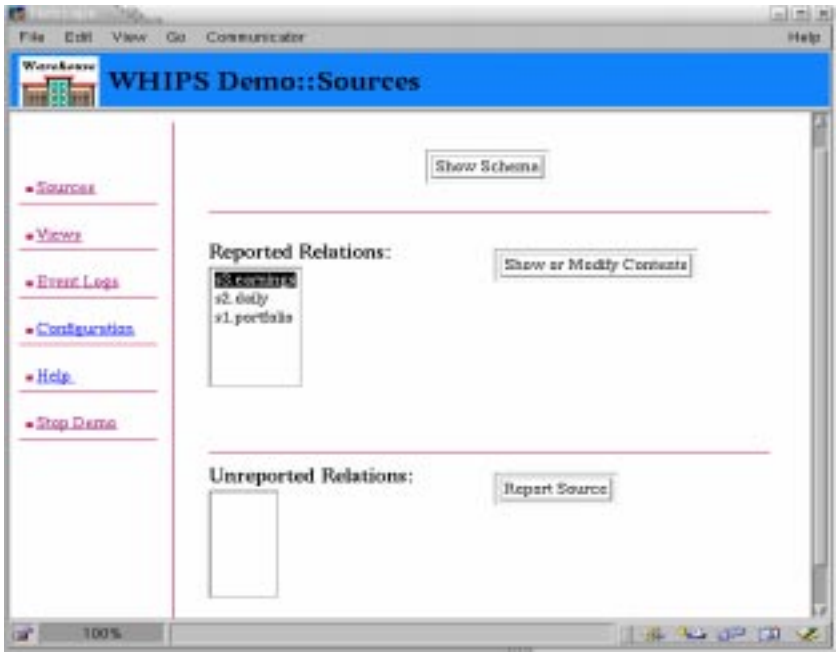


Figure 3.29: Data sources

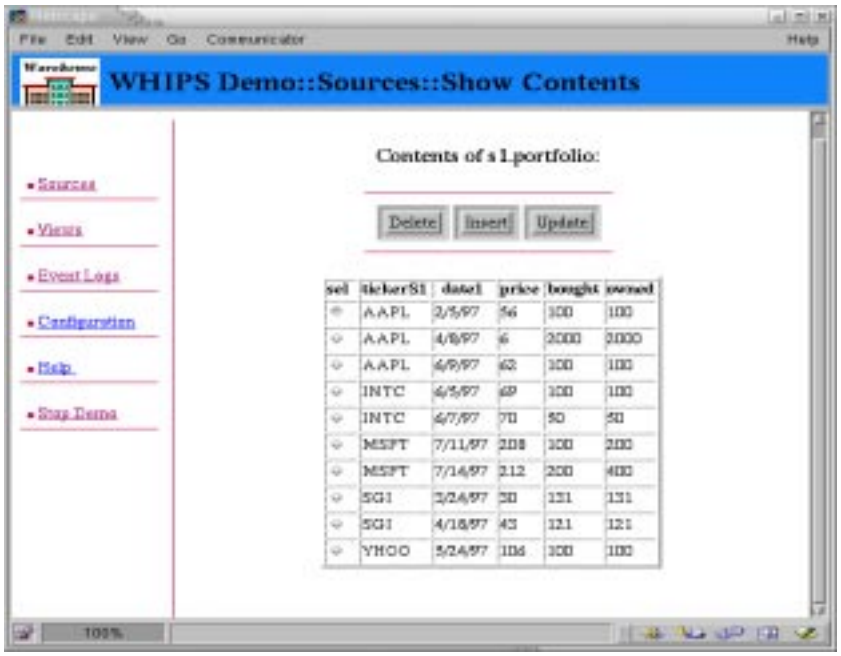


Figure 3.30: Source table `Portfolio`

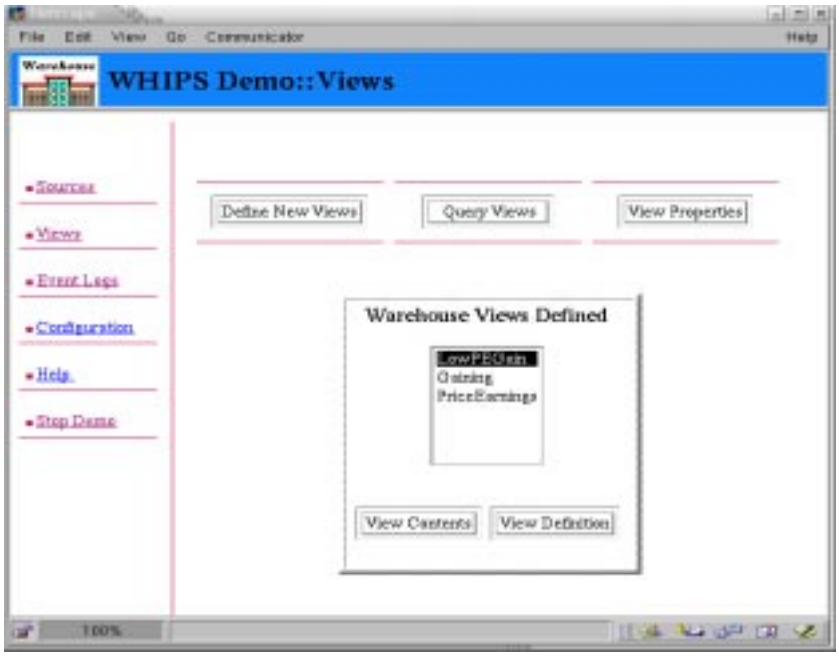


Figure 3.31: User views

latest price-earnings ratio for each stock. View `LowPEGain` is defined over views `Gaining` and `PriceEarnings` in the warehouse, and returns the gaining stocks with a price-earnings ratio lower than 100. The SQL definitions of the three views are as follows, where "trace remote" indicates that the view should enable lineage tracing.

```
CREATE VIEW Gaining TRACE REMOTE AS
SELECT P.tickerS1 AS tickerS1, D.closep AS closep,
       SUM(P.owned) AS totalShares,
       SUM(D.closep * P.owned - P.price * P.bought) AS gain
FROM   s2.daily D, s1.portfolio P
WHERE  D.tickerS2 = P.tickerS1 AND D.date2 = '10/17/97'
GROUP BY P.tickerS1, D.closep
HAVING SUM(D.closep * P.owned - P.price * P.bought) > 0;

CREATE VIEW PriceEarnings TRACE REMOTE AS
SELECT D.tickerS2 AS tickerS2,
       (D.closep / E.earnings) AS pe
FROM   s2.daily D, s3.earnings E
WHERE  D.tickerS2 = E.tickerS3 AND D.date2 = '10/17/97';
```

Figure 3.32: View `LowPEGain`

```
CREATE VIEW LowPEGain TRACE REMOTE AS
SELECT G.tickerS1 AS tickerS, G.gain
FROM   Warehouse.Gaining G, Warehouse.PriceEarnings P
WHERE  G.tickerS1 = P.tickerS2 AND P.pe < 100;
```

Figure 3.32 shows the contents of view `LowPEGain`. Through this interface, we can select any tuple in the view table, choose to trace its data lineage one level down the *view hierarchy*, or trace its data lineage to the source tables.

Suppose we want to trace the lineage of view tuple $\langle$`MSFT`, `11800`$\rangle$ to learn why MSFT stock is so profitable, and we decide to trace lineage one level at a time. After pressing "Trace One Level", the system applies the tracing procedure for `LowPEGain` to views `Gaining` and `PriceEarnings` over which `LowPEGain` is defined, and returns the lineage results in these view tables, as shown in Figure 3.33. We can further inspect the detailed process through which the selected view tuple is derived from its lineage tuples. Figure 3.34 shows the *derivation tree*. We can then trace the lineage of tuple $\langle$`MSFT`, `125`, `600`, `11800`$\rangle$ in view `Gaining` to the next level below; in this case, to the source tables. The result including the derivation tree is shown in Figure 3.35.

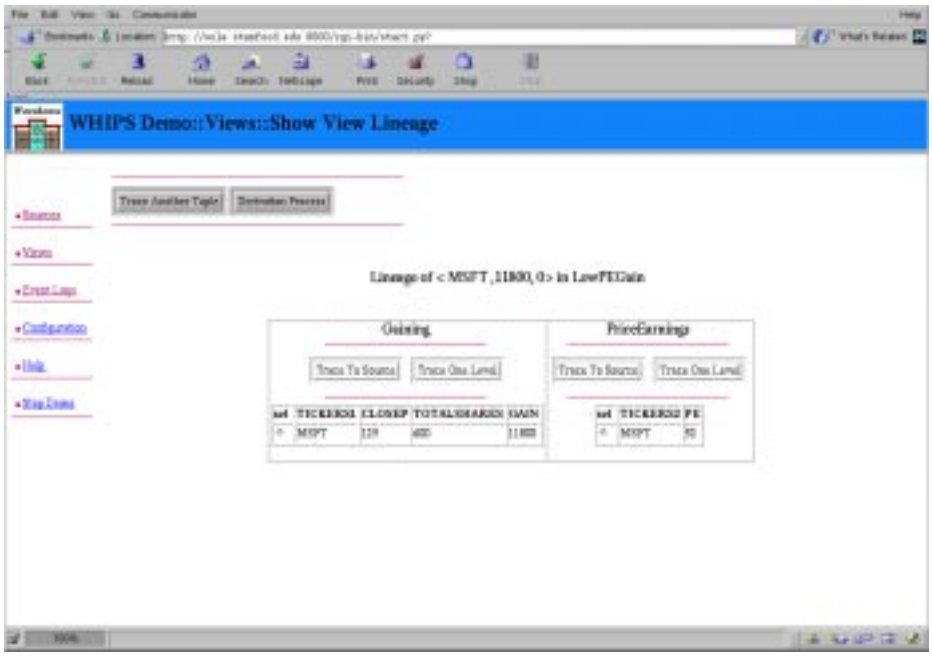


Figure 3.33: Lineage of ⟨11800⟩

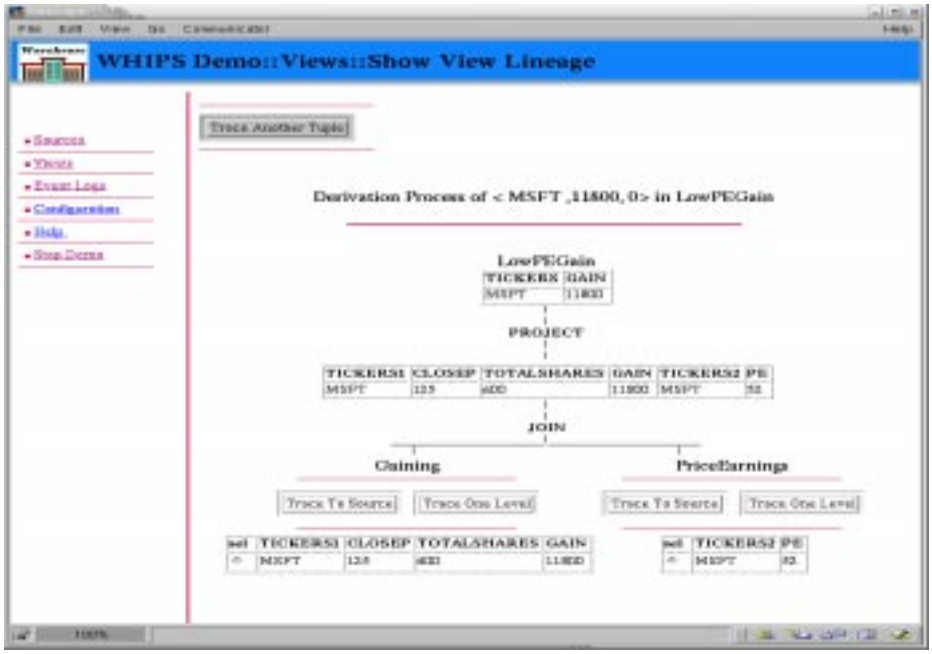


Figure 3.34: Derivation tree for ⟨11800⟩

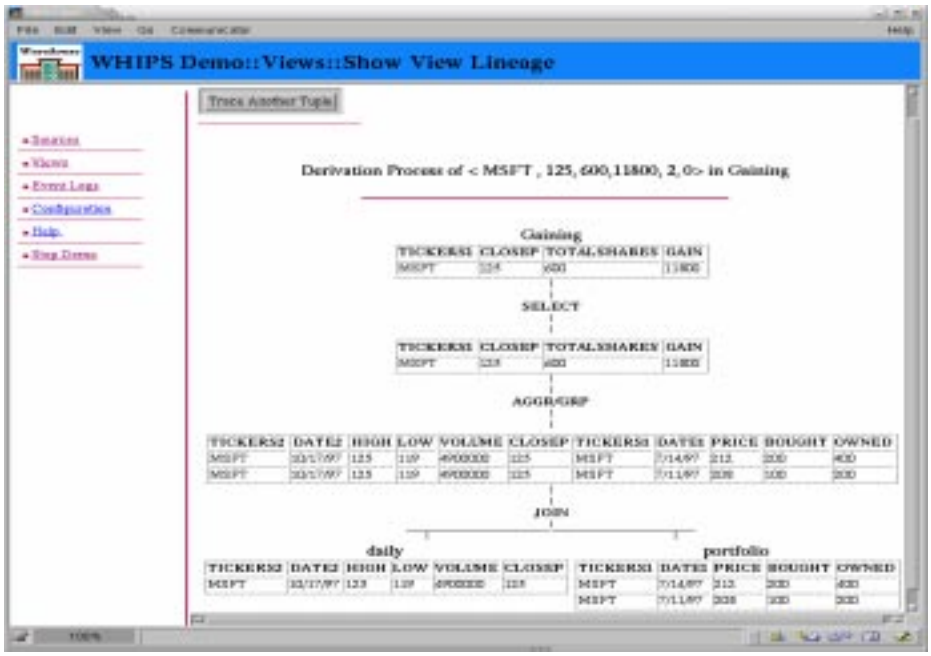


Figure 3.35: Derivation tree for $\langle$MSFT, 125, 600, 11800$\rangle$

We could instead choose to trace the lineage of tuple $\langle$MSFT, 11800$\rangle$ in view LowPEGain immediately all the way to the source tables. The system then recursively applies tracing procedures down the view hierarchy until it reaches the source tables. Figure 3.36 shows the result, including the entire derivation tree.

For the three user views in our example, the system stores three additional auxiliary views: an intermediate aggregate view (AuxView0) and a lineage view (AuxView1) for Gaining, and a lineage view (AuxView2) for PriceEarnings. See Figure 3.37. When the source tables change, these auxiliary views are maintained in a consistent fashion with the user views in order to guarrantee consistent lineage results. All three auxiliary views we store to improve lineage tracing also help the maintenance of the user views.

3.8.3 Implementation Experience

Our lineage tracing system was built as part of the WHIPS data warehousing prototype, on top of a commercial database management system (DBMS) that does not provide any lineage tracing functionality. WHIPS, and our lineage tracing system, operate as middleware

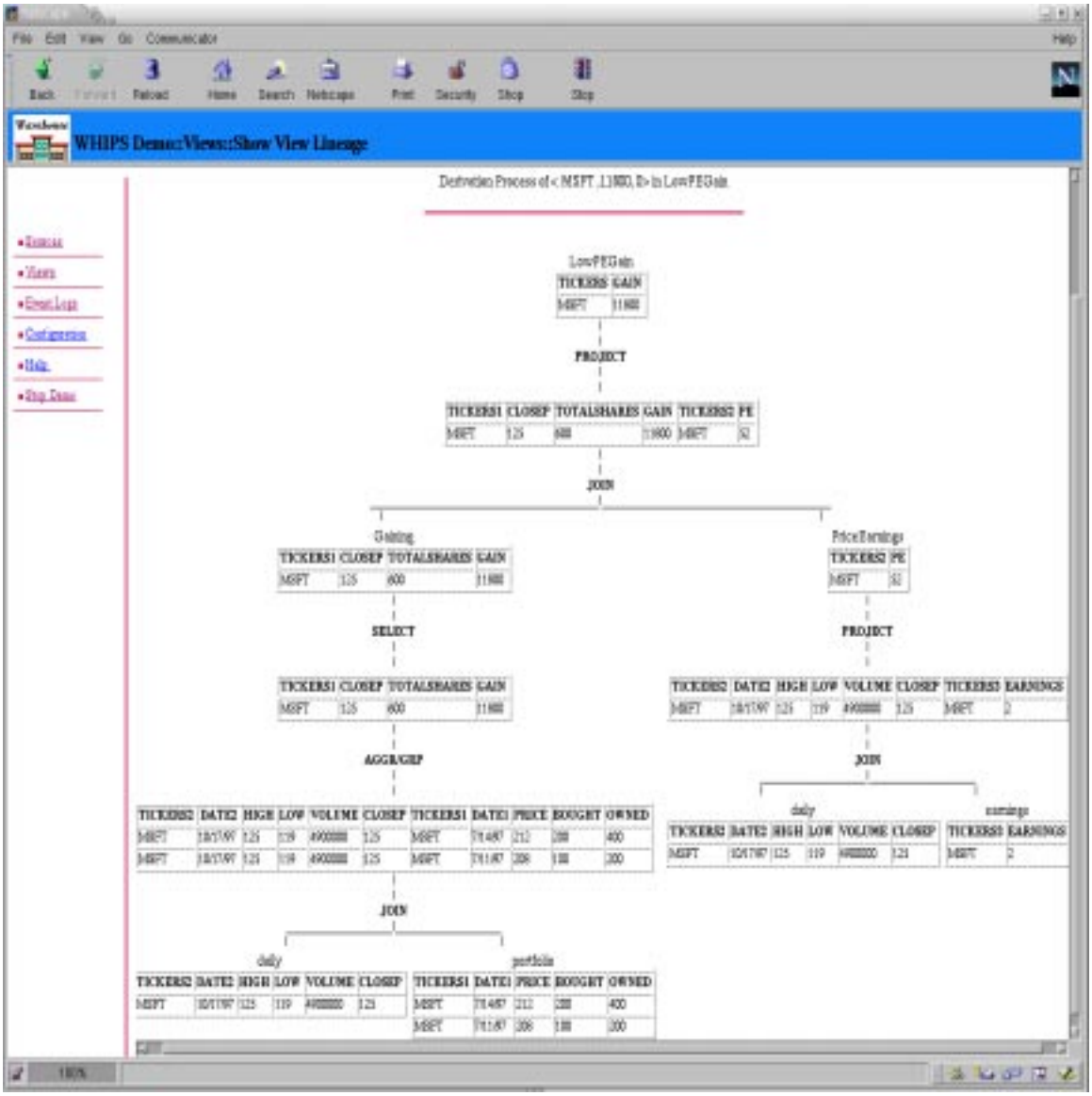


Figure 3.36: Entire derivation tree

between applications and the backend DBMS, instead of as an extension to any specific DBMS. As a result, the implemented system is almost independent of the backend DBMS. The lineage tracing system does, however, require that the backend DBMS supports stored procedures, since both lineage tracing and view maintenance are implemented using stored procedures.

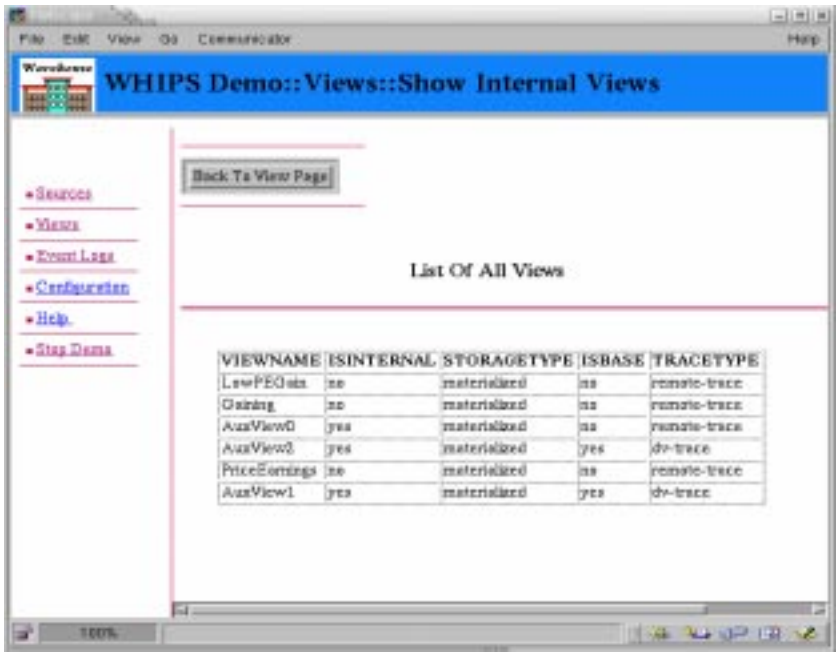


Figure 3.37: Auxiliary views

3.9 Related Work

Previous work most closely related to the auxiliary view problem discussed in this chapter falls into two categories: selecting views to materialize in order to minimize query costs, e.g., [HRU96, Gup97], and selecting auxiliary views to materialize in order to minimize the cost of maintaining given primary views, e.g., [LQA97, RSS96].

[HRU96] proposes a greedy algorithm for selecting auxiliary views to materialize, with the goal of minimizing the cost of queries over aggregate views given certain constraints such as the maximum number of views that can be materialized. The work considers data-cube views only, and can make certain simplifying assumptions based on this restriction. [Gup97] extends the work in [HRU96] to general relational views, and proves that the auxiliary view selection problem under maintenance cost constraints is NP-hard.

[RSS96] proposes an exhaustive algorithm for selecting auxiliary views to optimize view maintenance, and suggests simple search space pruning strategies when the view is too complex for exhaustive search. [LQA97] presents an A* algorithm for selecting auxiliary views and indexes on different join combinations for SPJ view maintenance. Both [RSS96]

and [LQA97] consider a single algorithm for selecting auxiliary views (and indexes in the case of [LQA97]), designed specifically for optimizing view maintenance. They consider as potential auxiliary views all nodes in all possible relevant query plans, making the search space doubly exponential in the view definition size.

Our work differs from the previous work discussed above in several ways:

- Unlike all previous work, we consider lineage tracing as well as view maintenance costs when selecting auxiliary views to materialize.

- Instead of considering a doubly exponential search space of auxiliary views (as in [HRU96, Gup97, LQA97, RSS96]), we consider a fixed set of auxiliary view schemes for SPJ views, then explore a relatively small (although still exponential) search space for ASPJ views based on our view definition normal form.

- For ASPJ views we propose several different auxiliary view selection algorithms, as opposed to a single algorithm, and we compare the performance of our algorithms (both running time and quality of solution) through experiments.

As discussed in Section 3.1, the previously studied problem of *self-maintainability* [QGMW96] determines which auxiliary views need to be added so that sources never need to be accessed for view maintenance. This problem differs from ours since we consider lineage tracing as well, and we do not prohibit accessing sources. However, we do use techniques from self-maintainability, as discussed in Sections 3.1 and 3.2. Also, we use existing incremental view maintenance techniques [GMS93, ZGMHW95] to estimate the cost of different auxiliary schemes and to maintain views in the WHIPS system.

3.10 Chapter Summary

In this chapter, we first introduced a family of schemes for storing auxiliary views for SPJ primary views that enable and improve the performance of lineage tracing and view maintenance in a distributed multi-source warehousing environment. We compared the lineage tracing and warehouse maintenance performance of the schemes through simulations. Our

performance studies show that different schemes offer different advantages and thus are suitable for different settings.

We then examined the problem of selecting auxiliary views to materialize in a data warehouse in order to reduce the overall lineage tracing and view maintenance cost for complex ASPJ views. We used our ASPJ normal form from Chapter 2 to define an initial search space of potentially beneficial auxiliary views, and presented four algorithms for exploring the search space and selecting a set of auxiliary views: *exhaustive*, *greedy*, *three-step*, and *naive*. We compared the optimality and running time of the algorithms using experiments based on the TPC-D benchmark, as well as on a variety of synthetic views and statistics. Our experiments indicate that in terms of running time and optimality, the three-step algorithm appears to be the best, although the running time of the greedy algorithm probably also is fast enough in practice for most complex warehouse views. (The exhaustive algorithm, on the other hand, becomes intractable quite quickly.) Both the greedy and three-step algorithms find the optimal auxiliary view set in most cases, although we have shown (complementary) situations in which either one algorithm or the other performs poorly. Our experiments also illustrate that even a naive selection of auxiliary views reduces overall cost dramatically in most cases, underscoring the importance of materializing auxiliary views for the dual purposes of view maintenance and lineage tracing in a warehousing environment.

Although we have presented our work in the context of selecting an auxiliary view set for a single primary warehouse view, our approach extends easily to considering multiple primary views together. Then the cost of an auxiliary view may be "shared" if the auxiliary view is beneficial to view maintenance or lineage tracing for more than one primary view. Furthermore, although we have studied an environment in which both view maintenance and lineage tracing are important, if only one type of activity is present our algorithms remain applicable.

Finally, we described our implementation of a lineage tracing system within the WHIPS [HGMW+95] data warehousing prototype at Stanford, based on the results presented in this chapter and Chapter 2.

Chapter 4

View Update Using Data Lineage

In Chapters 1–3 we discussed relational views as a technique for specifying and maintaining the contents of data warehouses. Even in the context of conventional relational database systems, data management through user-defined views is a standard and important feature [Sto75]. However, to make views truly first class, we must not only allow users to query and browse the database through views, as is typically supported in most DBMSs today, but we should also allow database updates through views. Note again we are now referring to standard database management systems. Data warehouses typically do not support user updates [CD97, LW95]. A number of problems arise when updating a database through views, yielding the well-known and well-studied *view update problem* [BS81, Cle78, DB82, KU84, LS91, Mas84, Shu96, Sto75, Tom94]. The core of the view update problem is to translate updates against views to updates against the base tables that the view is defined on.

As mentioned in Chapter 1, data lineage when considered independently of data warehouses is closely related to the view update problem for deletions. In this chapter, we study this relationship and use techniques based on data lineage for translating deletions against view tuples (referred to as *view deletions*) into deletions against base tables (referred to as *base deletions*), in a conventional database setting. Specifically, we provide a fully automatic algorithm that uses only the view definition at compile-time and the base data at view-update time to find a translation that is guaranteed to be *exact* (side-effect free) whenever an exact translation exists.

In Chapters 2 and 3, we considered warehouses consisting of *materialized* relational views. A view is materialized if a table is created for it, its contents are computed from the base tables over which it is defined, and it is kept up to date (maintained) as the base tables change. Conventional DBMSs more commonly support *virtual* views. A virtual view consists solely of the view definition. When a query references a virtual view, before processing the query is rewritten based on the view definition into an equivalent query that references base tables only. The work in this chapter applies to both kinds of views since actual view contents are never needed by our algorithms.

The remainder of the chapter proceeds as follows. Section 4.1 introduces the view update problem and gives a running example that will be used throughout the chapter. Section 4.2 then formalizes the view update problem for deletions. In Section 4.3, we explore the relationship between data lineage and view deletions, which is used as the basis for our translation algorithm for single-tuple deletions presented in Section 4.4. Sections 4.5 and 4.6 extend our algorithm to handle sets of deletions and deletions specified by a selection condition. We have implemented our translation algorithms, and empirical results are presented in Section 4.7. Section 4.8 surveys related work, and Section 4.9 concludes the chapter. The work presented in this chapter appeared originally in [CW01].

4.1 Introduction and Running Example

Let us assume a base relational database D and a view V defined over tables in D. In this chapter we focus our attention to SPJ views as defined originally in Chapter 2 (Section 2.2). The typical steps involved in a view update are:

(1) The user requests an update U_V to view V.

(2) Some process, referred to as *view update translation*, takes U_V and produces an update U_D to the underlying base data D.

(3) U_D is applied to the base database. If the view is materialized (as opposed to virtual), then the view is modified to reflect the base data update U_D.

Many previous approaches, e.g., [Cle78, Kel86, LS91, RS79], require participation from the view definer and/or the view updater in specifying view update translations. In this

chapter, we use techniques based on data lineage to devise a fully automatic algorithm that translates view deletions using only the view definition at compile-time and the base data at view-update time.

Referring back to the steps outlined above for view update, when the base data update U_D is applied in step (3), the view is modified, either logically or physically, as a result. Let U'_V denote the view modifications induced by database update U_D. U'_V should certainly contain the user's requested view update U_V from step (1), in which case U_D is called a *correct* view update translation. Even better, U'_V should be identical to U_V, in which case U_D is called an *exact* view update translation. Surprisingly, most previous work on the view update problem does not consider exactness, e.g., [Mas84, LS91], or guarantees exactness only for a restricted class of select-project-join (SPJ) views, e.g., [DB82, Kel85]. Unlike these algorithms, our run-time translation algorithm finds a translation for any view deletion against any SPJ view that is guaranteed to be exact whenever an exact translation exists. Data lineage helps us achieve this result.

There may be more than one translation for a view update—even more than one exact translation—leading to an inherent *ambiguity* in the view update translation process that has been the focus of much previous work, e.g., [BS81, Kel86, LS91]. We do not explore the ambiguity issue in our work. Instead, our goal is to find, as fast as possible and without view definer or user intervention, an exact translation that updates a small number of base tuples. Also we do not consider constraints on the base tables in our work: we assume that all deletions on base tables are valid. Some discussion on how base table constraints can affect the view update translation process can be found in [DB82, Kel85]. Finally, we do not consider insertions or modifications to views. After some exploratory work it is our belief that our work in data lineage is applicable only to view deletions, and by extension to a portion of the view modification problem (complemented by a solution to the insertion "half" of the problem). For a complete view update translation package, our deletion algorithm could be combined with any previous algorithm for translating view insertions, e.g., [DB82], although as noted earlier these algorithms do not guarantee exactness.

In summary, we focus on the automation and exactness aspects of the view update translation process, and present a fully automatic algorithm that uses techniques based on data lineage for translating deletions against SPJ views into deletions against the underlying

UserGroup

user	group
john	sale
lisa	mkt
lisa	eng1
lisa	eng6
joe	eng1
joe	eng2
mary	eng2
mary	eng4
ted	eng4
ted	eng5

GroupAccess

group	file
sale	f1
mkt	f2
eng1	f3
eng1	f4
eng2	f3
eng2	f4
eng4	f5
eng5	f5
eng5	f6
eng6	f3

Figure 4.1: Base tables

V

user	file
lisa	f3
lisa	f4
joe	f3
joe	f4
mary	f3
mary	f4
mary	f5
ted	f5
ted	f6

Figure 4.2: View contents

database. Our algorithm yields translations that are guaranteed to be exact whenever an exact translation exists.

4.1.1 Running Example

Consider a simple user access control database with two tables: $\texttt{UserGroup}(\texttt{user}, \texttt{group})$ and $\texttt{GroupAccess}(\texttt{group}, \texttt{file})$. The `UserGroup` table contains the groups that each user belongs to, and the `GroupAccess` table lists the files accessible by each group. Figure 4.1 shows a small example database.

Consider an SPJ view V defined in relational algebra as follows:

$$V = \pi_{\texttt{user},\texttt{file}}(\sigma_{\texttt{group}=\texttt{'eng\%'}}(\texttt{UserGroup} \bowtie \texttt{GroupAccess}))$$

V contains information about all files accessible by users in the "eng" (engineering) groups; its contents over our sample base data are shown in Figure 4.2. We consider six example view deletions, listed in Table 4.1. All translations shown in the table are exact except for Example 4, which has no exact translation. The remainder of this chapter will show how we can use data lineage techniques to find the translations shown in this example, and more generally to translate any deletions against any SPJ view.

Example #	View deletions	Translation
1	delete $\langle$`ted`, `f5`$\rangle$ from V	delete $\langle$`ted`, `eng4`$\rangle$ from `UserGroup` delete $\langle$`eng5`, `f5`$\rangle$ from `GroupAccess`
2	delete $\langle$`lisa`, `f3`$\rangle$ from V	delete $\langle$`lisa`, `eng6`$\rangle$ from `UserGroup` delete $\langle$`eng1`, `f3`$\rangle$ from `GroupAccess`
3	delete $\langle$`joe`, `f3`$\rangle$ from V	delete $\langle$`joe`, `eng2`$\rangle$ from `UserGroup` delete $\langle$`eng1`, `f3`$\rangle$ from `GroupAccess`
4	delete $\langle$`joe`, `f4`$\rangle$ from V	delete $\langle$`joe`, `eng1`$\rangle$, $\langle$`joe`, `eng2`$\rangle$ from `UserGroup`
5	delete $\langle$`lisa`, `f4`$\rangle$, $\langle$`joe`, `f4`$\rangle$ from V	delete $\langle$`eng1`, `f4`$\rangle$ from `GroupAccess`
6	delete from V where `user` $= '$`ted`$'$	delete $\langle$`ted`, `eng4`$\rangle$, $\langle$`ted`, `eng5`$\rangle$ from `UserGroup` delete $\langle$`eng5`, `f5`$\rangle$, $\langle$`eng5`, `f6`$\rangle$ from `GroupAccess`

Table 4.1: Example view deletions and translations

4.2 The View Update Problem for Deletions

We now formally define the view update translation problem for deletions. In this chapter we consider only select-project-join (SPJ) views, as defined originally in Section 2.2. As discussed in Section 2.4, every SPJ view V can be written as

$$V = \pi_A(\sigma_C(R_1 \bowtie \cdots \bowtie R_n))$$

where $R_1, \ldots, R_n$ are base tables, C is a boolean condition, and A is a list of projected attributes. We represent deletions on a base table R as ∇R and deletions on a database $D = R_1, \ldots, R_n$ as $\nabla D = \nabla R_1, \ldots, \nabla R_n$. We assume set semantics (no duplicates) and no base table constraints throughout the chapter.

Given a view V, we use $-t$ to denote a request to delete a single view tuple $t \in V$. (In Section 4.5 we generalize to view update requests that are sets of deleted tuples, as in Example 5 of Table 4.1, and in Section 4.6 we handle view update requests specified by a selection condition, as in Example 6 of Table 4.1.) Note that in this chapter we will use "V" generically to represent the name of a view, its definition, and sometimes its (virtual or actual) contents. We now formalize the concept of a *view deletion translation*. Concrete examples will be seen in Section 4.3.

Definition 4.2.1 (View Deletion Translation) Given a view V and a deletion request $-t$ where $t \in V$, we say that database deletion ∇D is a *translation for* $-t$ if ∇D causes the deletion of t from V when applied to database D. More formally, let V' be the new view

based on the updated database $D - \nabla D$, and let $\nabla V = V - V'$ be the actual deleted view tuples, which we call the *view deletions induced by* ∇D. We say that ∇D is a translation for $-t$ if $\{t\} \subseteq \nabla V$. If $\nabla V = \{t\}$ then ∇D is an *exact* translation for $-t$. Otherwise, ∇D is *inexact* and causes *side-effect* (or *extra deletions*) $E = \nabla V - \{t\}$. □

For SPJ views, when we perform a deletion ∇D on the base data, the changes to the view induced by ∇D are always deletions, never insertions or modifications. However, when we translate a deletion request $-t$ into updates on the base data, it is not necessary to produce only deletions. For example, in translating $-t$ we might choose to delete some existing base tuples, then insert some new ones to compensate for view side-effects caused by the deletions. We do not consider the compensation approach in our work, focusing only on the pure deletions-to-deletions translation problem. In considering the more general problem we found that cases where view deletions benefit from translations encompassing all types of base data updates are very rare, although such cases do exist.

4.3 Relationship Between Data Lineage and View Deletions

In this section, we formalize the relationship between data lineage and view deletions, to lay the groundwork for our deletion translation algorithm to be specified in Section 4.4. We continue to focus on single-tuple deletions, and we will extend our results to deletions of a view tuple set and deletions specified by a selection condition in Sections 4.5 and 4.6, respectively.

First, let us briefly review the lineage definition for SPJ views and introduce a new concept of *exclusive lineage* for view update. Consider an SPJ view $V = \pi_A(\sigma_C(R_1 \bowtie \cdots \bowtie R_n))$ over database D and a tuple $t \in V$. From Definition 2.3.9 in Chapter 2, a tuple $t_i \in R_i$ belongs to t's lineage R_i^* in R_i, which we also refer to as t's ith *lineage branch*, if and only if:

$$\{t\} \subseteq \pi_A(\sigma_C(R_1 \bowtie \cdots \bowtie R_{i-1} \bowtie \{t_i\} \bowtie R_{i+1} \bowtie \cdots \bowtie R_n))$$

Example 4.3.1 Recall view V from Section 4.1, containing user file access control information. Given tuple $t = \langle \texttt{ted}, \texttt{f5} \rangle$ in V, based on Definition 2.3.9 t's lineage is:

$$lineage(t, V, D) = \langle \texttt{UserGroup}^* = \{\langle \texttt{ted}, \texttt{eng4} \rangle, \langle \texttt{ted}, \texttt{eng5} \rangle\},$$
$$\texttt{GroupAccess}^* = \{\langle \texttt{eng4}, \texttt{f5} \rangle, \langle \texttt{eng5}, \texttt{f5} \rangle\}\rangle$$

In this chapter, we modify our formal lineage notation $v_D^{-1}(t)$ from Chapter 2 to the more procedural appearing $lineage(t, V, D)$, representing t's lineage $\langle R_1^*, \ldots, R_n^* \rangle$ in D according to V. □

The *exclusive lineage* of a view tuple t is the set of base tuples that contribute to t and only to t. Formally:

Definition 4.3.2 (Exclusive Lineage) Consider an SPJ view $V = \pi_A(\sigma_C(R_1 \bowtie \cdots \bowtie R_n))$ and a view tuple $t \in V$. A tuple $t_i \in R_i$ belongs to t's *exclusive lineage branch* R_i^{**} if and only if:

$$\{t\} = \pi_A(\sigma_C(R_1 \bowtie \cdots \bowtie R_{i-1} \bowtie \{t_i\} \bowtie R_{i+1} \bowtie \cdots \bowtie R_n))$$

(The difference from Definition 2.3.9 is simply = instead of $\subseteq$.) We denote t's entire exclusive lineage in D as $elineage(t, V, D) = \langle R_1^{**}, \ldots, R_n^{**} \rangle$. Note that $R_i^{**} \subseteq R_i^*$, $i = 1..n$. □

Example 4.3.3 Again, consider view V from Section 4.1 and tuple $t = \langle \texttt{ted}, \texttt{f5} \rangle$ in V. Based on Definition 4.3.2, t's exclusive lineage is:

$$elineage(t, V, D) = \langle \texttt{UserGroup}^{**} = \{\langle \texttt{ted}, \texttt{eng4} \rangle\},\ \texttt{GroupAccess}^{**} = \{\langle \texttt{eng5}, \texttt{f5} \rangle\}\rangle$$

Tuples $\langle \texttt{ted}, \texttt{eng5} \rangle$ and $\langle \texttt{eng4}, \texttt{f5} \rangle$ from t's lineage (Example 4.3.1) are not in t's exclusive lineage, because $\langle \texttt{ted}, \texttt{eng5} \rangle$ also is in the lineage of view tuple $\langle \texttt{ted}, \texttt{f6} \rangle$, and $\langle \texttt{eng4}, \texttt{f5} \rangle$ also is in the lineage of view tuple $\langle \texttt{bob}, \texttt{f5} \rangle$. □

Now consider a view deletion request $-t$ against V. Ideally, deleting t's exclusive lineage (Definition 4.3.2) from D induces the deletion of t from V, as in the following example.

Example 4.3.4 Consider Example 1 from Table 4.1, which deletes tuple $t = \langle \texttt{ted}, \texttt{f5} \rangle$ from V. Example 4.3.3 showed that t's exclusive lineage is:

$$elineage(t, V, D) = \langle \texttt{UserGroup}^{**} = \{\langle \texttt{ted}, \texttt{eng4} \rangle\},\ \texttt{GroupAccess}^{**} = \{\langle \texttt{eng5}, \texttt{f5} \rangle\} \rangle$$

If we delete these tuples from the base tables, t will be deleted from V. Therefore, $\nabla elineage(t, V, D)$ is a translation for $-t$. □

An important property of a translation that deletes the exclusive lineage of the deleted view tuple, as in Example 4.3.4, is that the translation is guaranteed to be exact. By Definition 4.3.2 of exclusive lineage, none of the tuples in $R_1^{**}, \ldots, R_n^{**}$ contribute to any view tuples other than t, so translation $\nabla D^{**} = \nabla R_1^{**}, \ldots, \nabla R_n^{**}$ can induce only the deletion of t. Unfortunately, ∇D^{**} is not always a translation, even if $-t$ has an exact translation, as shown by the following example.

Example 4.3.5 Consider Example 2 from Table 4.1, which deletes tuple $t = \langle \texttt{lisa}, \texttt{f3} \rangle$ from V. Based on Definition 4.3.2, t's exclusive lineage is:

$$elineage(t, V, D) = \langle \texttt{UserGroup}^{**} = \{\langle \texttt{lisa}, \texttt{eng6} \rangle\},\ \texttt{GroupAccess}^{**} = \varnothing \rangle$$

Deleting $elineage(t, V, D)$ does not induce the deletion of t. Thus, $\nabla elineage(t, V, D)$ is not a translation for $-t$. However, $-t$ does have an exact translation, namely, delete tuple $\langle \texttt{lisa}, \texttt{eng6} \rangle$ from UserGroup and $\langle \texttt{eng1}, \texttt{f3} \rangle$ from GroupAccess. □

When the exclusive lineage does not provide us with a translation, we can instead consider deleting one branch R_i^* of t's lineage. (We also could consider deleting all branches, but each branch individually always is sufficient to induce the deletion of t.) Unlike ∇D^{**}, any ∇R_i^* is a translation, but it may not be exact even when there is an exact translation, as shown by the following example.

Example 4.3.6 Consider again Example 1 from Table 4.1, which deletes tuple $t = \langle \texttt{ted}, \texttt{f5} \rangle$ from view V. Example 4.3.1 showed that t's lineage is:

$$\begin{aligned} lineage(t, V, D) = \langle & \texttt{UserGroup}^{*} = \{\langle \texttt{ted}, \texttt{eng4} \rangle, \langle \texttt{ted}, \texttt{eng5} \rangle\}, \\ & \texttt{GroupAccess}^{*} = \{\langle \texttt{eng4}, \texttt{f5} \rangle, \langle \texttt{eng5}, \texttt{f5} \rangle\} \rangle \end{aligned}$$

Deleting either lineage branch is a translation for $-t$, but neither translation is exact: $\nabla \texttt{UserGroup}^*$ also induces the deletion of view tuple $\langle \texttt{ted}, \texttt{f6} \rangle$, while $\nabla \texttt{GroupAccess}^*$

also induces the deletion of view tuple $\langle \texttt{mary}, \texttt{f5} \rangle$. Example 4.3.4 illustrated an exact translation for $-t$. □

So far we have seen that ∇D^{**} may provide an exact translation or may not be a translation at all, and ∇R_i^* (for any $i \in 1..n$) provides a translation but it may not be exact. In neither case are we guaranteed to be provided with an exact translation even if one exists. Nor is it the case that when ∇D^{**} fails ∇R_i^* succeeds, or vice-versa. Thus, if neither ∇D^{**} nor ∇R_i^* ($i = 1..n$) provides an exact translation, the brute-force approach is to enumerate all possible subsets of all base relations, checking if deleting those subsets is an exact translation. Fortunately we can reduce the search space considerably using the following theorem, which tells us that if there is an exact translation for $-t$, then there is an exact translation that is contained in t's lineage. We will further reduce the search space with pruning techniques in our algorithm presented in Section 4.4.

Theorem 4.3.7 Consider an SPJ view V over database D and a tuple $t \in V$. If deletion $-t$ has an exact translation, then $-t$ has an exact translation ∇D such that $\nabla D \subseteq lineage(t, V, D)$.[1] □

Proof: The proof relies on the following two facts, which are obvious enough that we omit detailed justification:

- Let $\langle R_1^*, \ldots, R_n^* \rangle = lineage(t, V, D)$. $R_i^* \neq \varnothing$ for $i = 1..n$.
- There exist $t_1 \in R_1, \ldots, t_n \in R_n$ such that $\pi_A(\sigma_C(\{t_1\} \bowtie \cdots \bowtie \{t_n\})) = \{t\}$.

Suppose $-t$ has an exact translation $\nabla D = \nabla R_1, \ldots, \nabla R_n$ where $\nabla R_i \subseteq R_i$, $i = 1..n$. We prove that $\nabla D' = \nabla R'_1, \ldots, \nabla R'_n$ where $\nabla R'_i = \nabla R_i \cap R_i^*$, $i = 1..n$, also is an exact translation. First, we prove that $\nabla D'$ is a translation, i.e., $t \notin V'$ where V' is the new view based on the updated database $D - \nabla D'$. Suppose for the sake of a contradiction that $t \in V'$. Then, we know there exist $t_i \in R_i - \nabla R'_i$ for $i = 1..n$ such that $\pi_A(\sigma_C(\{t_1\} \bowtie \cdots \bowtie \{t_n\})) = \{t\}$. Based on Definition 2.3.9 of lineage, $t_i \in R_i^*$, $i = 1..n$. Since $t_i \in R_i - \nabla R'_i$, we know that $t_i \notin \nabla R'_i = \nabla R_i \cap R_i^*$. Therefore since $t_i \in R_i^*$, $t_i \notin \nabla R_i$,

[1] Note that we abuse the subset symbol "$\subseteq$" to operate on lists of tables: $\nabla D \subseteq lineage(t, V, D)$ is shorthand for $\nabla R_1 \subseteq R_1^*, \ldots, \nabla R_n \subseteq R_n^*$.

and thus $t_i \in R_i - \nabla R_i$. Therefore $t \in \pi_A(\sigma_C((R_1 - \nabla R_1) \bowtie \cdots \bowtie (R_n - \nabla R_n)))$, which contradicts the fact that ∇D is a translation for $-t$. Thus, $t \notin V'$ and $\nabla D'$ is a translation for $-t$. Because ∇D is an exact translation and $\nabla D' \subseteq \nabla D$, by the definition of an exact translation $\nabla D'$ also is exact. □

4.4 View Tuple Deletion Algorithm

Based on our observations in Section 4.3, we specify an algorithm DELETE(t, V, D) in Figure 4.3 that translates deletion $-t$ on view V into deletions ∇D on base database D. The algorithm performs the deletions ∇D on the base database. We assume that if V is materialized (or if there are other materialized views over D), then a view maintenance algorithm will detect the deletions and modify any materialized views accordingly. Our algorithm can be modified easily to return ∇D and/or ∇V (the deletions on V induced by ∇D; note $\nabla V = \{t\}$ in the case of an exact translation) if desired. DELETE finds an exact translation for $-t$ whenever one exists. In cases when $-t$ has no exact translation, DELETE uses an inexact translation, with some attempt to minimize the side-effect. The algorithm proceeds as follows.

In line 1 of Figure 4.3, we compute t's lineage using the tracing query from Section 2.4. We then compute t's exclusive lineage D^{**} based on Definition 4.3.2 (lines 2–5), and delete it from D (line 6). Computing the exclusive lineage requires one n-way join for each tuple in t's lineage, but lineage is typically very small, and each join is expected to be cheap since one branch of the join is simply the lineage tuple $\{t_i\}$. If deleting t's exclusive lineage induces the deletion of t from V (line 7), then we are done, as in Example 4.3.4. We know that t has not been deleted if it is still derivable from its remaining lineage: $t \in \pi_A(\sigma_C((R_1^* - R_1^{**}) \bowtie \cdots \bowtie (R_n^* - R_n^{**})))$.

If deleting the exclusive lineage did not successfully delete t, then we try the next step discussed in Section 4.3, which is to consider deleting the remainder of some lineage branch R_i^*. Before exploring this option, we recompute t's lineage in our smaller D, i.e., in the D that remains after deleting D^{**} (line 8). Any tuple t_i that is eliminated from R_i^* in the recomputed lineage no longer contributes to t: the tuples it formerly joined with to

procedure DELETE($t, V = \pi_A(\sigma_C(R_1 \bowtie \cdots \bowtie R_n)), D$)
// Compute t's lineage:
(1) $D^* = \langle R_1^*, \ldots, R_n^* \rangle \leftarrow Split_{\mathbf{R}_1,\ldots,\mathbf{R}_n}(\sigma_{C \wedge A=t}(R_1 \bowtie \cdots \bowtie R_n))$;
// Compute and delete t's exclusive lineage:
(2) $D^{**} = \langle R_1^{**}, \ldots, R_n^{**} \rangle \leftarrow \langle \varnothing, \ldots, \varnothing \rangle$;
(3) **for** $i = 1..n$ **do**
(4) **for each** $t_i \in R_i^*$ **do**
(5) **if** $\pi_A(\sigma_C(R_1 \bowtie \cdots \bowtie \{t_i\} \bowtie \cdots \bowtie R_n)) = \{t\}$ **then** $R_i^{**} \leftarrow R_i^{**} \cup \{t_i\}$;
(6) $D \leftarrow D - D^{**}$;
// Check if t is deleted:
(7) **if** $t \notin \pi_A(\sigma_C((R_1^* - R_1^{**}) \bowtie \cdots \bowtie (R_n^* - R_n^{**})))$ **then return**;
// Recompute t's lineage in smaller D:
(8) $D^* = \langle R_1^*, \ldots, R_n^* \rangle \leftarrow Split_{\mathbf{R}_1,\ldots,\mathbf{R}_n}(\sigma_{C \wedge A=t}(R_1 \bowtie \cdots \bowtie R_n))$;
// Find a lineage branch with zero or smallest side-effect:
(9) $minE \leftarrow \infty$;
(10) **for** $i = 1..n$ **do**
(11) $E \leftarrow \pi_A(\sigma_C(R_1 \bowtie \cdots \bowtie R_i^* \bowtie \cdots \bowtie R_n)) - \{t\}$;
(12) $E \leftarrow E - \pi_A(\sigma_C(R_1 \bowtie \cdots \bowtie (R_i - R_i^*) \bowtie \cdots \bowtie R_n))$
(13) **if** $E = \varnothing$ **then** $R_i \leftarrow R_i - R_i^*$; **return**;
(14) **if** $|E| < minE$ **then** $m \leftarrow i$; $minE \leftarrow |E|$;
// Enumerate subsets of t's lineage with limited size:
(15) $k \leftarrow |\sigma_C(R_1^* \bowtie \cdots \bowtie R_n^*)|$;
(16) $S \leftarrow$ all subsets of D^* that contain at most k tuples;
(17) **for** each $D' = \langle R_1', \ldots, R_n' \rangle \in S$ **do**
(18) **if** $t \in \pi_A(\sigma_C((R_1^* - R_1') \bowtie \cdots \bowtie (R_n^* - R_n')))$
(19) **then** prune all subsets of D' from S;
(20) **else** $E \leftarrow \bigcup_{i=1..n} \pi_A(\sigma_C(R_1 \bowtie \cdots \bowtie R_i' \bowtie \cdots \bowtie R_n)) - \{t\}$;
(21) $E \leftarrow E - \pi_A(\sigma_C((R_1 - R_1') \bowtie \cdots \bowtie (R_n - R_n')))$;
(22) **if** $E = \varnothing$ **then** $D \leftarrow D - D'$; **return**;
(23) **else** prune all supersets of D' from S;
// Delete the lineage branch R_m^* with smallest side-effect:
(24) $R_m \leftarrow R_m - R_m^*$; **return**;

Figure 4.3: Translating the deletion of a view tuple t

produce t must have been deleted as part of the exclusive lineage.

In lines 9–14 we consider each remaining lineage branch R_i^* in turn. If ∇R_i^* is an exact

translation, then we are done. During the iteration, we keep track of the lineage branch with smallest side-effect, in case we eventually fail to find an exact translation at all. To determine the side-effect of ∇R_i^*, we first compute all potential extra deletions E using the query $\pi_A(\sigma_C(R_1 \bowtie \cdots \bowtie R_i^* \bowtie \cdots \bowtie R_n)) - \{t\}$ (line 11). We then remove from E tuples that are still derivable after deleting R_i^* (line 12). The remaining E is the actual side-effect induced by ∇R_i^*. In terms of efficiency, line 11 performs a join where one branch (R_i^*) is expected to be very small. Line 12 appears to be expensive—nearly a recomputation of the entire view—however a good optimizer can effectively introduce a semijoin with E into the right-hand operand of the difference to keep the join small. If we find a translation with no side-effect, i.e., $E = \varnothing$, then we are done (line 13). Otherwise, line 14 determines if we are seeing the smallest side-effect so far and, if so, records the side-effect size and current branch in $minE$ and m, respectively.

Example 4.4.1 Consider Example 2 from Table 4.1, which deletes tuple $t = \langle \texttt{lisa}, \texttt{f3} \rangle$ from V. In procedure DELETE, we compute t's lineage and exclusive lineage, and obtain:

$$lineage(t, V, D) = \langle \texttt{UserGroup}^* = \{\langle \texttt{lisa}, \texttt{eng1} \rangle, \langle \texttt{lisa}, \texttt{eng6} \rangle\}, \texttt{GroupAccess}^* = \{\langle \texttt{eng1}, \texttt{f3} \rangle, \langle \texttt{eng6}, \texttt{f3} \rangle\} \rangle$$

$$elineage(t, V, D) = \langle \texttt{UserGroup}^{**} = \{\langle \texttt{lisa}, \texttt{eng6} \rangle\},\ \texttt{GroupAccess}^{**} = \varnothing \rangle$$

We first delete t's exclusive lineage from the base database D (lines 1–6), which does not induce the deletion of t (line 7). We then recompute t's lineage in the updated D (line 8) and obtain:

$$lineage(t, V, D) = \langle \texttt{UserGroup}^* = \{\langle \texttt{lisa}, \texttt{eng1} \rangle\},\ \texttt{GroupAccess}^* = \{\langle \texttt{eng1}, \texttt{f3} \rangle\} \rangle$$

Deleting the lineage branch $\texttt{UserGroup}^*$ has the side-effect of deleting $\langle \texttt{lisa}, \texttt{f4} \rangle$ as well as $\langle \texttt{lisa}, \texttt{f3} \rangle$. However, deleting the lineage branch $\texttt{GroupAccess}^*$ has no side-effect, so we further delete $\texttt{GroupAccess}^*$. The entire exact translation is to delete tuple $\langle \texttt{lisa}, \texttt{eng6} \rangle$ from UserGroup and $\langle \texttt{eng1}, \texttt{f3} \rangle$ from GroupAccess. □

If t does not have a lineage branch whose deletion is side-effect free, we next look for a combination of tuples from different lineage branches whose deletion forms an exact translation for $-t$ (lines 15–23). We look for an exact translation by enumerating subsets of t's lineage, based on Theorem 4.3.7. Letting $x = |R_1^*| + |R_2^*| + \ \ldots\ + |R_n^*|$, we

must enumerate and check up to 2^x possible translations. Although x tends to be relatively small, in some cases we can further bound the size of the search space based on the number of different ways of deriving t as shown in Theorem 4.4.2. Recall that we consider set semantics in this chapter, and by Theorem 2.3.2 there is always a unique derivation for any given view tuple under set semantics. However, had the view been duplicate-preserving, t may have multiple derivations (Section 2.7), which effectively represent different ways of deriving t through the view. Here and later in this chapter, we use the term "derivation of t" to mean one of t's derivations had the view been duplicate-preserving.

Theorem 4.4.2 Consider an SPJ view $V = \pi_A(\sigma_C(R_1 \bowtie \cdots \bowtie R_n))$ over database D and a tuple $t \in V$. Let $lineage(t, V, D) = \langle R_1^*, \ldots, R_n^* \rangle$, and let $k = |\sigma_C(R_1^* \bowtie \cdots \bowtie R_n^*)|$. k is the number of different ways of deriving tuple t. If $-t$ has an exact translation, then it has an exact translation $\nabla D' = \nabla R_1', \ldots, \nabla R_n'$ such that $R_i' \subseteq R_i^*$ for $i = 1..n$, and $|R_1'| + \cdots + |R_n'| \leq k$. □

Proof: Informally, to delete t we need only delete one tuple from each of t's k derivations. If we have an exact translation for t, it must delete at least one tuple from each of t's derivations. We can eliminate from the exact translation all but one tuple from each derivation, and we still have an exact translation.[2]

The proof relies on the following fact, which is obvious enough that we omit detailed justification:

- For any $t_1 \in R_1^*, \ldots, t_n \in R_n^*$, $\pi_A(\sigma_C(\{t_1\} \bowtie \cdots \bowtie \{t_n\})) = \{t\}$ or $\pi_A(\sigma_C(\{t_1\} \bowtie \cdots \bowtie \{t_n\})) = \varnothing$. The number of distinct combinations such that $\pi_A(\sigma_C(\{t_1\} \bowtie \cdots \bowtie \{t_n\})) = \{t\}$ is k.

Let the following list enumerate the k distinct combinations of $t_1 \in R_1^*, \ldots, t_n \in R_n^*$ such that $\pi_A(\sigma_C(\{t_1\} \bowtie \cdots \bowtie \{t_n\})) = \{t\}$.

(1) $t_1^1,\ t_2^1, \ldots, t_n^1$

[2]This proof might lead us to consider an alternate approach: Instead of considering all lineage subsets up to size k, we compute the derivations of t and then consider all combinations of one tuple from each derivation. It turns out that this enumeration can actually be more expensive than the one we are using, because without expensive bookkeeping it might end up considering the same subset multiple times.

(2) $t_1^2,\ t_2^2, \ldots, t_n^2$

...

(k) $t_1^k,\ t_2^k, \ldots, t_n^k$

We know $-t$ has an exact translation $\nabla D'' = \nabla R_1'', \ldots, \nabla R_n''$. Without loss of generality (by Theorem 4.3.7), assume $R_i'' \subseteq R_i^*$, $i = 1..n$. We need to prove that there is an exact translation $\nabla D' = \nabla R_1', \ldots, \nabla R_n'$ such that $|R_1'| + \cdots + |R_n'| \leq k$. For each combination $j = 1..k$ in the list above, there exists at least one t_i^j, $i \in 1..n$, such that $t_i^j \in \nabla R_i''$, because otherwise t would remain in the view after deleting D''. Let $D' = \bigcup_{j=1..k} \{t_i^j\}$ which is of size k. $\nabla D'$ is a translation, because it deletes one tuple from each of the k combinations that produce t. Furthermore, it is an exact translation because it is a subset of $\nabla D''$, which is an exact translation. □

Lines 15–17 compute the bound k from Theorem 4.4.2 and initiate the subset enumeration. For each candidate translation $\nabla D'$, line 18 checks if $\nabla D'$ is indeed a translation, by checking if t is still derivable after deleting D'. If so, then $\nabla D'$ is not a translation, but it does give us information that we can use to further prune the search space: If deleting D' does not delete t, then nor can deleting any subset of D' (line 19). (This pruning technique seems to suggest we should enumerate larger subsets first, but in the other branch of the **if** statement we introduce a pruning technique that eliminates supersets.)

Lines 20–22 check if the candidate translation $\nabla D'$ is an exact translation, by checking if it has any side-effect. The procedure is similar but not identical to lines 11–13. We first compute all potential extra deletions E by joining each R_i' to the remaining relations (line 20). We then remove from E tuples that are still derivable after deleting D' (line 21). The remaining E is the actual side-effect induced by $\nabla D'$.[3] If E is empty, then we have found an exact translation, so we perform the deletion and are done (line 22). Otherwise, line 23 performs additional pruning: If deleting D' introduces a side-effect, then so will deleting any superset of D'.

Example 4.4.3 Consider Example 3 from Table 4.1, which deletes tuple $t = \langle \texttt{joe}, \texttt{f3} \rangle$ from V. In DELETE, we compute:

[3]Note that this computation is similar to incremental view maintenance [GMS93], which we used in Chapter 3.

$$lineage(t, V, D) = \langle \texttt{UserGroup}^* = \{\langle \texttt{joe}, \texttt{eng1}\rangle, \langle \texttt{joe}, \texttt{eng2}\rangle\},$$
$$\texttt{GroupAccess}^* = \{\langle \texttt{eng1}, \texttt{f3}\rangle, \langle \texttt{eng2}, \texttt{f3}\rangle\}\rangle$$
$$elineage(t, V, D) = \langle \texttt{UserGroup}^{**} = \varnothing,\ \texttt{GroupAccess}^{**} = \varnothing\rangle$$

We first delete t's exclusive lineage, which does not induce the deletion of t, and then we recompute t's lineage. (Obviously we can add an **if** statement in the algorithm to skip these deletion and recomputation steps when the exclusive lineage is empty, as in this example.) Deleting either remaining lineage branch of t will cause a side-effect, so we proceed to enumerate subsets of t's lineage. The number of t's derivations is $k = |\sigma_{\texttt{group}='\texttt{eng}\%'}(\texttt{UserGroup}^* \bowtie \texttt{GroupAccess}^*)| = 2$, so we only need to consider lineage subsets containing one or two tuples. Enumerating these subsets, we find an exact translation $\nabla D'$ for $-t$ where $D' = \langle \texttt{UserGroup}' = \{\langle \texttt{joe}, \texttt{eng2}\rangle\}, \texttt{GroupAccess}' = \{\langle \texttt{eng1}, \texttt{f3}\rangle\}\rangle$. □

Let us consider the complexity of lines 15–23. The number of iterations in the **for** loop is bounded by $|D^*|^k$, where k is the number of derivations of t. In general, both $|D^*|$ and k are small, and we use pruning techniques to further reduce the number of iterations. Within each iteration, the complexity of line 18 is similar to that of line 7, the complexity of line 20 is n times the complexity of line 11 (recall our view is an n-way join), and the complexity of line 21 is similar to line 12. Empirical results are reported in Section 4.7.

As the final step of the algorithm (line 24), if we cannot find a lineage subset D' whose deletion causes exactly the deletion of t, then by Theorem 4.3.7 $-t$ has no exact translation. In this case, we delete the lineage branch R^*_m with the smallest side-effect of all lineage branches, which we saved in line 14.

Example 4.4.4 Consider Example 4 from Table 4.1, which deletes tuple $t = \langle \texttt{joe}, \texttt{f4}\rangle$ from V. In DELETE, we compute:

$$lineage(t, V, D) = \langle \texttt{UserGroup}^* = \{\langle \texttt{joe}, \texttt{eng1}\rangle, \langle \texttt{joe}, \texttt{eng2}\rangle\},$$
$$\texttt{GroupAccess}^* = \{\langle \texttt{eng1}, \texttt{f4}\rangle, \langle \texttt{eng2}, \texttt{f4}\rangle\}\rangle$$
$$elineage(t, V, D) = \langle \texttt{UserGroup}^{**} = \varnothing,\ \texttt{GroupAccess}^{**} = \varnothing\rangle$$

Through lines 1–23 in the DELETE algorithm, we discover that $-t$ has no exact translation: The number of t's derivations is $k = 2$, and deleting any subset of t's lineage with no more

than two tuples either does not cause the deletion of t or causes deletions in addition to t. Thus, we delete $\texttt{UserGroup}^*$, which has the smaller side-effect of the two lineage branches. □

To our knowledge, all previous view update algorithms either do not guarantee exact translations for deletions even when they exist, e.g., [Mas84, LS91], or they guarantee exactness for a restricted class of views satisfying certain functional dependencies or foreign key constraints, e.g., [DB82, Kel86]. Algorithm DELETE is the first to guarantee exact deletion translations whenever they exist for general SPJ views.

4.5 Deleting Sets of View Tuples

So far we have considered translating the deletion of a single tuple t from view V. In general, users may want to request the deletion of a set of tuples $T \subseteq V$, specified either by value or by a selection condition. In this section we extend our algorithm to translate a view deletion request $-T$, while in the next section we consider condition-based deletions.

We might be tempted to translate view deletion $-T$ ($T \subseteq V$) by translating deletion $-t$ for each $t \in T$ using algorithm DELETE from Section 4.4. In addition to being inefficient, this approach is too conservative: it is possible that there is an exact translation for $-T$ even when there is no exact translation for some tuples $t \in T$. This case occurs when extra deletions from an inexact translation for $-t$, $t \in T$, are not actually a side-effect, because the extra deletions are contained in T. Thus, we must translate $-T$ as a unit.

First let us generalize Definition 4.2.1 of a view deletion translation:

Definition 4.5.1 (View Deletion Translation for $-T$) Consider a view V, a deletion request $-T$ where $T \subseteq V$, and a database deletion ∇D. Let V' be the new view based on the updated database $D - \nabla D$, and let $\nabla V = V - V'$ be the actual deleted view tuples. We say that ∇D is a *translation* for $-T$ if $T \subseteq \nabla V$. If $T = \nabla V$ then ∇D is an *exact* translation, otherwise ∇D causes *side-effect* $E = \nabla V - T$. □

Recall from Section 2.4 that we can compute T's lineage according to V using one query $\langle R_1^*, \ldots, R_n^* \rangle = Split_{\mathbf{R}_1,\ldots,\mathbf{R}_n}(\sigma_C(R_1 \bowtie \cdots \bowtie R_n) \ltimes T)$, and recall from Section

4.3 that a tuple $t_i \in R_i^*$ belongs to T's exclusive lineage R_i^{**} if and only if:

$$\pi_A(\sigma_C(R_1 \bowtie \cdots \bowtie R_{i-1} \bowtie \{t_i\} \bowtie R_{i+1} \bowtie \cdots \bowtie R_n)) \subseteq T$$

Thus, the observations we made on the relationship between data lineage and single-tuple deletions in Section 4.3 all carry over directly to the deletion of a view tuple set. Furthermore, Theorem 4.4.2 in Section 4.4 can be extended to cover $-T$ as follows.

Theorem 4.5.2 Consider an SPJ view $V = \pi_A(\sigma_C(R_1 \bowtie \cdots \bowtie R_n))$ over database D and a tuple set $T \subseteq V$. Let $lineage(T, V, D) = \langle R_1^*, \ldots, R_n^* \rangle$. If $-T$ has an exact translation, then it has an exact translation that deletes at most k tuples in T's lineage, where $k = |\sigma_C(R_1^* \bowtie \cdots \bowtie R_n^*) \ltimes T|$ is the total number of derivations of all tuples $t \in T$. □

Proof: The proof is very similar to the proof of Theorem 4.4.2, with the only notable difference being the computation of k. Here, we reduce the size of k after joining all lineage branches by performing a semijoin with T, because joining lineage tuples that result from two different view tuples t_1 and t_2 in T may result in view tuples t_3 outside of T. We do not want to accidentally count derivations of t_3 in our bound k.

The proof relies on the following fact, which is obvious enough that we omit detailed justification:

- Consider all combinations $t_1 \in R_1^*, \ldots, t_n \in R_n^*$. The number of distinct combinations such that $\pi_A(\sigma_C(\{t_1\} \bowtie \cdots \bowtie \{t_n\})) = \{t\}$ and $t \in T$ is k.

Let the following list enumerate the k distinct combinations of $t_1 \in R_1^*, \ldots, t_n \in R_n^*$ such that $\pi_A(\sigma_C(\{t_1\} \bowtie \cdots \bowtie \{t_n\})) = \{t\}$ and $t \in T$:

(1) $t_1^1,\ t_2^1, \ldots, t_n^1$
(2) $t_1^2,\ t_2^2, \ldots, t_n^2$
...
(k) $t_1^k,\ t_2^k, \ldots, t_n^k$

We know T has an exact translation $\nabla D'' = \nabla R_1'', \ldots, \nabla R_n''$. Without loss of generality (by the obvious set generalization of Theorem 4.3.7), assume $R_i'' \subseteq R_i^*$, $i = 1..n$. We need to prove that there is an exact translation $\nabla D' = \nabla R_1', \ldots, \nabla R_n'$ such that

$|R'_1| + \cdots + |R'_n| \leq k$. For each combination $j = 1..k$ in the list above, there exists at least one t_i^j, $i \in 1..n$, such that $t_i^j \in \nabla R''_i$, because otherwise some $t \in T$ would remain in the view after deleting D''. Let $D' = \bigcup_{j=1..k}\{t_i^j\}$ which is of size k. $\nabla D'$ is a translation, because it deletes one tuple from each of the k combinations that produce all of the t's in T. Furthermore, it is an exact translation because it is a subset of $\nabla D''$, which is an exact translation. □

Thus, we need only small modifications to our DELETE algorithm from Section 4.4 to translate the deletion of a view subset T. Figure 4.4 shows the modified algorithm DELETE-SET with all changes underlined. DELETE-SET finds a translation for $-T$ that is guaranteed to be exact whenever an exact translation exists.

Example 4.5.3 Consider Example 5 from Table 4.1, which deletes a tuple set $T = \{\langle \texttt{joe}, \texttt{f4}\rangle, \langle \texttt{lisa}, \texttt{f4}\rangle\}$ from V. In DELETE-SET, we compute T's lineage and exclusive lineage as:

$$lineage(T, V, D) = \langle \texttt{UserGroup}^* = \{\langle \texttt{joe}, \texttt{eng1}\rangle, \langle \texttt{lisa}, \texttt{eng1}\rangle\}, \texttt{GroupAccess}^* = \{\langle \texttt{eng1}, \texttt{f4}\rangle\}\rangle$$

$$elineage(T, V, D) = \langle \texttt{UserGroup}^{**} = \varnothing,\ \texttt{GroupAccess}^{**} = \{\langle \texttt{eng1}, \texttt{eng4}\rangle\}\rangle$$

Deleting T's exclusive lineage induces the deletion of T. Thus, we find an exact translation $\nabla elineage(T, V, D)$ for $-T$.

Now consider what would have happened had we translated the deletion of the two tuples $\langle \texttt{joe}, \texttt{f4}\rangle$ and $\langle \texttt{lisa}, \texttt{f4}\rangle$ separately, using our original algorithm DELETE. Suppose we translate deletion $t = \langle \texttt{joe}, \texttt{f4}\rangle$ first. In Example 4.4.4 we showed that deleting $\langle \texttt{joe}, \texttt{f4}\rangle$ has no exact translation, so the algorithm chooses to delete lineage branch $\texttt{UserGroup}^* = \{\langle \texttt{joe}, \texttt{eng1}\rangle, \langle \texttt{joe}, \texttt{eng2}\rangle\}$. The side-effect $E = \{\langle \texttt{joe}, \texttt{f3}\rangle\}$ of this translation is not contained in the original view request $T = \{\langle \texttt{joe}, \texttt{f4}\rangle, \langle \texttt{lisa}, \texttt{f4}\rangle\}$, so regardless of the translation chosen for tuple deletion $t = \langle \texttt{lisa}, \texttt{f4}\rangle$, the final translation for $-T$ will be inexact. □

procedure DELETE-SET($\underline{\underline{T}}, V = \pi_A(\sigma_C(R1 \bowtie \cdots \bowtie R_n)), D$)
// Compute $\underline{\underline{T}}$'s lineage:
(1) $D^* = \langle R_1^*, \ldots, R_n^* \rangle \leftarrow Split_{\mathbf{R}_1,\ldots,\mathbf{R}_n}(\sigma_C(R_1 \bowtie \cdots \bowtie R_n) \underline{\underline{\ltimes T}})$;

// Compute and delete $\underline{\underline{T}}$'s exclusive lineage:
(2) $D^{**} = \langle R_1^{**}, \ldots, R_n^{**} \rangle \leftarrow \langle \varnothing, \ldots, \varnothing \rangle$;
(3) **for** $i = 1..n$ **do**
(4) **for** each $t_i \in R_i^*$ **do**
(5) **if** $\pi_A(\sigma_C(R_1 \bowtie \cdots \bowtie \{t_i\} \bowtie \cdots \bowtie R_n)) \underline{\underline{\subseteq T}}$ **then** $R_i^{**} \leftarrow R_i^{**} \cup \{t_i\}$;
(6) $D \leftarrow D - D^{**}$;

// Check if $\underline{\underline{T}}$ is deleted:
(7) **if** $\underline{\underline{T \cap}}\, \pi_A(\sigma_C((R_1^* - R_1^{**}) \bowtie \cdots \bowtie (R_n^* - R_n^{**}))) \underline{\underline{= \varnothing}}$ **then return**;

// Recompute $\underline{\underline{T}}$'s lineage in smaller D:
(8) $D^* = \langle R_1^*, \ldots, R_n^* \rangle \leftarrow Split_{\mathbf{R}_1,\ldots,\mathbf{R}_n}(\sigma_C(R_1 \bowtie \cdots \bowtie R_n) \underline{\underline{\ltimes T}})$;

// Find a lineage branch with zero or smallest side-effect:
(9) $minE \leftarrow \infty$;
(10) **for** $i = 1..n$ **do**
(11) $E \leftarrow \pi_A(\sigma_C(R_1 \bowtie \cdots \bowtie R_i^* \bowtie \cdots \bowtie R_n)) \underline{\underline{- T}}$;
(12) $E \leftarrow E - \pi_A(\sigma_C(R_1 \bowtie \cdots \bowtie (R_i - R_i^*) \bowtie \cdots \bowtie R_n))$
(13) **if** $E = \varnothing$ **then** $R_i \leftarrow R_i - R_i^*$; **return**;
(14) **if** $|E| < minE$ **then** $m \leftarrow i$; $minE \leftarrow |E|$;

// Enumerate subsets of $\underline{\underline{T}}$'s lineage with limited size:
(15) $k \leftarrow |\sigma_C(R_1^* \bowtie \cdots \bowtie R_n^*) \underline{\underline{\ltimes T}}|$;
(16) $S \leftarrow$ all subsets of D^* that contain at most k tuples;
(17) **for** each $D' = \langle R_1', \ldots, R_n' \rangle \in S$ **do**
(18) **if** $\underline{\underline{T \cap}}\, \pi_A(\sigma_C((R_1^* - R_1') \bowtie \cdots \bowtie (R_n^* - R_n'))) \underline{\underline{\neq \varnothing}}$
(19) **then** prune all subsets of D' from S;
(20) **else** $E \leftarrow \bigcup_{i=1..n} \pi_A(\sigma_C(R_1 \bowtie \cdots \bowtie R_i' \bowtie \cdots \bowtie R_n)) \underline{\underline{- T}}$;
(21) $E \leftarrow E - \pi_A(\sigma_C((R_1 - R_1') \bowtie \cdots \bowtie (R_n - R_n')))$;
(22) **if** $E = \varnothing$ **then** $D \leftarrow D - D'$; **return**;
(23) **else** prune all supersets of D' from S;

// Delete the lineage branch R_m^* with smallest side-effect:
(24) $R_m \leftarrow R_m - R_m^*$; **return**;

Figure 4.4: Translating the deletion of a view subset T

4.6 Deleting View Tuples Specified by Selection Conditions

Consider as usual our SPJ view $V = \pi_A(\sigma_C(R_1 \bowtie \cdots \bowtie R_n))$. Suppose that instead of specifying an actual view tuple or set of tuples to be deleted, the user performs a condition-based deletion as in SQL: "`DELETE FROM` V `WHERE` C'" where C' can be any selection condition, similar to the conditions C used in view definitions. A naive approach is to first compute the tuple set $T = \sigma_{C'}(V)$, then translate $-T$ using algorithm DELETE-SET from Section 4.5. However, the computation of $T = \sigma_{C'}(V)$ can be expensive, particularly if V is a virtual view. Fortunately, instead of computing T, we can incorporate the selection condition C' into our translation algorithm. The new algorithm DELETE-COND is specified in Figure 4.5, with the changes from DELETE-SET underlined.

In lines 1 and 8, to compute the lineage of $T = \sigma_{C'}(V)$ according to V, we use query $Split_{\mathbf{R}_1,\ldots,\mathbf{R}_n}(\sigma_{C \wedge C'}(R_1 \bowtie \cdots \bowtie R_n))$, instead of performing a semijoin with T as in DELETE-SET. In line 5, we test whether a tuple $t_i \in R_i^*$ belongs to the exclusive lineage of $T = \sigma_{C'}(V)$ by testing whether $\pi_A(\sigma_{C \wedge \neg C'}(R_1 \bowtie \cdots \bowtie \{t_i\} \bowtie \cdots \bowtie R_n)) = \varnothing$, because this condition is satisfied if and only if t_i does not contribute to any view tuple that fails condition C'. We can similarly compute the potential side-effect of a base deletion ∇R_i^* using $\pi_A(\sigma_{C \wedge \neg C'}(R_1 \bowtie \cdots \bowtie R_i^* \bowtie \cdots \bowtie R_n))$ in lines 11 and 20. Finally, in lines 7 and 18, we test whether deleting base tuples $D' = \langle R_1', \ldots, R_n' \rangle$ is a translation by testing whether $\pi_A(\sigma_{C \wedge C'}((R_1^* - R_1') \bowtie \cdots \bowtie (R_n^* - R_n'))) = \varnothing$.

Example 4.6.1 Consider Example 6 from Table 4.1, which deletes all tuples with $\texttt{user} = \texttt{'ted'}$ from view V. Let $\mathbf{U}$ and $\mathbf{G}$ denote the schemas of base tables `UserGroup` and `GroupAccess`, respectively. In DELETE-COND, we compute the lineage of $T = \sigma_{\texttt{user='ted'}}(V)$ as follows:

$$\langle \texttt{UserGroup}^*, \texttt{GroupAccess}^* \rangle = Split_{\mathbf{U},\mathbf{G}}(\sigma_{\texttt{group='eng\%'} \wedge \texttt{user='ted'}}(\texttt{UserGroup} \bowtie \texttt{GroupAccess}))$$

The result is shown in Figure 4.6. We then compute T's exclusive lineage and obtain $D^{**} = \langle \texttt{UserGroup}^{**}, \texttt{GroupAccess}^{**} \rangle$ in Figure 4.6. ∇D^{**} is a translation, and therefore is an exact translation for the condition-based view deletion. □

procedure DELETE-COND($\underline{\underline{C'}}, V = \pi_A(\sigma_C(R1 \bowtie \cdots \bowtie R_n)), D$)
// Compute $\underline{\underline{T = \sigma_{C'}(V)}}$'s lineage:
(1) $D^* = \langle R_1^* \ldots R_n^* \rangle \leftarrow Split_{\mathbf{R}_1,\ldots,\mathbf{R}_n}(\sigma_{C \underline{\underline{\wedge C'}}}(R_1 \bowtie \cdots \bowtie R_n))$;

// Compute and delete T's exclusive lineage:
(2) $D^{**} = \langle R_1^{**}, \ldots, R_n^{**} \rangle \leftarrow \langle \varnothing, \ldots, \varnothing \rangle$;
(3) **for** $i = 1..n$ **do**
(4) **for** each $t_i \in R_i^*$ **do**
(5) **if** $\pi_A(\sigma_{C \underline{\underline{\wedge \neg C'}}}(R_1 \bowtie \cdots \bowtie \{t_i\} \bowtie \cdots \bowtie R_n)) = \varnothing$ **then** $R_i^{**} \leftarrow R_i^{**} \cup \{t_i\}$;
(6) $D \leftarrow D - D^{**}$;

// Check if T is deleted:
(7) **if** $\pi_A(\sigma_{C \underline{\underline{\wedge C'}}}((R_1^* - R_1^{**}) \bowtie \cdots \bowtie (R_n^* - R_n^{**}))) = \varnothing$ **then return**;

// Recompute T's lineage in smaller D:
(8) $D^* = \langle R_1^* \ldots R_n^* \rangle \leftarrow Split_{\mathbf{R}_1,\ldots,\mathbf{R}_n}(\sigma_{C \underline{\underline{\wedge C'}}}(R_1 \bowtie \cdots \bowtie R_n))$;

// Find a lineage branch with zero or smallest side-effect:
(9) $minE \leftarrow \infty$;
(10) **for** $i = 1..n$ **do**
(11) $E \leftarrow \pi_A(\sigma_{C \underline{\underline{\wedge \neg C'}}}(R_1 \bowtie \cdots \bowtie R_i^* \bowtie \cdots \bowtie R_n))$;
(12) $E \leftarrow E - \pi_A(\sigma_C(R_1 \bowtie \cdots \bowtie (R_i - R_i^*) \bowtie \cdots \bowtie R_n))$
(13) **if** $E = \varnothing$ **then** $R_i \leftarrow R_i - R_i^*$; **return**;
(14) **if** $|E| < minE$ **then** $m \leftarrow i$; $minE \leftarrow |E|$;

// Enumerate subsets of T's lineage with limited size:
(15) $k \leftarrow |\sigma_{C \underline{\underline{\wedge C'}}}(R_1^* \bowtie \cdots \bowtie R_n^*)|$;
(16) $S \leftarrow$ all subsets of D^* that contain at most k tuples;
(17) **for** each $D' = \langle R_1', \ldots, R_n' \rangle \in S$ **do**
(18) **if** $\pi_A(\sigma_{C \underline{\underline{\wedge C'}}}((R_1^* - R_1') \bowtie \cdots \bowtie (R_n^* - R_n'))) \neq \varnothing$
(19) **then** prune all subsets of D' from S;
(20) **else** $E \leftarrow \bigcup_{i=1..n} \pi_A(\sigma_{C \underline{\underline{\wedge \neg C'}}}(R_1 \bowtie \cdots \bowtie R_i' \bowtie \cdots \bowtie R_n))$;
(21) $E \leftarrow E - \pi_A(\sigma_C((R_1 - R_1') \bowtie \cdots \bowtie (R_n - R_n')))$;
(22) **if** $E = \varnothing$ **then** $D \leftarrow D - D'$; **return**;
(23) **else** prune all supersets of D' from S;

// Delete the lineage branch R_m^* with smallest side-effect:
(24) $R_m \leftarrow R_m - R_m^*$; **return**;

Figure 4.5: Translating view deletions based on a selection condition C'

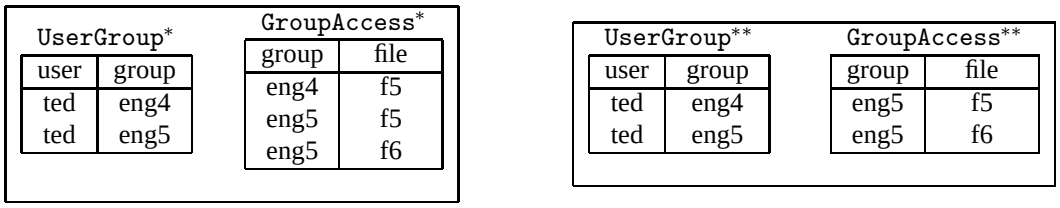


Figure 4.6: $T = \sigma_{\texttt{user}='\texttt{ted}'}(V)$, its lineage and exclusive lineage

4.7 Empirical Results

We have implemented our view deletion translation algorithm using stored procedures in a standard commercial relational DBMS. In this section, we present some preliminary performance results that explore the following two questions:

1. Roughly how often is there an exact translation for a view deletion $-t$ that deletes the exclusive lineage of t? That deletes a lineage branch of t? That deletes a subset of t's lineage? How often is there no exact translation at all?

2. How does our algorithm perform as we scale up the size of the base data?

We use synthetic data for the experiments based on the user access control database in our example from Section 4.1. Recall that the two base tables are $\texttt{UserGroup}(\texttt{user}, \texttt{group})$ and $\texttt{GroupAccess}(\texttt{group}, \texttt{file})$. The view we consider is $V = \pi_{\texttt{user},\texttt{file}}(\texttt{UserGroup} \bowtie \texttt{GroupAccess})$. (We omit the selection condition from our original example view.) Contents of the base tables are generated using the parameters listed in Table 4.2. There are *#groups* groups. The users in each group and the files that each group has access to are picked randomly from a fixed set of users and files with sizes *#users* and *#files*, respectively. The base data size depends on *#groups*, as well as parameters *users-per-group* (*UG* for short) and *files-per-group* (*FG* for short). The actual number of users for a given group is selected from a Gaussian distribution centered at *UG* with a standard deviation of $UG/2$, and similarly for the number of files each group has access to. For our experiments we always set $UG = FG$, and we refer to this parameter as *density*. When this number is low, the join is very selective, while when it is high, the join is heavily intertwined. Our experiments focus on the single-tuple deletion case, i.e., each request is to delete a single view tuple.

Parameter	*Description*
#groups	total number of groups
#users	maximum number of users
#files	maximum number of files
users-per-group (UG)	average number of users that belong to each group
files-per-group (FG)	average number of files that each group has access to

Table 4.2: Data generation parameters

4.7.1 Four Cases of Translations

Given a view deletion request $-t$, there are four possible outcomes for the translation according to algorithm DELETE:

1. $-t$ has an exact translation ∇D^{**} that deletes t's exclusive lineage D^{**}. In this case, our algorithm finds the exact translation ∇D^{**} and returns at line 7 of Figure 4.3.

2. Case (1) fails, but $-t$ has an exact translation ∇R_i^* that deletes a lineage branch R_i^* of $-t$. In this case, our algorithm finds an exact translation ∇R_i^* and returns at line 13. The number of iterations of lines 10–14 will vary from deletion to deletion.

3. Cases (1) and (2) both fail, but $-t$ has an exact translation. In this case, our algorithm finds an exact translation ∇D by deleting a subset of t's lineage. It returns at line 22, and the number of iterations of lines 17–23 will vary from deletion to deletion.

4. $-t$ has no exact translations, so the algorithm returns at line 24.

In our experiment, we study how often each of the above four cases occurs under different base data scenarios. Specifically, when generating the base data, we fix *#groups*, *#users*, and *#files* at 50, and we increase the density (*UG* and *FG*) from 1 to 30. For each base data scenario, we issue a deletion request $-t$ for each tuple $t \in V$, and count the number of translations that belong to each of the four cases. Figure 4.7 presents the percentage of each case among all four cases.

The result shows that for very sparse base data, case (1) happens frequently: most deletions $-t$ have an exact translation that deletes t's exclusive lineage. At the extreme, when the density is 1, each group g has exactly one user u and one file f, so all lineage

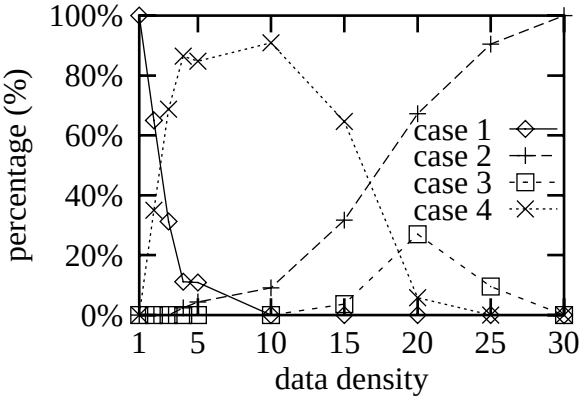


Figure 4.7: Four cases of translations

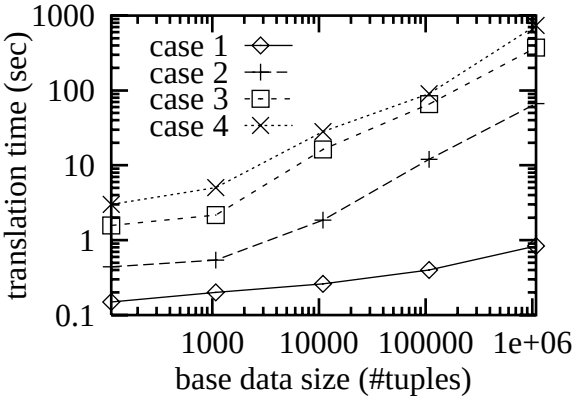


Figure 4.8: Translation time

tuples of any view tuple $t = \langle u, f \rangle$ are exclusive. Therefore deleting the entire exclusive lineage is always an exact translation. As the density increases, case (4) occurs more often. The intuition is that as more tuples are joined with each other, deleting any lineage tuple of t is likely to affect other view tuples, so $-t$ often does not have an exact translation. However, as the density further increases, cases (2) and (3) start to occur more frequently. Now, because each view tuple often has several derivations, although deleting a branch or subset of t's lineage may affect some derivations of another view tuple t', t' often can be derived in another way, so $-t$ still has an exact translation. Finally, when the base data is very dense, case (2) completely takes over.

4.7.2 Algorithm Scalability

In this experiment, we study the scalability of our algorithm to the size of the base data. Specifically, we fix the density at 8 and increase *#groups* from 10 to 100,000. (We also set *#users* = *#files* = *#groups*, but these values do not affect the base data size.) We continue to consider the four translation cases from Section 4.7.1 separately, and we delete view tuples one at a time until we have several representative instances of each case. Figure 4.8 shows the average time for the translation in each case (in seconds) as the total size of the base data increases. From the results plotted on log-scale x and y axes, we see that the translation time for each case grows at worst approximately linearly in the size of the base data.

4.8 Related Work

One of the biggest differences between our approach to the view update problem and most previous approaches is that we determine view update translations at view-update time instead of at view-definition time. Consequently, other approaches may miss exact view update translations when they exist, because those approaches cannot take all data-dependent information into account. Of course, there is a cost for this extra accuracy, namely examining the database state at run-time to compute the exact translation. However, in certain cases it is a cost that applications may want to pay.

Some previous approaches ask the view definer to specify, as part of a view definition, the set of permitted view update operations together with their translations, e.g., [Cle78, RS79]. A somewhat more automated approach is for the system to enumerate different view update translations at view definition time and let the view definer choose the preferable ones, e.g., [Kel86]. Another approach, proposed in [LS91], translates view updates using user-provided information at view-definition time as well as at view-update time. All of these approaches require input from the view definer or view updater, unlike our approach which can be fully automatic, allowing exact view update translations without any additional effort at view definition or update time.

A fully automatic approach is suggested in [Mas84], which provides ten general rules for translating view deletions and insertions on select, project, and join views. The concept of *view complements* is introduced in [BS81] to supply extra semantic information for selecting a view update translation. Neither of these approaches guarantees translation exactness.

An algorithm proposed in [DB82] guarantees exact translations in restricted cases by requiring certain functional dependencies on the base data. A concept of *source-tuple* is defined in [DB82], similar in spirit but with a different definition than data lineage. [Kel85] also proposed an algorithm that guarantees exact translations for a very restricted classes of SPJ views: to be updatable a view must include all key and join attributes of the base tables, and the base tables must join on the key attributes and satisfy foreign key constraints. Our approach makes no restrictions on the SPJ views we consider.

4.9 Chapter Summary

In this chapter we considered the view update problem in a conventional database (not data warehousing) environment. Exploiting the data lineage techniques developed in Chapter 2, we developed a fully automatic algorithm for translating deletions against any relational SPJ view into deletions against the underlying database. Deletions can be specified as a single tuple, a set of tuples, or as a selection condition over the view. Our algorithm can be parameterized by the view definition at compile-time, and by the view update request (and base data) at run-time. By taking data-dependent information into account at run-time and using techniques based on data lineage, our algorithm finds translations that are guaranteed to be exact whenever such translations exist. No previous algorithms we know of can make the same claim. Empirical results show that our algorithm often finds exact translations very quickly, and that it scales at worst linearly in the database size.

Chapter 5

Data Lineage for General Data Transformations

In Chapters 2–3 we studied the data lineage problem for warehouse data defined as relational views over relational sources, i.e., views specified using SQL or relational algebra. In many deployed production data warehouses, however, data imported from the sources may be "cleansed", integrated, and summarized through a sequence or graph of *transformations*. Many commercial warehousing systems provide tools for creating and managing such transformations as part of the *extract-transform-load* (*ETL*) process, e.g., [Inf, Mic, PPD, Sag]. The transformations may vary from simple algebraic operations or aggregations to complex procedural code.

In this chapter we consider the problem of lineage tracing in data warehouses created by general transformations. In this environment, the lineage tracing problem becomes considerably more difficult and open-ended, because we no longer have the luxury of a fixed set of operators or the algebraic properties offered by relational views. Furthermore, since transformation graphs in real ETL processes can often be quite complex—containing as many as 60 or more transformations—the storage requirements and runtime overhead associated with lineage tracing are very important considerations.

To define data lineage and provide lineage tracing algorithms for general transformations, we take advantage of known structure or properties of transformations when present.

Our lineage tracing algorithms apply to single transformations, to linear sequences of transformations, and to arbitrary acyclic transformation graphs. We also propose optimization techniques for improving tracing performance in the case of large transformation graphs. Our results can guide the design of transformational data warehouses that enable effective lineage tracing.

The chapter proceeds as follows. Section 5.1 introduces our running example for lineage tracing in the presence of general data transformations, and Section 5.2 formalizes the problem. In Section 5.3 we identify a set of relevant transformation properties. We then provide formal definitions of lineage and tracing algorithms for general warehouse transformations based on these properties. Section 5.4 presents an algorithm for combining transformations in a transformation sequence to improve tracing performance. Section 5.5 extends the algorithms to handle transformations with multiple inputs and outputs, and Section 5.6 presents our overall approach for tracing lineage through an arbitrary transformation graph. We have studied the performance of our algorithms empirically and we present the results in Section 5.7. Section 5.8 surveys related work and Section 5.9 concludes the chapter.

5.1 Running Example

We now present a running example to be used throughout the chapter. Consider a data warehouse with retail store data derived from two source tables:

`Product`(`prod-id`, `prod-name`, `category`, `price`, `valid`)
`Order`(`order-id`, `cust-id`, `date`, `prod-list`)

The `Product` table is mostly self-explanatory. Attribute `valid` specifies the time period during which a price is effective,[1] and is part of the key because a given product may have different prices over time, and thus multiple tuples in `Product`. The `Order` table also is mostly self-explanatory. Attribute `prod-list` specifies the list of ordered products with product ID and (parenthesized) quantity for each. Sample contents of small source tables

[1] We assume that `valid` is a simple string, which unfortunately is a typical ad-hoc treatment of time.

prod-id	prod-name	category	price	valid
111	Apple IMAC	computer	1200	10/1/1998–
222	Sony VAIO	computer	3280	9/1/1998–11/30/1998
222	Sony VAIO	computer	2250	12/1/1998–9/30/1999
222	Sony VAIO	computer	1950	10/1/1999–
333	Canon A5	electronics	400	4/2/1999–
444	Sony VAIO	computer	2750	12/1/1998–

Figure 5.1: Source data set for `Product`

order-id	cust-id	date	prod-list
0101	AAA	2/1/1999	333(10), 222(10)
0102	BBB	2/8/1999	111(10)
0379	CCC	4/9/1999	222(5), 333(5)
0524	DDD	6/9/1999	111(20), 333(20)
0761	EEE	8/21/1999	111(10)
0952	CCC	11/8/1999	111(5)
1028	DDD	11/24/1999	222(10)
1250	BBB	12/15/1999	222(10), 333(10)

Figure 5.2: Source data set for `Order`

are shown in Figures 5.1 and 5.2.

Suppose an analyst wants to build a warehouse table listing computer products that had a significant sales jump in the last quarter: the last quarter sales were more than twice the average sales for the preceding three quarters. A table `SalesJump` is defined in the data warehouse for this purpose. Figure 5.3 shows how the contents of table `SalesJump` can be specified using a *transformation graph* $\mathcal{G}$ with inputs `Order` and `Product`. $\mathcal{G}$ is a directed acyclic graph composed of the following seven transformations:

- $\mathcal{T}_1$ splits each input order according to its product list into multiple orders, each with a single ordered product and quantity. The output has schema $\langle$`order-id`, `cust-id`, `date`, `prod-id`, `quantity`$\rangle$.

- $\mathcal{T}_2$ filters out products not in the computer category.

- $\mathcal{T}_3$ effectively performs a relational join on the outputs from $\mathcal{T}_1$ and $\mathcal{T}_2$, with $\mathcal{T}_1$.`prod-id` $= \mathcal{T}_2$.`prod-id` and $\mathcal{T}_1$.`date` occurring in the period of $\mathcal{T}_2$.`valid`. $\mathcal{T}_3$ also drops attributes `cust-id` and `category`, so the output has schema $\langle$`order-id`, `date`, `prod-id`, `quantity`, `prod-name`, `price`, `valid`$\rangle$.

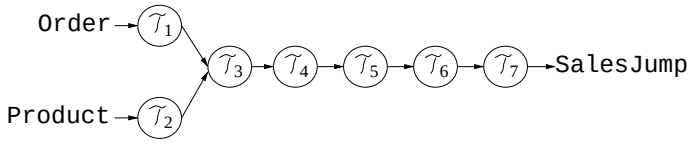


Figure 5.3: Transformations to derive SalesJump

Name	Description
$\mathcal{T}_1$	split orders
$\mathcal{T}_2$	select on product category
$\mathcal{T}_3$	join products and orders
$\mathcal{T}_4$	aggregate and pivot quarterly sales
$\mathcal{T}_5$	add a column avg3
$\mathcal{T}_6$	select on avg3
$\mathcal{T}_7$	remove columns

Figure 5.4: Transformation summary

- $\mathcal{T}_4$ computes the quarterly sales for each product. It groups the output from $\mathcal{T}_3$ by prod-name, computes the total sales for each product for the four previous quarters, and pivots the results to output a table with schema $\langle$prod-name, q1, q2, q3, q4$\rangle$, where q1–q4 are the quarterly sales.

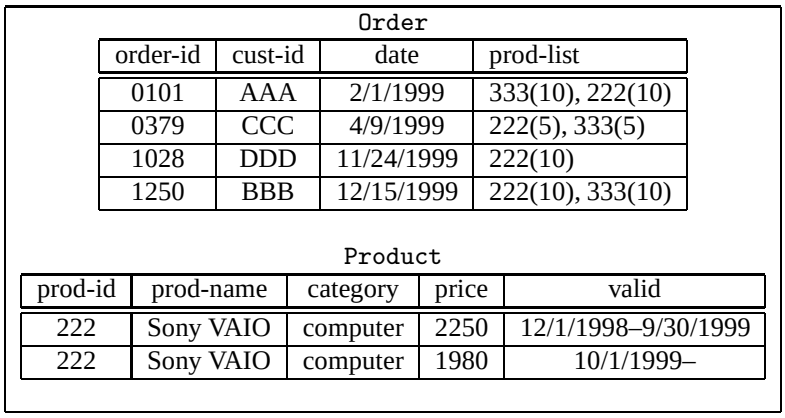
- $\mathcal{T}_5$ computes from the output of $\mathcal{T}_4$ the average sales of each product in the first three quarters. The output schema is $\langle$prod-name, q1, q2, q3, avg3, q4$\rangle$, where avg3 is the average sales $(\texttt{q1} + \texttt{q2} + \texttt{q3})/3$.

- $\mathcal{T}_6$ selects those products whose last quarter's sales were greater than twice the average of the preceding three quarters.

- $\mathcal{T}_7$ performs a final projection to output SalesJump with schema $\langle$prod-name, avg3, q4$\rangle$.

Figure 5.4 summarizes the transformations in $\mathcal{G}$. Note that some of these transformations ($\mathcal{T}_2$, $\mathcal{T}_5$, $\mathcal{T}_6$, and $\mathcal{T}_7$) could be expressed as standard relational operations, while others ($\mathcal{T}_1$, $\mathcal{T}_3$, and $\mathcal{T}_4$) could not.

As a simple lineage example, for the data in Figures 5.1 and 5.2 the warehouse table SalesJump contains tuple $t = \langle$Sony VAIO, 11250, 39600$\rangle$, indicating that the sales of VAIO computers jumped from an average of 11250 in the first three quarters to 39600 in the last quarter. An analyst may want to see the relevant detailed information by tracing the



Order

order-id	cust-id	date	prod-list
0101	AAA	2/1/1999	333(10), 222(10)
0379	CCC	4/9/1999	222(5), 333(5)
1028	DDD	11/24/1999	222(10)
1250	BBB	12/15/1999	222(10), 333(10)

Product

prod-id	prod-name	category	price	valid
222	Sony VAIO	computer	2250	12/1/1998–9/30/1999
222	Sony VAIO	computer	1980	10/1/1999–

Figure 5.5: Lineage of $\langle$Sony VAIO, 11250, 39600$\rangle$

lineage of tuple t, that is, by inspecting the original input data items that produced t. Using the techniques to be developed in this chapter, from the source data in Figures 5.1 and 5.2 the analyst will be presented with the lineage result in Figure 5.5.

5.2 Transformations and Their Data Lineage

In this section, we formalize general data transformations and data lineage for transformations. We then briefly motivate why transformation properties can help us with lineage tracing.

5.2.1 Transformations

In this chapter we generally consider a *data set* to be any set of data items—tuples, values, complex objects—with no duplicates in the set. (The effect duplicates have on lineage tracing has been addressed in some detail earlier in Section 2.7, and the issues do not change fundamentally in the presence of transformations.) A *transformation* $\mathcal{T}$ is any procedure that takes data sets as input and produces data sets as output. For now, we will consider only transformations that take a single data set as input and produce a single output set. We will extend our results to transformations with multiple input sets and output sets in Section 5.5. For any input data set I, we say that the application of $\mathcal{T}$ to I resulting in an output set O, denoted $\mathcal{T}(I) = O$, is an *instance* of $\mathcal{T}$.

Given transformations $\mathcal{T}_1$ and $\mathcal{T}_2$, their *composition* $\mathcal{T} = \mathcal{T}_1 \circ \mathcal{T}_2$ is the transformation that first applies $\mathcal{T}_1$ to I to obtain I', then applies $\mathcal{T}_2$ to I' to obtain O. $\mathcal{T}_1$ and $\mathcal{T}_2$ are called $\mathcal{T}$'s *component transformations*. We assume the composition operation is associative: $(\mathcal{T}_1 \circ \mathcal{T}_2) \circ \mathcal{T}_3 = \mathcal{T}_1 \circ (\mathcal{T}_2 \circ \mathcal{T}_3)$. Thus, given transformations $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_n$, we represent the composition $((\mathcal{T}_1 \circ \mathcal{T}_2) \circ \ldots) \circ \mathcal{T}_n$ as a *transformation sequence* $\mathcal{T}_1 \circ \cdots \circ \mathcal{T}_n$. A transformation that is not defined as a composition of other transformations is *atomic*.

For now we will assume that all of our transformations are *stable* and *deterministic*. A transformation $\mathcal{T}$ is stable if it never produces spurious output items, i.e., $\mathcal{T}(\varnothing) = \varnothing$. A transformation is deterministic if it always produces the same output set given the same input set. All of the example transformations we have seen so far are stable and deterministic. An example of an unstable transformation is one that appends a fixed data item or set of items to every output set, regardless of the input. An example of a nondeterministic transformation is one that transforms a random sample of the input set. In practice we usually require transformations to be stable but often do not require them to be deterministic. We will defer our discussion of when the deterministic assumption can be dropped to Section 5.3.5, after we have formalized data lineage for transformations and presented our lineage tracing algorithms.

5.2.2 Data Lineage

In the general case a transformation may inspect the entire input data set to produce each item in the output data set, but in most cases there is a much more fine-grained relationship between the input and output data items: a data item o in the output set may have been derived from a small subset of the input data items (maybe only one), as opposed to the entire input data set. Given a transformation instance $\mathcal{T}(I) = O$ and an output item $o \in O$, the lineage of o is the actual set $I^* \subseteq I$ of input data items that contributed to o's derivation. We denote o's lineage as $I^* = \mathcal{T}^*(o, I)$. The lineage of a set of output data items $O^* \subseteq O$ is the union of the lineage of each item in the set: $\mathcal{T}^*(O^*, I) = \bigcup_{o \in O^*} \mathcal{T}^*(o, I)$. A detailed definition of data lineage for different types of transformations will be given in Section 5.3.

Knowing something about the workings of a transformation is important for tracing data lineage—if we know nothing, any input data item may have participated in the derivation

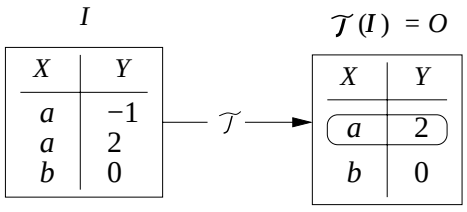


Figure 5.6: A transformation instance

of an output item. Let us consider an example. Given a transformation $\mathcal{T}$ and its instance $\mathcal{T}(I) = O$ in Figure 5.6, the lineage of the output item $\langle a, 2\rangle$ depends on $\mathcal{T}$'s definition, as we will illustrate. Suppose $\mathcal{T}$ is a transformation that filters out input items with a negative Y value (i.e., $\mathcal{T} = \sigma_{Y \geq 0}$ in relational algebra). Then the lineage of output item $o = \langle a, 2\rangle$ should include only input item $\langle a, 2\rangle$. Now, suppose instead that $\mathcal{T}$ groups the input data items based on their X values and computes the sum of their Y values multiplied by 2 (i.e., $\mathcal{T} = \alpha_{X,2*sum(Y)\ as\ Y}$ in relational algebra, where α performs grouping and aggregation). Then the lineage of output item $o = \langle a, 2\rangle$ should include input items $\langle a, -1\rangle$ and $\langle a, 2\rangle$, because o is computed from both of them. We will refer back to these two transformations later (along with our earlier examples from Section 5.1), so let us call the first one $\mathcal{T}_8$ and the second one $\mathcal{T}_9$.

Given a transformation specified as a standard relational operator or view, we can define and retrieve the exact data lineage for any output data item using the techniques introduced in Chapter 2. On the other hand, if we know nothing at all about a transformation, then the lineage of an output item must be defined as the entire input set. In reality transformations often lie between these two extremes—they are not standard relational operators, but they have some known structure or properties that can help us identify and trace data lineage.

The transformation properties we will consider often can be specified easily by the transformation author, or they can be inferred from the transformation definition (as relational operators, for example), or possibly even "learned" from the transformation's behavior. We do not focus on how properties are specified or discovered, but rather on how they are exploited for lineage tracing.

5.3 Lineage Tracing Using Transformation Properties

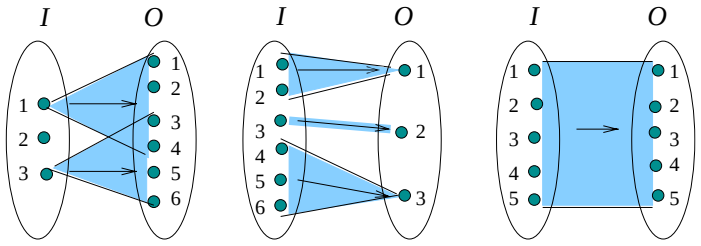
We consider three overall kinds of properties and provide algorithms that trace data lineage using these properties. First, each transformation is in a certain *transformation class* based on how it maps input data items to output items (Section 5.3.1). Second, we may have one or more *schema mappings* for a transformation, specifying how certain output attributes relate to input attributes (Section 5.3.2). Third, a transformation may be accompanied by a *tracing procedure* or *inverse transformation*, which is the best case for lineage tracing (Section 5.3.3). When a transformation exhibits many properties, we determine the best one to exploit for lineage tracing based on a property hierarchy (Section 5.3.4). We also discuss nondeterministic transformations (Section 5.3.5), and how indexes can be used to further improve tracing performance (Section 5.3.6).

5.3.1 Transformation Classes

In this section, we define three transformation classes: *dispatchers*, *aggregators*, and *black-boxes*. For each class, we give a formal definition of data lineage and specify a lineage tracing procedure. We also consider several subclasses for which we specify more efficient tracing procedures. Our informal studies have shown that about 95% of the transformations used in real data warehouses are dispatchers, aggregators, or their compositions (covered in Sections 5.4–5.6), and a large majority fall into the more efficient subclasses.

Dispatchers

A transformation $\mathcal{T}$ is a *dispatcher* if, for each individual data item in an input set, $\mathcal{T}$ produces zero or more of the output data items: $\forall I,\ \mathcal{T}(I) = \bigcup_{i \in I} \mathcal{T}(\{i\})$. Figure 5.7(a) illustrates a dispatcher, in which input item 1 produces output items 1–4, input item 3 produces output items 3–6, and input item 2 produces no output items. The lineage of an output item o according to a dispatcher $\mathcal{T}$ is defined as $\mathcal{T}^*(o, I) = \{i \in I \mid o \in \mathcal{T}(\{i\})\}$. As an example, a relational selection or projection operator is a typical dispatcher, while a transformation that computes the sum of a list of input numbers is not a dispatcher, because it computes an output data item based on two or more input data items.



(a) dispatcher (b) aggregator (c) black−box

Figure 5.7: Transformation classes

A simple procedure `TraceDS(`$\mathcal{T}, O^*, I$`)` in Figure 5.8 can be used to trace the lineage of a set of output items $O^* \subseteq O$ according to a dispatcher $\mathcal{T}$. The procedure applies $\mathcal{T}$ to the input data items one at a time and returns those items that produce one or more items in O^*.[2] Note that all of our tracing procedures are specified to take a set of output items as a parameter instead of a single output item, for generality and also so tracing procedures can be composed when we consider transformation sequences (Section 5.4) and graphs (Section 5.6).

Example 5.3.1 (Lineage Tracing for Dispatchers) Transformation $\mathcal{T}_1$ in Section 5.1 is a dispatcher, because each input order produces one or more output orders via $\mathcal{T}_1$. Given an output item $o = \langle \texttt{0101}, \texttt{AAA}, \texttt{2/1/1999}, \texttt{222}, \texttt{10} \rangle$ based on the sample data of Figure 5.2, we can trace o's lineage according to $\mathcal{T}_1$ using procedure `TraceDS(`$\mathcal{T}_1, \{o\}$`,Order)` to obtain $\mathcal{T}_1^*(o, \texttt{Order}) = \{\langle \texttt{0101}, \texttt{AAA}, \texttt{2/1/1999}, \text{“}\texttt{333(10)}, \texttt{222(10)}\text{”} \rangle\}$. Transformations $\mathcal{T}_2$, $\mathcal{T}_5$, $\mathcal{T}_6$, and $\mathcal{T}_7$ in Section 5.1 and $\mathcal{T}_8$ in Section 5.2.2 all are dispatchers, and we can similarly trace data lineage for them. □

`TraceDS` requires a complete scan of the input data set. For each input item i, it calls transformation $\mathcal{T}$ over $\{i\}$, which can be very expensive if $\mathcal{T}$ has significant overhead (e.g., startup time). In Section 5.3.6 we will discuss how indexes can be used to improve the performance of `TraceDS`. However, next we introduce a common subclass of dispatchers, *filters*, for which lineage tracing is trivial.

[2] For now we are assuming that the input set is readily available. Cases where the input set is unavailable or unnecessary will be considered later.

```
procedure TraceDS(T, O*, I)
  I* ← ∅;
  for each i ∈ I do
    if T({i}) ∩ O* ≠ ∅ then I* ← I* ⊎ {i};
return I*;
```

Figure 5.8: Tracing procedure for dispatchers

Filters. A dispatcher $\mathcal{T}$ is a *filter* if each input item produces either itself or nothing: $\forall i \in I$, $\mathcal{T}(\{i\}) = \{i\}$ or $\mathcal{T}(\{i\}) = \varnothing$. Thus, the lineage of any output data item is the same item in the input set: $\forall o \in O$, $\mathcal{T}^*(o) = \{o\}$. The tracing procedure for a filter $\mathcal{T}$ simply returns the traced item set O^* as its own lineage. It does not need to call the transformation $\mathcal{T}$ or scan the input data set, which can be a significant advantage in many cases (see Section 5.4). Transformation $\mathcal{T}_8$ in Section 5.2.2 is a filter, and the lineage of output item $o = \langle a, 2\rangle$ is the same item $\langle a, 2\rangle$ in the input set. Other examples of filters are $\mathcal{T}_2$ and $\mathcal{T}_6$ in Section 5.1.

Aggregators

A transformation $\mathcal{T}$ is *complete* if each input data item always contributes to some output data item: $\forall I \neq \varnothing$, $\mathcal{T}(I) \neq \varnothing$. A transformation $\mathcal{T}$ is an *aggregator* if $\mathcal{T}$ is complete, and for all I and $\mathcal{T}(I) = O = \{o_1, \ldots, o_n\}$, there exists a unique disjoint partition $I_1, \ldots, I_n$ of I such that $\mathcal{T}(I_k) = \{o_k\}$ for $k = 1..n$. $I_1, \ldots, I_n$ is called the *input partition*, and I_k is o_k's lineage according to $\mathcal{T}$: $\mathcal{T}^*(o_k, I) = I_k$. Figure 5.7(b) illustrates an aggregator, where the lineage of output item 1 is input items $\{1, 2\}$, the lineage of output item 2 is $\{3\}$, and the lineage of output item 3 is $\{4, 5, 6\}$.

Transformation $\mathcal{T}_9$ in Section 5.2.2 is an aggregator. The input partition is $I_1 = \{\langle a, -1\rangle, \langle a, 2\rangle\}$, $I_2 = \{\langle b, 0\rangle\}$, and the lineage of output item $o = \langle a, 2\rangle$ is I_1. Among the transformations in Section 5.1, $\mathcal{T}_4$, $\mathcal{T}_5$, and $\mathcal{T}_7$ are aggregators. Note that transformations can be both aggregators and dispatchers (e.g., $\mathcal{T}_5$ and $\mathcal{T}_7$ in Section 5.1). We will address how overlapping properties affect lineage tracing in Section 5.3.4.

To trace the lineage of an output subset O^* according to an aggregator $\mathcal{T}$, we can use the procedure `TraceAG(`$\mathcal{T}, O^*, I$`)` in Figure 5.9 that enumerates subsets of input I. It

```
procedure TraceAG(T, O*, I)
  L ← all subsets of I sorted by size;
  for each I* ∈ L in increasing order do
    if T(I*) = O* then
      if T(I − I*) = O − O* then break;
      else L = all supersets of I* sorted by size;
return I*;
```

Figure 5.9: Tracing procedure for aggregators

returns the unique subset I^* such that I^* produces exactly O^*, i.e., $\mathcal{T}(I^*) = O^*$, and the rest of the input set produces the rest of the output set, i.e., $\mathcal{T}(I - I^*) = O - O^*$. During the enumeration, we examine the subsets in increasing size. If we find a subset I' such that $\mathcal{T}(I') = O^*$ but $\mathcal{T}(I - I') \neq O - O^*$, we then need to examine only supersets of I', which can reduce the work significantly.

`TraceAG` may call $\mathcal{T}$ as many as $2^{|I|}$ times in the worst case, which can be prohibitive. We introduce two common subclasses of aggregators, *context-free aggregators* and *key-preserving aggregators*, which allow us to apply much more efficient tracing procedures.

Context-Free Aggregators. An aggregator $\mathcal{T}$ is *context-free* if any two input data items either always belong to the same input partition, or they always do not, regardless of the other items in the input set. In other words, a context-free aggregator determines the partition that an input item belongs to based on its own value, and not on the values of any other input items. All example aggregators we have seen are context-free. As an example of a non-context-free aggregator, consider a transformation $\mathcal{T}$ that clusters input data points based on their x-y coordinates and outputs some aggregate value of items in each cluster. Suppose $\mathcal{T}$ specifies that any two points within distance d from each other must belong to the same cluster. $\mathcal{T}$ is an aggregator, but it is not context-free, since whether two items belong to the same cluster or not may depend on the existence of a third item near to both.

We specify lineage tracing procedure `TraceCF(`$\mathcal{T}, O^*, I$`)` in Figure 5.10 for context-free aggregators. This procedure first scans the input data set to create the partitions (which we could not do linearly if the aggregator were not context-free). Then it checks each

```
procedure TraceCF(T, O*, I)
  I* ← ∅;
  pnum ← 0;
  for each i ∈ I do
    if pnum = 0 then I_1 ← {i}; pnum ← 1; continue;
    for (k ← 1; k ≤ pnum; k++) do
      if |T(I_k ∪ {i})| = 1 then I_k ← I_k ∪ {i}; break;
    if k > pnum then pnum ← pnum + 1; I_pnum ← {i};
  for k ← 1..pnum do
    if T(I_k) ⊆ O* then I* ← I* ∪ I_k;
return I*;
```

Figure 5.10: Tracing procedure for context-free aggregators

partition to find those that produce items in O^*. `TraceCF` reduces the number of transformation calls to $|I^2| + |I|$ in the worst case, which is a significant improvement.

Key-Preserving Aggregators. Suppose each input item and output item contains a unique *key* value in the relational sense, denoted $i.key$ for item i. An aggregator $\mathcal{T}$ is *key-preserving* if given any input set I and its input partition $I_1, \ldots, I_n$ for output $\mathcal{T}(I) = \{o_1, \ldots, o_n\}$, all subsets of I_k produce a single output item with the same key value as o_k, for $k = 1..n$. That is, $\forall I' \subseteq I_k$: $\mathcal{T}(I') = \{o'_k\}$ and $o'_k.key = o_k.key$.

Theorem 5.3.2 All key-preserving aggregators are context-free. □

Proof: Given a key-preserving aggregator $\mathcal{T}$, we want to prove that $\mathcal{T}$ is context-free, i.e, $\forall i$ and i', they either always belong to the same input partition or they always do not. Suppose for the sake of a contradiction that $\mathcal{T}$ is not context-free. Then there exist input sets I and I' and items i and i' in both I and I' such that

1. i and i' belong to the same input partition I_j in transformation instance $\mathcal{T}(I) = O$. Let $\mathcal{T}(I_j) = o_j$. According to the definition of key-preserving aggregator, $\mathcal{T}(\{i\})$ and $\mathcal{T}(\{i'\})$ produce output items with the same key value as $o_j.key$.

2. i and i' belong to different input partitions I_j and I_k respectively in $\mathcal{T}(I') = O'$. Let $\mathcal{T}(I_j) = o_j$ and $\mathcal{T}(I_k) = o_k$. According to the definition of key-preserving

```
procedure TraceKP(T, O*, I)
  I* ← ∅;
  for each i ∈ I do
    if π_key(T({i})) ⊆ π_key(O*) then I* ← I* ⊎ {i};
return I*;
```

Figure 5.11: Tracing procedure for key-preserving aggregators

aggregator, $\mathcal{T}(\{i\})$ produces an item with key value $o_k.key$, while $\mathcal{T}(\{i'\})$ produces an item with key value $o_j.key$.

Since $o_j.key \neq o_k.key$ by the definition of a key, we obtain our contradiction. □

All example aggregators we have seen are key-preserving. As an example of a context-free but non-key-preserving aggregator, consider a relational groupby-aggregation that does not retain the grouping attribute.

`TraceKP(`$\mathcal{T}, O^*, I$`)` in Figure 5.11 traces the lineage of O^* according to a key-preserving aggregator $\mathcal{T}$. It scans the input data set once and returns all input items that produce output items with the same key as items in O^*. `TraceKP` reduces the number of transformation calls to $|I|$, with each call operating on a single input data item. We can further improve performance of `TraceKP` using an index, as discussed in Section 5.3.6.

Black-box Transformations

An atomic transformation is called a *black-box* transformation if it is neither a dispatcher nor an aggregator, and it does not have a *provided lineage tracing procedure* (Section 5.3.3). In general, any subset of the input items may have been used to produce a given output item through a black-box transformation, as illustrated in Figure 5.7(c), so all we can say is that the entire input data set is the lineage of each output item: $\forall o \in O$, $\mathcal{T}^*(o, I) = I$. Thus, the tracing procedure for a black-box transformation simply returns the entire input I.

As an example of a true black-box, consider a transformation $\mathcal{T}$ that sorts the input data items and attaches a serial number to each output item according to its sorted position. For instance, given input data set $I = \{\langle f, 10\rangle, \langle b, 20\rangle, \langle c, 5\rangle\}$ and sorting by the first attribute, the output is $\mathcal{T}(I) = \{\langle \mathbf{1}, b, 20\rangle, \langle \mathbf{2}, c, 5\rangle, \langle \mathbf{3}, f, 10\rangle\}$, and the lineage of each output data

item is the entire input set I. Note that in this case each output item, in particular its serial number, is indeed derived from all input data items.

5.3.2 Schema Mappings

Schema information can be very useful in the ETL process, and many data warehousing systems require transformation programmers to provide some schema information. In this section, we discuss how we can use schema information to improve lineage tracing for dispatchers and aggregators. Sometimes schema information also can improve lineage tracing for a black-box transformation $\mathcal{T}$, specifically when $\mathcal{T}$ can be combined with another non-black-box transformation based on $\mathcal{T}$'s schema information (Section 5.4). A schema specification may include:

input schema $\mathbf{A} = \langle A_1, \ldots, A_p \rangle$, and *input key* $A_{key} \subseteq \mathbf{A}$

output schema $\mathbf{B} = \langle B_1, \ldots, B_q \rangle$, and *output key* $B_{key} \subseteq \mathbf{B}$

The specification also may include *schema mappings*, defined as follows.

Definition 5.3.3 (Schema Mappings) Consider a transformation $\mathcal{T}$ with input schema $\mathbf{A}$ and output schema $\mathbf{B}$. Let $A \subseteq \mathbf{A}$ and $B \subseteq \mathbf{B}$ be lists of input and output attributes. Let $i.A$ denote the A attribute values of i, and similarly for $o.B$. Let f and g be functions from tuples of attribute values to tuples of attribute values. We say that $\mathcal{T}$ has a *forward schema mapping* $f(A) \xrightarrow{\mathcal{T}} B$ if we can partition any input set I into $I_1, \ldots, I_m$ based on equality of $f(A)$ values,[3] and partition the output set $O = \mathcal{T}(I)$ into $O_1, \ldots, O_n$ based on equality of B values, such that $m \geq n$ and:

1. for $k = 1..n$, $\mathcal{T}(I_k) = O_k$ and $I_k = \{i \in I \mid f(i.A) = o.B \text{ for some } o \in O_k\}$.
2. for $k = (n+1)..m$, $\mathcal{T}(I_k) = \varnothing$.

Similarly, we say that $\mathcal{T}$ has a *backward schema mapping* $A \xleftarrow{\mathcal{T}} g(B)$ if we can partition any input set I into $I_1, \ldots, I_m$ based on equality of A values, and partition the output set $O = \mathcal{T}(I)$ into $O_1, \ldots, O_n$ based on equality of $g(B)$ values, such that $m \geq n$ and:

1. for $k = 1..n$, $\mathcal{T}(I_k) = O_k$ and $I_k = \{i \in I \mid i.A = g(o.B) \text{ for some } o \in O_k\}$.

[3] That is, two input items $i_1 \in I$ and $i_2 \in I$ are in the same partition I_k iff $f(i_1.A) = f(i_2.A)$.

2. for $k = (n+1)..m$, $\mathcal{T}(I_k) = \varnothing$.

When f (or g) is the identity function, we simply write $A \xrightarrow{\mathcal{T}} B$ (or $A \xleftarrow{\mathcal{T}} B$). If $A \xrightarrow{\mathcal{T}} B$ and $A \xleftarrow{\mathcal{T}} B$ we write $A \xleftrightarrow{\mathcal{T}} B$. □

Although Definition 5.3.3 may seem cumbersome, it formally and accurately captures the intuitive notion of schema mappings (certain input attributes producing certain output attributes) that transformations do frequently exhibit.

Example 5.3.4 Schema information for transformation $\mathcal{T}_5$ in Section 5.1 can be specified as:

Input schema and key: $\mathbf{A} = \langle \texttt{prod-name}, \texttt{q1}, \texttt{q2}, \texttt{q3}, \texttt{q4} \rangle$, $A_{key} = \langle \texttt{prod-name} \rangle$
Output schema and key: $\mathbf{B} = \langle \texttt{prod-name}, \texttt{q1}, \texttt{q2}, \texttt{q3}, \texttt{avg3}, \texttt{q4} \rangle$. $B_{key} = \langle \texttt{prod-name} \rangle$
Schema mappings: $\langle \texttt{prod-name}, \texttt{q1}, \texttt{q2}, \texttt{q3}, \texttt{q4} \rangle \xleftrightarrow{\mathcal{T}_5} \langle \texttt{prod-name}, \texttt{q1}, \texttt{q2}, \texttt{q3}, \texttt{q4} \rangle$
$f(\langle \texttt{q1}, \texttt{q2}, \texttt{q3} \rangle) \xrightarrow{\mathcal{T}_5} \langle \texttt{avg3} \rangle$, where $f(\langle a, b, c \rangle) = (a+b+c)/3$ □

Theorem 5.3.5 Consider a transformation $\mathcal{T}$ that is a dispatcher or an aggregator, and consider any instance $\mathcal{T}(I) = O$. Given any output item $o \in O$, let I^* be o's lineage according to $\mathcal{T}$. If $\mathcal{T}$ has a forward schema mapping $f(A) \xrightarrow{\mathcal{T}} B$, then $I^* \subseteq \{i \in I \mid f(i.A) = o.B\}$. If $\mathcal{T}$ has a backward schema mapping $A \xleftarrow{\mathcal{T}} g(B)$, then $I^* \subseteq \{i \in I \mid i.A = g(o.B)\}$. □

Proof: Consider a dispatcher or aggregator $\mathcal{T}$ and an instance $\mathcal{T}(I) = O$. We prove that if $\mathcal{T}$ has a forward schema mapping $f(A) \xrightarrow{\mathcal{T}} B$, then the lineage of any output item $o \in O$ is a subset of $\{i \in I \mid f(i.A) = o.B\}$. The case for $\mathcal{T}$ with backward schema mapping $A \xleftarrow{\mathcal{T}} g(B)$ can be proved in the same manner.

If $\mathcal{T}$ is a dispatcher, o's lineage is $\{i\}$ such that $o \in \mathcal{T}(\{i\})$, according to the lineage definition for dispatchers in Section 5.3.1. Thus, we just need to prove $f(i.A) = o.B$. Consider transformation instance $\mathcal{T}(\{i\}) = O'$. By Definition 5.3.3 of schema mappings, this transformation instance has a single input partition, i.e., $m = 1$, because $\{i\}$ contains a single item. Furthermore, $n \geq 1$ because O' is nonempty. Since $m \geq n$, we have $m = n = 1$. Thus, according to Definition 5.3.3, $f(i.A) = o.B$.

Now suppose $\mathcal{T}$ is an aggregator. From Definition 5.3.3 we can partition instance $\mathcal{T}(I) = O$ according to the schema mapping to obtain input and output partitions $I_1, \ldots, I_m$

and $O_1, \ldots, O_n$. Since aggregators are complete, we know $m = n$. For $k = 1..n$, the pair of partitions I_k and O_k generate a transformation instance $\mathcal{T}(I_k) = O_k = \{o_k^1, \ldots, o_k^{l_k}\}$. From the definition of aggregator, given transformation instance $\mathcal{T}(I_k) = O_k$, there exists a unique partition $I_k^1 \ldots, I_k^{l_k}$ of I_k such that $\mathcal{T}(I_k^j) = \{o_k^j\}$ for $j = 1..l_k$, and $f(i.A) = o_k^j.B$ for all $i \in I_k^j$. Considering the entire input set I and $\mathcal{T}(I) = O = \{o_1^1, \ldots, o_n^{l_n}\}$, $I_1^1, \ldots, I_n^{l_n}$ is a partition of I such that $\mathcal{T}(I_k^j) = \{o_k^j\}$, so it is the unique such partition. According to the lineage definition for aggregators in Section 5.3.1, given an output data item $o = o_k^j \in O$, o_k^j's lineage is I_k^j. Since $f(i.A) = o_k^j.B$ for all $i \in I_k^j$, and $o = o_j^k$, we conclude $I_k^j \subseteq \{i \in I \mid f(i.A) = o.B\}$. □

Based on Theorem 5.3.5, when tracing lineage for a dispatcher or aggregator, we can narrow down the lineage of any output data item to a (possibly very small) subset of the input data set based on a schema mapping. We can then retrieve the exact lineage within that subset using the algorithms in Section 5.3.1. For example, consider an aggregator $\mathcal{T}$ with a backward schema mapping $A \overset{\mathcal{T}}{\leftarrow} g(B)$. When tracing the lineage of an output item $o \in O$ according to $\mathcal{T}$, we can first find the input subset $I' = \{i \in I \mid i.A = g(o.B)\}$, then enumerate subsets of I' using `TraceAG(`$\mathcal{T}, o, I'$`)` to find o's lineage $I^* \subseteq I'$. If we have multiple schema mappings for $\mathcal{T}$, we can use the intersection of the subsets for improved tracing efficiency.

The narrowing technique of the previous paragraph is an improvement over the tracing procedures based on lineage definition (Section 5.3.1). However, when schema mappings satisfy certain additional conditions, we can do even better.

Definition 5.3.6 (Schema Mapping Properties) Consider a transformation $\mathcal{T}$ with input schema $\mathbf{A}$, input key A_{key}, output schema $\mathbf{B}$, and output key B_{key}.

1. $\mathcal{T}$ is a *forward key-map (fkmap)* if it is complete ($\forall I \neq \varnothing$, $\mathcal{T}(I) \neq \varnothing$) and it has a forward schema mapping to the output key: $f(A) \overset{\mathcal{T}}{\rightarrow} B_{key}$.

2. $\mathcal{T}$ is a *backward key-map (bkmap)* if it has a backward schema mapping to the input key: $A_{key} \overset{\mathcal{T}}{\leftarrow} g(B)$.

3. $\mathcal{T}$ is a *backward total-map (btmap)* if it has a backward schema mapping to all input attributes: $\mathbf{A} \overset{\mathcal{T}}{\leftarrow} g(B)$. □

Suppose that schema information and mappings are given for all transformations in Section 5.1. Then all of the transformations except $\mathcal{T}_4$ are backward key-maps; $\mathcal{T}_2$, $\mathcal{T}_5$, and $\mathcal{T}_6$ are backward total-maps; $\mathcal{T}_4$, $\mathcal{T}_5$, and $\mathcal{T}_7$ are forward key-maps.

Theorem 5.3.7 (1) All filters are backward total-maps. (2) All backward total-maps are backward key-maps. (3) All backward key-maps are dispatchers. (4) All forward key-maps are key-preserving aggregators. □

Proof:

1. Given a filter $\mathcal{T}$, each input item either produces itself or nothing, i.e., $\forall i \in I$: $\mathcal{T}(\{i\}) = \{i\}$ or $\varnothing$. Thus, I and O always have the same schema, call it $\mathbf{A}$, and we can partition any input I and output $O = \mathcal{T}(I)$ into singleton sets $I_1, \ldots, I_m$ and $O_1, \ldots, O_n$ such that $m \geq n$ and:

   (a) for $k = 1..n$, $\mathcal{T}(I_k) = O_k$, and I_k and O_k both contain the same single item, so $I_k = \{i \in I \mid i = o \text{ for some } o \in O_k\}$.

   (b) for $k = (n+1)..m$, $\mathcal{T}(I_k) = \varnothing$.

   Therefore, we have a backward schema mapping $\mathbf{A} \overset{\mathcal{T}}{\leftarrow} \mathbf{A}$ to all input attributes, so $\mathcal{T}$ is a backward total-map.

2. Consider a backward total-map $\mathcal{T}$ where the input has schema $\mathbf{A}$, so we have $\mathbf{A} \overset{\mathcal{T}}{\leftarrow} g(B)$. Since $\mathbf{A}$ is an input key (we are assuming set semantics), $\mathcal{T}$ is a backward key-map.

3. Let $\mathcal{T}$ be a backward key-map with $A_{key} \overset{\mathcal{T}}{\leftarrow} g(B)$. We want to prove that $\mathcal{T}$ is a dispatcher, i.e., $\forall I$: $\mathcal{T}(I) = \bigcup_{i \in I} \mathcal{T}(\{i\})$. Given any instance $\mathcal{T}(I) = O$, according to Definition 5.3.3 of schema mapping, we can partition I into $I_1, \ldots, I_m$ based on A_{key} equality and partition O into $O_1, \ldots, O_n$ based on $g(B)$ equality such that $\mathcal{T}(I_k) = O_k$ for $k = 1..n$ and $\mathcal{T}(I_k) = \varnothing$ for $k = (n+1)..m$. Thus, $\mathcal{T}(I) = \bigcup_{k=1..n} O_k = \bigcup_{k=1..m} \mathcal{T}(I_k)$. Since each I_k contains items with the same A_{key} value, it contains a single item by the definition of key. Therefore, $\mathcal{T}(I) = \bigcup_{i \in I} \mathcal{T}(\{i\})$, and $\mathcal{T}$ is a dispatcher.

4. Let $\mathcal{T}$ be a forward key-map with schema mapping $f(A) \xrightarrow{\mathcal{T}} B_{key}$. We want to prove that $\mathcal{T}$ is a key-preserving aggregator. We first prove that $\mathcal{T}$ is an aggregator, i.e., $\mathcal{T}$ is complete and given any instance $\mathcal{T}(I) = O = \{o_1, \ldots, o_n\}$, there exists a unique partition $I_1, \ldots, I_n$ of I such that $\mathcal{T}(I_k) = \{o_k\}$. By Definition 5.3.6 of forward key-map, $\mathcal{T}$ is complete. Consider any $\mathcal{T}(I) = O$. From Definition 5.3.3 of schema mapping we obtain input and output partitions $I_1, \ldots, I_m$ and $O_1, \ldots, O_n$, where $m = n$ since $\mathcal{T}$ is complete. We will prove that $I_1, \ldots, I_n$ is the unique input partition for $\mathcal{T}(I) = O$ as specified in the aggregator definition (Section 5.3.1). Since each O_k, $k = 1..n$, contains items with the same B_{key} value, it contains a single item; let it be o_k. Then, $I_1, \ldots, I_n$ is an input partition such that $\mathcal{T}(I_k) = \{o_k\}$, and by Definition 5.3.3 $I_k = \{i \in I \mid f(i.A) = o_k.B_{key}\}$. We now prove that this input partition is unique. Suppose there exists another input partition $I'_1, \ldots, I'_n$ such that $\mathcal{T}(I'_k) = \{o_k\}$ for $k = 1..n$. Consider transformation instance $\mathcal{T}(I'_k) = \{o_k\}$ for any given k. Because its output contains a single item and $\mathcal{T}$ is complete, this transformation instance has a single input partition and a single output partition based on $\mathcal{T}$'s schema mapping. From Definition 5.3.3, we know that $\forall i \in I'_k$: $f(i.A) = o_k.B_{key}$. Thus, $I'_k \subseteq I_k$, for $k = 1..n$. Since $I_1 \cup \cdots \cup I_n = I'_1 \cup \cdots \cup I'_n = I$ and the partitions are disjoint, we know that $I_k = I'_k$ for $k = 1..n$. Therefore, $I_1, \ldots, I_n$ is unique, and $\mathcal{T}$ is an aggregator.

   We now prove that $\mathcal{T}$ is key-preserving. Consider instance $\mathcal{T}(I) = O$ with aggregator partition $I_1, \ldots, I_n$ for output $\{o_1, \ldots, o_n\}$. $\forall I'_k \subseteq I_k$: $\mathcal{T}(I'_k) = \{o'_k\}$ where $o'_k.B_{key} = o_k.B_{key}$, $k = 1..n$. Consider any $k = 1..n$ and any $I'_k \subseteq I_k$ and let $O'_k = \mathcal{T}(I'_k)$. Since all items in I'_k have the same $f(i.A)$, according to Definition 5.3.3, we can partition this instance based on the schema mapping to obtain a single input partition and a single output partition. Thus, O'_k contains items with the same $o.B_{key}$, which means O'_k contains a single item; let it be o'_k. Then, $\forall i \in I'_k$: $f(i.A) = o'_k.B_{key}$. Since $I'_k \subseteq I_k$, $\forall i \in I'_k$: $i \in I_k$. Further because $\forall i \in I_k$: $f(i.A) = o_k.B_{key}$, we know that $o'_k.B_{key} = o_k.B_{key}$. Therefore, $\mathcal{T}$ is key-preserving. □

Theorem 5.3.8 Consider a transformation instance $\mathcal{T}(I) = O$. Given an output item $o \in O$, let I^* be o's lineage based on $\mathcal{T}$'s transformation class as defined in Section 5.3.1.

1. If $\mathcal{T}$ is a forward key-map with schema mapping $f(A) \xrightarrow{\mathcal{T}} B_{key}$, then $I^* = \{i \in I \mid f(i.A) = o.B_{key}\}$.
2. If $\mathcal{T}$ is a backward key-map with schema mapping $A_{key} \xleftarrow{\mathcal{T}} g(B)$, then $I^* = \{i \in I \mid i.A_{key} = g(o.B)\}$.
3. If $\mathcal{T}$ is a backward total-map with schema mapping $\mathbf{A} \xleftarrow{\mathcal{T}} g(B)$, then $I^* = \{g(o.B)\}$.

□

Proof:

1. Let $\mathcal{T}$ be a forward key-map with schema mapping $f(A) \xrightarrow{\mathcal{T}} B_{key}$. Consider an instance $\mathcal{T}(I) = O = \{o_1, \ldots, o_n\}$. We want to prove that $\forall o_k$, o_k's lineage is $\{i \in I \mid f(i.A) = o_k.B_{key}\}$. According to Theorem 5.3.7, $\mathcal{T}$ is an aggregator, and from part (4) of our proof for Theorem 5.3.7, we know that partition $I_1, \ldots, I_n$ where $I_k = \{i \in I \mid f(i.A) = o_k.B_{key}\}$ is the unique input partition such that $\mathcal{T}(I_k) = \{o_k\}$. According to the lineage definition for aggregators, o_k's lineage is $I_k = \{i \in I \mid f(i.A) = o_k.B_{key}\}$.

2. Let $\mathcal{T}$ be a backward key-map with schema mapping $A_{key} \xleftarrow{\mathcal{T}} g(B)$. Consider an instance $\mathcal{T}(I) = O$. We want to prove that $\forall o \in O$, o's lineage is $\{i \in I \mid i.A_{key} = g(o.B)\}$. According to Theorem 5.3.7, $\mathcal{T}$ is a dispatcher. Thus, according to the lineage definition for dispatchers, o's lineage is $I^* = \{i \in I \mid o \in \mathcal{T}(\{i\})\}$. According to Theorem 5.3.5, I^* is a subset of $I' = \{i \in I \mid i.A_{key} = g(o.B)\}$. Since I' contains a single item by the definition of key, I^* is either empty or contains a single item. We will prove that I^* is nonempty by contradiction, from which we can conclude that $I^* = I' = \{i \in I \mid i.A_{key} = g(o.B)\}$. Suppose that I^* is empty. Then, $\forall i \in I$, $o \notin \mathcal{T}(\{i\})$. Therefore, since $\mathcal{T}$ is a dispatcher according to Theorem 5.3.7, $o \notin \bigcup_{i \in I} \mathcal{T}(\{i\}) = \mathcal{T}(I)$, which contradicts $o \in O$.

3. Let $\mathcal{T}$ be a backward total-map with schema mapping $\mathbf{A} \xleftarrow{\mathcal{T}} g(B)$ to the entire input schema $\mathbf{A}$. We want to prove that $\forall I$ and $\forall o \in O = \mathcal{T}(I)$, o's lineage is $\{g(o.B)\}$. Since a backward total-map is a backward key-map, by part(2) of this proof we know that the lineage of o is $I^* = \{i \in I \mid i.A = g(o.B)\}$. Since $i.A = i$ for all $i \in I$, $I^* = \{g(o.B)\}$. □

procedure TraceFM($\mathcal{T}, O^*, I$) // let $f(A) \xrightarrow{\mathcal{T}} B_{key}$
  $I^* \leftarrow \varnothing$;
  for each $i \in I$ **do**
    if $f(i.A) \in \pi_{B_{key}}(O^*)$ **then** $I^* \leftarrow I^* \uplus \{i\}$;
return I^*;

procedure TraceBM($\mathcal{T}, O^*, I$) // let $A_{key} \xleftarrow{\mathcal{T}} g(B)$
  $I^* \leftarrow \varnothing$;
  for each $i \in I$ **do**
    if $i.A_{key} \in \pi_{g(B)}(O^*)$ **then** $I^* \leftarrow I^* \uplus \{i\}$;
return I^*;

procedure TraceTM($\mathcal{T}, O^*$) // let $\mathbf{A} \xleftarrow{\mathcal{T}} g(B)$
return $\pi_{g(B)}(O^*)$;

Figure 5.12: Tracing procedures using schema mappings

According to Theorem 5.3.8, we can use the tracing procedures shown in Figure 5.12 for transformations with the schema mapping properties specified in Definition 5.3.6. For example, procedure `TraceFM(`$\mathcal{T}, O^*, I$`)` performs lineage tracing for a forward key-map $\mathcal{T}$, which by Theorem 5.3.7 also could be traced using procedure `TraceKP` of Figure 5.11. Both algorithms scan each input item once, however `TraceKP` applies transformation $\mathcal{T}$ to each item, while `TraceFM` applies function f to some attributes of each item. Certainly f is very unlikely to be more expensive than $\mathcal{T}$, since $\mathcal{T}$ effectively computes f and may do other work as well; f may in fact be quite a bit cheaper. `TraceBM(`$\mathcal{T}, O^*, I$`)` uses a similar approach for a backward key-map, and is usually more efficient than `TraceDS(`$\mathcal{T}, O^*, I$`)` of Figure 5.8 for the same reasons. `TraceTM(`$\mathcal{T}, O^*$`)` performs lineage tracing for a backward total-map, which is very efficient since it does not need to scan the input data set and makes no transformation calls.

Example 5.3.9 Considering some examples from Section 5.1:

- $\mathcal{T}_1$ is a backward key-map with schema mapping `order-id` $\xleftarrow{\mathcal{T}_1}$ `order-id`. We can trace the lineage of an output data item o using `TraceBM`, which simply retrieves items in `Order` that have the same `order-id` as o.

- $\mathcal{T}_4$ is a forward key-map with schema mapping `prod-name` $\stackrel{\mathcal{T}_4}{\rightarrow}$ `prod-name`. We can trace the lineage of an output data item o using `TraceFM`, which simply retrieves the input items that have the same `prod-name` as o.

- $\mathcal{T}_5$ is a backward total-map with $\langle$`prod-name`, `q1`, `q2`, `q3`, `q4`$\rangle \stackrel{\mathcal{T}_5}{\leftarrow} \langle$`prod-name`, `q1`, `q2`, `q3`, `q4`$\rangle$. We can trace the lineage of an output data item o using `TraceTM`, which directly constructs $\langle o.$`prod-name`, $o.$`q1`, $o.$`q2`, $o.$`q3`, $o.$`q4`$\rangle$ as o's lineage. □

In Section 5.3.6 we will discuss how indexes can be used to further speed up procedures `TraceFM` and `TraceBM`.

5.3.3 Provided Tracing Procedure or Transformation Inverse

If we are very lucky, a lineage *tracing procedure* may be provided along with the specification of a transformation $\mathcal{T}$. The tracing procedure `TP` may require access to the input data set, i.e., `TP`(O^*, I) returns O^*'s lineage according to $\mathcal{T}$, or the tracing procedure may not require access to the input, i.e., `TP`(O^*) returns O^*'s lineage. A related but not identical situation is when we are provided with the *inverse* for a transformation $\mathcal{T}$. Sometimes, but not always, the inverse of $\mathcal{T}$ can be used as $\mathcal{T}$'s tracing procedure.

Definition 5.3.10 (Inverse Transformation) A transformation $\mathcal{T}$ is *invertible* if there exists a transformation $\mathcal{T}^{-1}$ such that $\forall I$, $\mathcal{T}^{-1}(\mathcal{T}(I)) = I$, and $\forall O$, $\mathcal{T}(\mathcal{T}^{-1}(O)) = O$. $\mathcal{T}^{-1}$ is called $\mathcal{T}$'s *inverse*. □

Theorem 5.3.11 If a transformation $\mathcal{T}$ is an aggregator with inverse $\mathcal{T}^{-1}$, then for all instances $\mathcal{T}(I) = O$ and all $o \in O$, the lineage of o according to $\mathcal{T}$ is $\mathcal{T}^{-1}(\{o\})$. □

Proof: Consider an aggregator $\mathcal{T}$ with inverse $\mathcal{T}^{-1}$. We want to prove that given any instance $\mathcal{T}(I) = O$ and $o \in O$, o's lineage is $\mathcal{T}^{-1}(\{o\})$. Since $\mathcal{T}$ is an aggregator, according to the definition of aggregator in Section 5.3.1, for $\mathcal{T}(I) = O = \{o_1, \ldots, o_n\}$ there exists a unique partition $I_1, \ldots, I_n$ of I such that $\mathcal{T}(I_k) = \{o_k\}$ for $k = 1..n$, and I_k is o_k's lineage. According to Definition 5.3.10 $\mathcal{T}^{-1}(\mathcal{T}(I)) = I$. Therefore, $\forall o_k \in O$, $\mathcal{T}^{-1}(\{o_k\}) = \mathcal{T}^{-1}(\mathcal{T}(I_k)) = I_k$, which is o_k's lineage. □

Note that in fact a full inverse $\mathcal{T}^{-1}$ is not needed for Theorem 5.3.11 to hold, but only a *weak inverse*. A weak inverse guarantees $\forall I$, $\mathcal{T}^{-1}(\mathcal{T}(I)) = I$, but not necessarily $\forall O$, $\mathcal{T}(\mathcal{T}^{-1}(O)) = O$.

According to Theorem 5.3.11, we can use a transformation's inverse for lineage tracing if the invertible transformation is an aggregator, as we will illustrate in Example 5.3.12(1). However, if the invertible transformation is a dispatcher or black-box, we cannot always use its inverse for lineage tracing, as we will illustrate in Example 5.3.12(2).

Example 5.3.12 (Lineage Tracing Using Inverses)

1. Consider a transformation $\mathcal{T}$ that performs list merging, essentially the opposite of transformation $\mathcal{T}_1$ in Section 5.1. $\mathcal{T}$ takes a two-attribute input set and produces a two-attribute output set. It groups the input set according to the first attribute. For each group, it produces one output item containing the grouping value along with a list of the second attribute values from the input. For instance, given input data set $I = \{\langle 1, a\rangle, \langle 1, c\rangle, \langle 2, b\rangle, \langle 2, g\rangle\, \langle 2, h\rangle\}$, the output is $O = \mathcal{T}(I) = \{\langle 1, \text{“}a, c\text{”}\rangle, \langle 2, \text{“}b, g, h\text{”}\rangle\}$. $\mathcal{T}$ has an inverse $\mathcal{T}^{-1}$ which splits the second attribute of its input data items to produce multiple output items, i.e., $\mathcal{T}^{-1}(O) = I$.

   $\mathcal{T}$ is an aggregator, so according to Theorem 5.3.11 we can perform lineage tracing for $\mathcal{T}$ by applying its inverse $\mathcal{T}^{-1}$ to the traced data item(s). For example, given output item $o = \langle 2, \text{“}b, g, h\text{”}\rangle \in O$, o's lineage is $\mathcal{T}^{-1}(\{o\}) = \{\langle 2, b\rangle, \langle 2, g\rangle, \langle 2, h\rangle\}$.

2. Now consider transformation $\mathcal{T}_1$ from Section 5.1, which is a dispatcher and has an inverse $\mathcal{T}_1^{-1}$ that assembles lists in a similar manner to transformation $\mathcal{T}$ as described in (1). If we apply inverse transformation $\mathcal{T}_1^{-1}$ to output item $\langle \texttt{0101}, \texttt{AAA}, \texttt{2/1/1999}, 222, \texttt{10}\rangle$ then we obtain $\{\langle \texttt{0101}, \texttt{AAA}, \texttt{2/1/1999}, \text{“}\texttt{222(10)}\text{”}\rangle\}$, instead of the correct lineage $\{\langle \texttt{0101}, \texttt{AAA}, \texttt{2/1/1999}, \text{“}\texttt{333(10)}, \texttt{222(10)}\text{”}\rangle\}$. □

Although we can guarantee very little about the accuracy or efficiency of provided tracing procedures or transformation inverses in the general case, it is our experience that, when provided, they are usually the most effective way to perform lineage tracing. We will make this assumption in the remainder of the chapter.

Property	Tracing Procedure	# of Transformation Calls	# of Input Accesses
dispatcher	`TraceDS`	$\|I\|$	$\|I\|$
filter	**return** o	0	0
aggregator	`TraceAG`	$O(2^{\|I\|})$	$O(2^{\|I\|})$
context-free aggregator	`TraceCF`	$O(\|I\|^2)$	$O(\|I\|^2)$
key-preserving aggregator	`TraceKP`	$\|I\|$	$\|I\|$
black-box	**return** I	0	0
forward key-map	`TraceFM`	0	$\|I\|$
backward key-map	`TraceBM`	0	$\|I\|$
backward total-map	`TraceTM`	0	0
tracing procedure requiring input	`TP`	?	?
tracing procedure not requiring input	`TP`	?	0

Table 5.1: Summary of transformation properties

5.3.4 Transformation Property Summary and Hierarchy

Table 5.1 summarizes the transformation properties covered in the previous three sections. The table specifies which tracing procedure is applicable for each property, along with the number of transformation calls and number of input data item accesses for each procedure. We omit transformation inverses from the table, since when applicable they are equivalent to a provided tracing procedure not requiring input.

As discussed earlier, a transformation may satisfy more than one property. Some properties are better than others: tracing procedures may be more efficient, they may return a more accurate lineage result, or they may not require access to input data. Figure 5.13 specifies a hierarchy for determining which property is best to use for a specific transformation. In the hierarchy, a solid arrow from property p_1 to p_2 means that p_2 is more restrictive than p_1, i.e., all transformations that satisfy property p_2 also satisfy property p_1. Further, according to Table 5.1, whenever p_2 is more restrictive than p_1, the tracing procedure for p_2 is no less efficient (and usually more efficient) by any measure: number of transformation calls, number of input accesses, and whether the input data is required at all. (Black-box transformations, which are the only type with less accurate lineage results, are placed in a separate branch of the hierarchy.) A dashed arrow from property p_1 to p_2 in the hierarchy means that even though p_2 is not strictly more restrictive than p_1, p_2 does yield a tracing

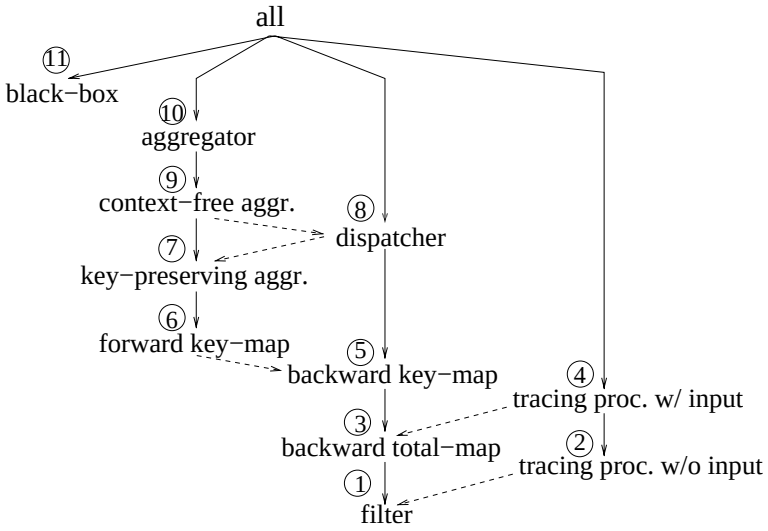


Figure 5.13: Transformation property hierarchy

procedure that again is no less efficient (and usually more efficient) by any measure.[4]

Let us make the reasonable assumption that a provided tracing procedure requiring input is more efficient than `TraceBM`, and that a tracing procedure not requiring input is more efficient than `TraceTM`. Then we can derive a total order of the properties as shown by the numbers in Figure 5.13: the lower the number, the better the property is for lineage tracing. Given a set of properties for a transformation $\mathcal{T}$, we always use the best one, i.e., the one with the lowest number, to trace data lineage for $\mathcal{T}$. Table 5.2 lists the best property for example transformations $\mathcal{T}_1$–$\mathcal{T}_7$ from Section 5.1 and $\mathcal{T}_8$–$\mathcal{T}_9$ from Section 5.2.2, along with other properties satisfied by these transformations. Note that we list only the most restrictive property on each branch of the hierarchy.

5.3.5 Nondeterministic Transformations

Recall from Section 5.2 that we have assumed all transformations to be deterministic. The reason we sometimes require determinism is that several of our tracing procedures call

[4] In some cases the tracing efficiency difference represented by a solid or dashed arrow is significant, while in other cases it is less so. This issue is discussed further in Section 5.4.

	Best Property	Additional Properties
$\mathcal{T}_1$	backward key-map	
$\mathcal{T}_2$	filter	
$\mathcal{T}_3$	see Section 5.5	
$\mathcal{T}_4$	forward key-map	
$\mathcal{T}_5$	backward total-map	forward key-map
$\mathcal{T}_6$	filter	
$\mathcal{T}_7$	backward key-map	forward key-map
$\mathcal{T}_8$	filter	
$\mathcal{T}_9$	forward key-map	

Table 5.2: Properties of $\mathcal{T}_1$–$\mathcal{T}_9$

transformation $\mathcal{T}$, often repeatedly, as part of the lineage tracing process, specifically procedures `TraceAG`, `TraceCF`, `TraceKP`, and `TraceDS`. Those procedures that do not call transformation $\mathcal{T}$—`TraceBM`, `TraceTM`, `TraceFM`, tracing procedures for filters and black-boxes, and user-provided tracing procedures—do not require determinism. Note that a transformation that selects a random sample of the input is a filter, and a transformation that attaches timestamps to tuples is a backward total-map, so neither of these common nondeterministic transformations poses a problem in our approach.

5.3.6 Improving Tracing Performance Using Indexes

Several of the lineage tracing algorithms presented in Sections 5.3.1–5.3.3 can be sped up if we can build indexes on the input data set. We consider two types of indexes:

- *Conventional indexes*, which allow us to quickly locate data items matching a given value. We can use a conventional index on a key for I to speed up procedure `TraceBM`, as well as the schema mapping "narrowing down" technique in Section 5.3.2.

- *Functional indexes* (e.g., [Ora]), which are constructed for a given function F and allow us to quickly locate data items i such that $F(i) = V$ for a value V. We can use a functional index with $F = f$, where f is the schema mapping function, to speed up procedure `TraceFM` (and again the schema mapping "narrowing down" in Section 5.3.2). We can also use a functional index with $F = \mathcal{T}$ to speed up procedures `TraceDS` and `TraceKP`.

Of course we could also build a complete *lineage index*, which maps the key of an output data item o to the set of input items that comprise o's lineage. This approach is similar to using *annotations* to record instance-level lineage, which can be very expensive and tends to be worthwhile only for lineage-intensive warehouses. We will discuss annotation-based lineage tracing further in Section 6.1.1. Some intermediate kinds of indexes—less expensive than lineage indexes but more specialized than the two index types discussed above—may also be beneficial for some of our tracing procedures, but are not explored further in this thesis.

5.4 Lineage Tracing through a Transformation Sequence

Now that we have specified how to perform lineage tracing for a single transformation with one input set and one output set, we will consider lineage tracing for sequences of such transformations. Multiple input and output sets are discussed in Section 5.5, and arbitrary acyclic transformation graphs are covered in Section 5.6.

5.4.1 Data Lineage for a Transformation Sequence

Consider a simple sequence of two transformations, such as $\mathcal{T}_1 \circ \mathcal{T}_2$ in Figure 5.14 composed from Figures 5.7(a) and 5.7(b). For an input data set I, let $I_2 = \mathcal{T}_1(I)$ and $O = \mathcal{T}_2(I_2)$. Given an output data item $o \in O$, if $I_2^* \subseteq I_2$ is the lineage of o according to $\mathcal{T}_2$, and $I^* \subseteq I$ is the lineage of I_2^* according to $\mathcal{T}_1$, then I^* is the lineage of o according to $\mathcal{T}_1 \circ \mathcal{T}_2$. For example, in Figure 5.14 if $o \in O$ is item 3, then I_2^* is items $\{4, 5, 6\}$ in I_2, and I^* is items $\{1, 3\}$ in I. This lineage definition generalizes to arbitrarily long transformation sequences using the associativity of composition.

Consider a transformation sequence $\mathcal{T}_1 \circ \cdots \circ \mathcal{T}_n$ as illustrated in Figure 5.15, where each I_k is the intermediate result output from $\mathcal{T}_{k-1}$ and input to $\mathcal{T}_k$. A correct but brute-force approach is to store all intermediate results $I_2, \ldots, I_n$ (in addition to initial input I) at loading time, then trace lineage backward through one transformation at a time. This approach is inefficient both due to the large number of tracing procedure calls when iterating through all transformations in the sequence, and due to the high storage cost for all

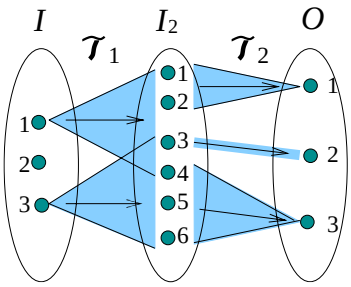


Figure 5.14: $\mathcal{T}_1 \circ \mathcal{T}_2$

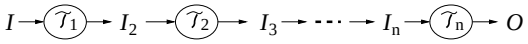


Figure 5.15: Transformation sequence

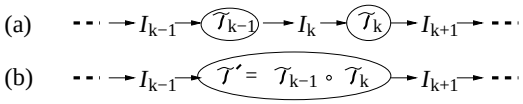


Figure 5.16: Combining transformations

intermediate results. The longer the sequence, the less efficient the overall tracing process. For realistic transformation sequences (in practice sometimes as many as 60 transformations) the cost can be prohibitive. Furthermore, if any transformation $\mathcal{T}$ in the sequence is a black-box, we will end up tracing the lineage of the entire input to $\mathcal{T}$ regardless of what transformations follow $\mathcal{T}$ in the sequence. Fortunately, it is often possible to mitigate these problems by *combining* adjacent transformations in a sequence for the purpose of lineage tracing. Also since we do not always need input sets for lineage tracing as discussed in Sections 5.3.1–5.3.3, some intermediate results can be discarded.

We will use the following overall strategy.

- When a transformation sequence $\mathcal{S} = \mathcal{T}_1 \circ \cdots \circ \mathcal{T}_n$ is defined, we first *normalize* the sequence, to be specified in Section 5.4.2, by combining transformations in $\mathcal{S}$ when it is beneficial to do so. We then determine which intermediate results need to be saved for lineage tracing, based on the best properties for the remaining transformations.

- When data is loaded through the transformation sequence, the necessary intermediate results are saved.

- We can then trace the lineage of any output data item o in the warehouse through the normalized transformation sequence using the iterative tracing procedure described at the beginning of this section.

5.4.2 Transformation Sequence Normalization

As discussed in Section 5.4.1, we want to combine transformations in a sequence for the purpose of lineage tracing when it is beneficial to do so. Specifically, we can *combine* transformations $\mathcal{T}_{k-1}$ and $\mathcal{T}_k$ as shown in Figure 5.16(a) by replacing the two transformations with the single transformation $\mathcal{T}' = \mathcal{T}_{k-1} \circ \mathcal{T}_k$ as shown in Figure 5.16(b), eliminating the intermediate result I_k and tracing through the combined transformation in one step.

To decide whether combining a pair of transformations is beneficial, and to use combined transformations for lineage tracing, as a first step we need to determine the properties of a combined transformation based on the properties of its component transformations. Let us associate with each transformation $\mathcal{T}_k$ all known schema information (input schema $\mathcal{T}_k.\mathbf{A}$, input key $\mathcal{T}_k.A_{key}$, output schema $\mathcal{T}_k.\mathbf{B}$, output key $\mathcal{T}_k.B_{key}$), all known schema mappings (forward mappings $\mathcal{T}_k$.*fmappings* and backward mappings $\mathcal{T}_k$.*bmappings*), whether $\mathcal{T}_k$ is complete ($\mathcal{T}_k$.*complete*), and a set $\mathcal{T}_k$.*properties* of all known properties $\mathcal{T}_k$ satisfies from the hierarchy in Figure 5.13. Procedure `Combine(`$\mathcal{T}_1, \mathcal{T}_2$`)` in Figure 5.17 sets these features for combined transformation $\mathcal{T} = \mathcal{T}_1 \circ \mathcal{T}_2$ based on the features for $\mathcal{T}_1$ and $\mathcal{T}_2$. Note from the algorithm that we need all of these features in order to properly determine $\mathcal{T}$.*properties* from $\mathcal{T}_1$ and $\mathcal{T}_2$. However, only the *properties* sets will be important in our final decision of whether to combine transformations.

Theoretically we can combine any adjacent transformations in a sequence. In fact we can collapse the entire sequence into one large transformation. However, combined transformations may have less desirable properties than their component transformations, leading to less efficient or less accurate lineage tracing. Thus, we want to combine transformations only if it is beneficial to do so. Given a transformation sequence, determining the best way to combine transformations in the sequence is a difficult combinatorial problem—solving it accurately, or even just determining accurately when it is beneficial to combine two transformations, would require a detailed cost model that takes into account transformation properties, the cost of applying a transformation, the cost of storing intermediate results, and an estimated workload (including, e.g., data size and tracing frequency). Developing such a cost model is beyond the scope of this thesis.

procedure Combine($\mathcal{T}_1, \mathcal{T}_2$)
  $\mathcal{T} \leftarrow \mathcal{T}_1 \circ \mathcal{T}_2$;
  $\mathcal{T}.\mathbf{A} \leftarrow \mathcal{T}_1.\mathbf{A}$; $\mathcal{T}.A_{key} \leftarrow \mathcal{T}_1.A_{key}$;
  $\mathcal{T}.\mathbf{B} \leftarrow \mathcal{T}_2.\mathbf{B}$; $\mathcal{T}.B_{key} \leftarrow \mathcal{T}_2.B_{key}$;
  $\mathcal{T}$.*fmappings* $\leftarrow \varnothing$; $\mathcal{T}$.*bmappings* $\leftarrow \varnothing$; $\mathcal{T}$.*properties* $\leftarrow \varnothing$;
  $\mathcal{T}$.*complete* $\leftarrow \mathcal{T}_1$.*complete* **and** $\mathcal{T}_2$.*complete*;
  for each property p in {*aggregator, dispatcher, filter, tracing-proc w/o input*} **do**
    if $p \in \mathcal{T}_1$.*properties* **and** $p \in \mathcal{T}_2$.*properties* **then** add p to $\mathcal{T}$.*properties*;
  for each $f_1(A) \rightarrow A'$ in $\mathcal{T}_1$.*fmappings* **do**
    if $\exists\ f_2(A') \rightarrow B$ in $\mathcal{T}_2$.*fmappings* **then** add $f_1 \circ f_2(A) \rightarrow B$ to $\mathcal{T}$.*fmappings*;
  for each $A \leftarrow g_1(A')$ in $\mathcal{T}_1$.*bmappings* **do**
    if $\exists\ A' \leftarrow g_2(B)$ in $\mathcal{T}_2$.*bmappings* **then** add $A \leftarrow g_2 \circ g_1(B)$ to $\mathcal{T}$.*bmappings*;
  if $\exists\ f(A) \rightarrow \mathcal{T}.B_{key}$ in $\mathcal{T}$.*fmappings* **and** $\mathcal{T}$.*complete* **then** add *fkmap* to $\mathcal{T}$.*properties*;
  if $\exists\ \mathcal{T}.A_{key} \leftarrow g(B)$ in $\mathcal{T}$.*bmappings* **then** add *bkmap* to $\mathcal{T}$.*properties*;
  if $\exists\ \mathcal{T}.\mathbf{A} \leftarrow g(B)$ in $\mathcal{T}$.*bmappings* **then** add *btmap* to $\mathcal{T}$.*properties*;
return $\mathcal{T}$;

Figure 5.17: Combining Transformation Pairs

Instead, we suggest a greedy algorithm `Normalize` shown in Figure 5.18. The algorithm repeatedly finds beneficial combinations of transformation pairs in the sequence, combines the "best" pair, and continues until no more beneficial combinations are found. In general, a combination should be considered beneficial only if it reduces the overall tracing cost while improving or retaining tracing accuracy. We determine whether it is beneficial to combine two transformations based solely on their properties using the following two heuristics. First, we do not combine transformations into black-boxes, unless we are certain that the combination will not degrade the accuracy of the lineage result, which can only be determined as a last step of the `Normalize` procedure. Second, we do not combine transformations if their composition is significantly worse for lineage tracing, i.e., it has much higher tracing cost or leads to a less accurate result. We divide the properties in Figure 5.13 into five groups: group 1 contains properties 1–3, group 2 contains properties 4–8, group 3 contains property 9, group 4 contains property 10, and group 5 contains property 11. Within each group, the efficiency and accuracy of the tracing procedures are fairly similar, while they differ significantly across groups. The group of a transformation $\mathcal{T}$, denoted $group(\mathcal{T})$, is the group that $\mathcal{T}$'s best property belongs to. The lower the group number, the better $\mathcal{T}$ is for lineage tracing, and we consider it beneficial to combine two

```
procedure Normalize(S = T_1 ∘ ··· ∘ T_n)
  while (k ← BestCombo(S)) ≠ 0
    replace T_k and T_{k+1} in S with T ← Combine(T_k, T_{k+1});
  if S contains black-box transformations then
    j ← lowest index of a black-box in S;
    replace transformations T_j, ..., T_n in S with T ← T_j ∘ ··· ∘ T_n;
```

```
procedure BestCombo(S = T_1 ∘ ··· ∘ T_n)
  k ← 0; N[1..5] ← [0,0,0,0,0];
  for j = 1..n do g ← group(T_j); N[g] ← N[g] + 1; // initialize vector
  for j = 1..n − 1 do
    curN ← N;
    T ← Combine(T_j, T_{j+1});
    g ← group(T);
    if g < 5 then // T is not a black-box
      g_1 ← group(T_j); g_2 ← group(T_{j+1});
      curN[g] ← curN[g] + 1; curN[g_1] ← curN[g_1] − 1; curN[g_2] ← curN[g_2] − 1;
      if curN < N then k ← j; N ← curN; // see text for defn. of curN < N
return k;
```

Figure 5.18: Normalizing a transformation sequence

transformations $\mathcal{T}_1$ and $\mathcal{T}_2$ only if $group(\mathcal{T}_1 \circ \mathcal{T}_2) \leq max(group(\mathcal{T}_1), group(\mathcal{T}_2))$.[5]

Based on the above approach, procedure `BestCombo` in Figure 5.18, called by `Normalize`, finds the best pair of adjacent transformations to combine in sequence $\mathcal{S}$, and returns its index. The procedure returns 0 if no combination is beneficial. We consider a beneficial combination to be the best if the combination leaves the fewest "bad" transformations in the sequence, compared with other candidates. Formally, we associate with $\mathcal{S}$ a vector $N[1..5]$, where $N[j]$ is the number of transformations in $\mathcal{S}$ that belong to group j. (So $\sum_{j=1..5} N[j]$ equals the length of $\mathcal{S}$.) Given two sequences $\mathcal{S}_1$ and $\mathcal{S}_2$ with vectors N_1 and N_2 respectively, let k be the highest index in which $N_1[k]$ differs from $N_2[k]$. We say $N_1 < N_2$ if $N_1[k] < N_2[k]$, which implies that $\mathcal{S}_1$ has fewer "bad" transformations than $\mathcal{S}_2$. Then we say that the best combination is the one that leads to the lowest vector N for

[5]Note that the presence of non-key-map schema mappings (Section 5.3.2) or indexes (Section 5.3.6) for a transformation $\mathcal{T}$ does not improve $\mathcal{T}$'s tracing efficiency to the point of moving it to a different group. Thus, we do not take these factors into account in our decision process.

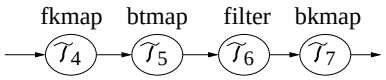


Figure 5.19: Before normalization

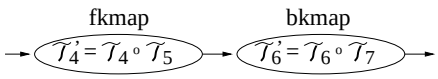


Figure 5.20: After normalization

the resulting sequence.

After we finish combining transformations as described above, suppose the sequence still contains one or more black-box transformations. During lineage tracing, we will end up tracing the lineage of the entire input to the earliest (left-most) black-box $\mathcal{T}$ in $\mathcal{S}$, regardless of what transformations follow $\mathcal{T}$. Therefore, as a final step we combine $\mathcal{T}$ with all transformations that follow $\mathcal{T}$ to eliminate unnecessary tracing and storage costs.

Our `Normalize` procedure has complexity $O(n^2)$ for a transformation sequence of length n. Although we use a greedy algorithm and heuristics for estimating the benefit of combining transformations, our approach is quite effective in improving tracing performance for sequences, as we will see in Section 5.7.

Example 5.4.1 Consider the sequence of transformations $\mathcal{S} = \mathcal{T}_4 \circ \mathcal{T}_5 \circ \mathcal{T}_6 \circ \mathcal{T}_7$ from Section 5.1. Figure 5.19 shows the sequence and the best property of each transformation. The initial vector of $\mathcal{S}$ is $N = [2, 2, 0, 0, 0]$. Using our greedy normalization algorithm, we first consider combining $\mathcal{T}_4 \circ \mathcal{T}_5$ into $\mathcal{T}_4'$ with best property *fkmap*, combining $\mathcal{T}_5 \circ \mathcal{T}_6$ into $\mathcal{T}_5'$ with best property *btmap*, or combining $\mathcal{T}_6 \circ \mathcal{T}_7$ into $\mathcal{T}_6'$ with best property *bkmap*. It turns out that all these combinations reduce $\mathcal{S}$'s vector N to $[1, 2, 0, 0, 0]$. So let us combine $\mathcal{T}_4 \circ \mathcal{T}_5$ obtaining $\mathcal{T}_4'$, $\mathcal{T}_6$, and $\mathcal{T}_7$. In the new sequence, combining $\mathcal{T}_4' \circ \mathcal{T}_6$ results in a black-box, which is disallowed, while combining $\mathcal{T}_6 \circ \mathcal{T}_7$ results in a transformation $\mathcal{T}_6'$ with best property *bkmap*, which reduces N to $[0, 2, 0, 0, 0]$. Therefore, we choose to combine $\mathcal{T}_6 \circ \mathcal{T}_7$ obtaining $\mathcal{T}_4'$ and $\mathcal{T}_6'$. Combining these two transformations would result in a black-box, so we stop at this point. The final normalized sequence is shown in Figure 5.20. □

5.5 Transformations with Multiple Input and Output Sets

So far we have addressed the lineage tracing problem for transformations with a single input set and single output set (Section 5.3), and for linear sequences of such transformations (Section 5.4). In this section, we extend our approach to handle individual transformations

with multiple input and/or multiple output sets. In Section 5.6, we put everything together to tackle the lineage tracing problem for arbitrary acyclic transformation graphs.

5.5.1 Multiple-Input Single-Output Transformations

We first consider transformations with multiple input sets but only one output set, which we call *Multiple-Input Single-Output* (*MISO*) transformations. What is most relevant about a transformation $\mathcal{T}$ with multiple input sets is how exactly $\mathcal{T}$ combines its input sets to produce its output. After studying MISO transformations in practice we determined that although it is possible to define a large number of narrow MISO transformation classes, it makes more sense to define just two broad classes—*exclusive* and *inclusive* MISO transformations. These classes still enable efficient and accurate lineage tracing in almost all cases. Furthermore, as will be seen, this approach to MISO transformations exploits our entire framework for single-input transformations, simply adding to it the mechanics for handling transformations that operate on multiple inputs.

Exclusive MISO Transformations

Consider a MISO transformation $\mathcal{T}$ with m input sets $I_1, \ldots, I_m$ and one output set O. Informally, $\mathcal{T}$ is an *exclusive* transformation if it effectively transforms each input set independently and produces as output the union of the results. More formally, $\forall I_1, \ldots, I_m$, $\mathcal{T}(I_1, \ldots, I_m) = \bigcup_{j=1..m} \mathcal{T}(\varnothing, \ldots, I_j, \ldots, \varnothing)$. Note that since we are considering set union, it is still possible for an output item $o \in O$ to be produced by more than one input set.

By the above definition of an exclusive MISO transformation, we can "split" $\mathcal{T}$ into j independent transformations, one for each input. Specifically, for $j = 1..m$ we define a single-input *split transformation* $\mathcal{T}[j](I) = \mathcal{T}(\varnothing, \ldots, \varnothing_{j-1}, I, \varnothing_{j+1}, \ldots, \varnothing)$. Now let $O[j] = \mathcal{T}[j](I_j)$ for $j = 1..m$; that is, $O[j]$ is the portion of the output produced from the jth input. We define the lineage of an output item $o \in O$ according to $\mathcal{T}$ to be the lineage of o according to all split transformations $\mathcal{T}[j]$ where $o \in O[j]$.

Based on these definitions, let us now consider how we enable and perform lineage tracing. Since an exclusive MISO transformation effectively transforms each input independently, we assume that the transformation author can understand and specify the properties

of each split transformation $\mathcal{T}[j]$, just as he would do for a single-input transformation. Any of the properties from our hierarchy in Figure 5.13 can be specified. Thus, to trace the lineage of an output item $o \in O$, we first determine all j's such that $o \in O[j]$ (we may want to compute and store the $O[j]$'s at load time for this purpose). Then we trace o through each relevant $\mathcal{T}[j]$ to input I_j based on $\mathcal{T}[j]$'s specified properties, using the algorithms of Section 5.3.

The most obvious example of an exclusive MISO transformation is set union, for which each split transformation is a filter.

Inclusive MISO Transformations

The other class of MISO transformations—all MISO transformations that are not exclusive—is the *inclusive* MISO transformations. Informally, MISO transformations are inclusive when all of their input sets need to be combined together in some fashion to produce the output set. (A MISO transformation could treat some inputs exclusively and others inclusively, but such transformations are rare in practice and we treat them as purely inclusive.)

For inclusive transformations we also define the notion of *split transformations* to enable the definition of lineage and the tracing process. However, the definition of split transformation differs from that for exclusive MISO transformations, in that it must be based on each transformation instance. For an inclusive MISO transformation instance $\mathcal{T}(I_1, \ldots, I_m) = O$, the split transformation $\mathcal{T}[j]$ is defined as $\mathcal{T}[j](I) = \mathcal{T}(I_1, \ldots, I_{j-1}, I, I_j, \ldots, I_m)$. That is, $\mathcal{T}[j]$ takes as a parameter the jth input set and treats the other input sets as constants. Note that $\mathcal{T}[j](I_j) = O$ for all $j = 1..m$, so we do not need the concept of $O[j]$. Otherwise, we proceed in the same way as for the exclusive case: The transformation author specifies properties for each split transformation. To trace the lineage of an output item $o \in O$, we trace its lineage through each split transformation $\mathcal{T}[j]$.

It is easy to see intuitively why standard relational operators such as join, intersection, and difference are inclusive: they need all of their inputs in order to produce their output. As a concrete example, consider transformation $\mathcal{T}_3$ in Section 5.1, which joins the order information in input I_1 and the product information in input I_2. $\mathcal{T}_3$'s split transformation $\mathcal{T}_3[1](I_1)$ effectively takes each order $i \in I_1$, finds (in I_2) the corresponding product information, and attaches it to i to produce an output item. $\mathcal{T}_3[1]$ has a backward schema

mapping `order-id` $\overset{\mathcal{T}_3[1]}{\leftarrow}$ `order-id` to the input key, so it is a backward key-map. Similarly, $\mathcal{T}_3[2]$ is a backward key-map. Thus, to trace an output item o through $\mathcal{T}_3$, we find the corresponding order tuple in input I_1 through $\mathcal{T}_3[1]$ using `TraceBM`, and the corresponding product tuple in input I_2 through $\mathcal{T}_3[2]$ also using `TraceBM`.

5.5.2 Multiple-Input Multiple-Output Transformations

Consider a multiple-input multiple-output transformation $\mathcal{T}$ that takes m input sets $I_1, \ldots, I_m$ and produces n output sets $O_1, \ldots, O_n$. To trace the lineage of an output item $o \in O_k$, we consider a *restriction* $\mathcal{T}^k$ of $\mathcal{T}$ on output O_k such that $\mathcal{T}^k(I_1, \ldots, I_m) = O_k$. That is, $\mathcal{T}^k$ acts like $\mathcal{T}$ but produces only O_k, ignoring the other output sets. We define the lineage of o according to $\mathcal{T}$ as o's lineage according to $\mathcal{T}^k$. Since $\mathcal{T}^k$ is a MISO transformation, we proceed as in Section 5.5.1. Note that the transformation author must understand the restricted transformations $\mathcal{T}^k$, but they are usually very straightforward and frequently symmetric.

5.6 Lineage Tracing Through Transformation Graphs

Finally we consider the most general case: lineage tracing for arbitrary acyclic graphs of transformations. Consider a transformation graph $\mathcal{G}$ with m initial inputs $I_1, \ldots, I_m$ and n outputs $O_1, \ldots, O_n$. Each transformation in $\mathcal{G}$ can have any number of inputs and outputs, and we know from Section 5.5.2 how to trace through any such transformation. Thus, to trace the lineage of an output item $o \in O_k$, we can trace o through the entire graph backwards, similar to our approach for sequences but possibly needing to follow multiple backward paths through the graph. To enable tracing in this fashion, at loading time we may need to store all intermediate results for each edge of the graph, add indexes as appropriate, and store split transformation outputs for exclusive multiple-input transformations as described in Section 5.5.1.

It is not immediately obvious how to fully generalize our idea of improving lineage tracing performance by combining transformations (Section 5.4) to the graph case, and doing so in depth is beyond the scope of this thesis. However, because our approach to lineage tracing for multiple-input transformations is based on the notion of single-input split

transformations (Sections 5.5.1 and 5.5.1), often we can improve overall performance of lineage tracing through a transformation graph by applying techniques developed earlier in this chapter—we normalize and trace lineage through each path in the graph independently:

1. When $\mathcal{G}$ is defined, we create a *tracing sequence* $\mathcal{S}$ for each path in $\mathcal{G}$ from an initial input set to a final output set. If a transformation $\mathcal{T}$ on the path has multiple inputs and/or multiple outputs, then in sequence $\mathcal{S}$ we replace $\mathcal{T}$ with its restricted (Section 5.5.2) and split (Section 5.5.1) transformation $\mathcal{T}^k[j]$, where the previous transformation in $\mathcal{S}$ provides $\mathcal{T}$'s jth input in $\mathcal{G}$, and the next transformation in $\mathcal{S}$ takes $\mathcal{T}$'s kth output in $\mathcal{G}$. Given our overall approach, the transformation author will already have specified properties for all $\mathcal{T}^k[j]$'s. We then normalize each sequence $\mathcal{S}$ as described in Section 5.4.2. When finished, we have q normalized tracing sequences $\mathcal{S}_1, \ldots, \mathcal{S}_q$ for q paths in $\mathcal{G}$.

2. When the warehouse data is loaded, intermediate results required for tracing through each normalized tracing sequence $\mathcal{S}_1, \ldots, \mathcal{S}_q$ are saved (sharing is sometimes possible), and indexes are built as desired.

3. When tracing the lineage of output item $o \in O_k$, we simply trace o through each normalized tracing sequence $\mathcal{S}$ that ends in O_k, and combine the results.

Example 5.6.1 (Lineage Tracing for `SalesJump`) Recall the warehouse table `SalesJump` from Section 5.1, created by transformation graph $\mathcal{G}$ in Figure 5.3. From $\mathcal{G}$, we create two tracing sequences as shown in Figure 5.21(a), where $\mathcal{T}_3[1]$ and $\mathcal{T}_3[2]$ are split transformations of $\mathcal{T}_3$. The figure also shows the best property for each transformation in the sequences. We then use our greedy algorithm `Normalize` to combine transformations in the two tracing sequences as described in Section 5.4.2. We obtain the two normalized tracing sequences shown in Figure 5.21(b). At loading time, to enable lineage tracing through our two normalized tracing sequences, we only need to save the initial inputs and the intermediate results from transformations $\mathcal{T}_3$ and $\mathcal{T}_5$ in the original graph $\mathcal{G}$. At tracing time, consider as an example output data item $o = \langle$`Sony VAIO`, `11250`, `39600`$\rangle$ in `SalesJump`, which was also used as a motivating example in Section 5.1. We first trace o's lineage in `Order` through

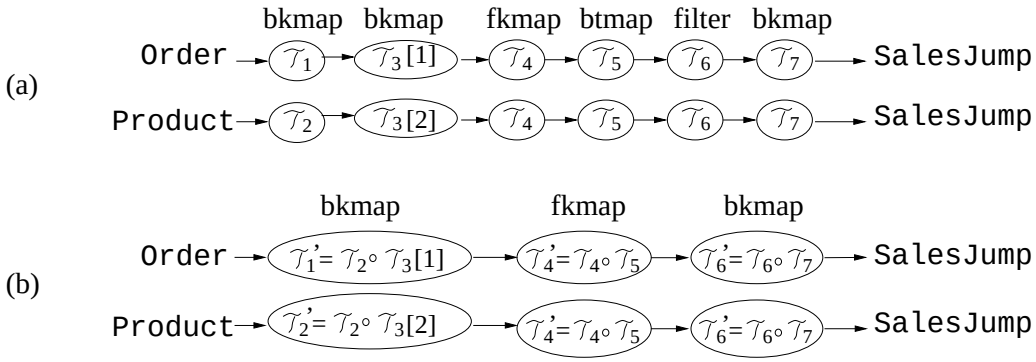


Figure 5.21: Tracing sequences (unnormalized and normalized) for `SalesJump`

order-id	date	prod-id	quantity	prod-name	price	valid
0101	2/1/1999	222	10	Sony VAIO	2250	12/1/1998–9/30/1999
0379	4/9/1999	222	5	Sony VAIO	2250	12/1/1998–9/30/1999
1028	11/24/1999	222	10	Sony VAIO	1980	10/1/1999–
1250	12/15/1999	222	10	Sony VAIO	1980	10/1/1999–

Figure 5.22: I_4^*

the upper normalized tracing sequence in Figure 5.21(b). We trace o through transformation $\mathcal{T}_6'$ using `TraceBM` to obtain $I_6^* = \{\langle$`Sony VAIO, 22500, 11250, 0, 11250, 39600`$\rangle\}$. We then trace I_6^* through $\mathcal{T}_4'$ using `TraceFM` to obtain I_4^* as shown in Figure 5.22. Finally, we trace I_4^* through $\mathcal{T}_1'$ using `TraceBM` to obtain the `Order` tuples shown in Figure 5.5. We also trace o's lineage in `Product` through the lower normalized tracing sequence in Figure 5.21(b). The final result is the `Product` tuples shown in Figure 5.5. □

5.7 Performance Experiments

We have implemented all of the algorithms described in this chapter in a prototype data lineage tracing system for general warehouse transformations. For convenience we decided to use a standard commercial relational DBMS for storing all data (input sets, output sets, and intermediate results), and we implemented our lineage tracing procedures as well as our test suite of data transformations as parameterized stored procedures.

In this section we provide some performance results. Specifically, we consider the following three questions for a few representative cases:

1. Roughly how fast are the lineage tracing procedures?
2. How much speedup can we obtain using indexes?
3. How much faster can we trace through transformation sequences when we combine transformations as in Section 5.4?

The data we used for our experiments is based on the TPC-D benchmark [TPC96]. Specifically, our input tables are `LineItem`, `Order`, and `PartSupp` from the benchmark. Contents of the tables were generated by the standard `dbgen` program supplied with the benchmark. Note that a TPC-D scale factor of 1.0 means that the entire warehouse is about 1GB in size.

5.7.1 Tracing Performance

In the first experiment, we consider a simple relational transformation $\mathcal{T}$ which could be defined in SQL as follows:

```
SELECT PartKey, SuppKey, AvailQty*SupplyCost FROM PartSupp;
```

This transformation is a dispatcher, and also is a backward key-map with a schema mapping from the output key attributes `PartKey` and `SuppKey` to the same input attributes, so we can trace its data lineage using either `TraceDS` or `TraceBM`. We vary the input table scale factor from 0.08 to 1, and we measure lineage tracing time for a single tuple using `TraceDS` and `TraceBM` with and without the indexes described in Section 5.3.6. From the results shown in Figure 5.23, we see that when indexes are not used procedure `TraceBM` is significantly faster than `TraceDS`. (Note the log scale on the y axis.) With indexes both tracing procedures are very fast.

5.7.2 Combining Versus Not Combining

Our second experiment studies the benefit of combining transformations using the techniques introduced in Section 5.4. We consider the transformation graph $\mathcal{G}$ in Figure 5.25, which is based on query Q12 from the TPC-D benchmark. The graph takes as input tables `LineItem` and `Order`, and produces as output the number of high and low priority orders on a selected set of line items for each shipment mode. (Recall that α represents grouping

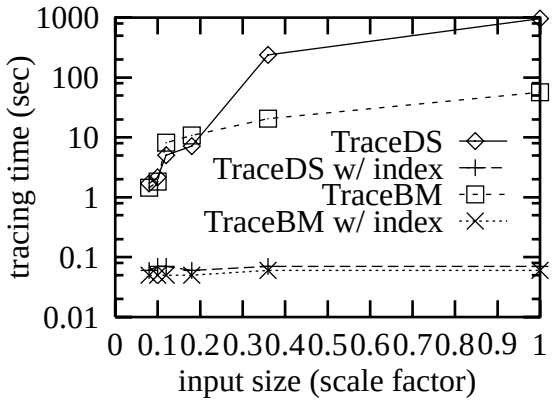


Figure 5.23: Tracing one transformation

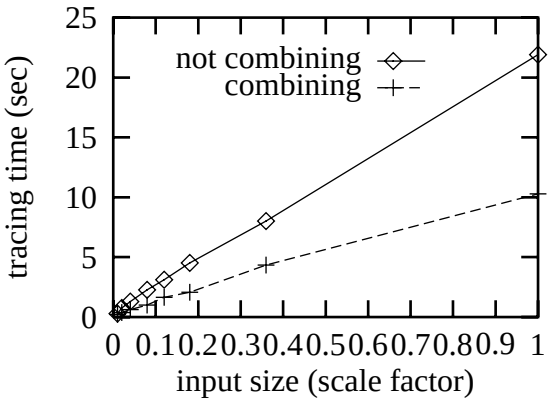


Figure 5.24: Combining vs. not combining

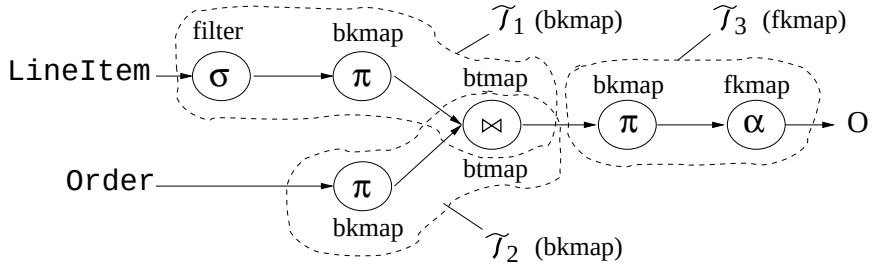


Figure 5.25: Transformation graph for Q12 in TPC-D

and aggregation.) Given this transformation graph, in one experiment we trace lineage one transformation at a time (for two paths), based on the individual transformation properties marked in Figure 5.25. In the other experiment, we first normalize the two sequences, which results in the combined transformations $\mathcal{T}_1$, $\mathcal{T}_2$, and $\mathcal{T}_3$ shown by the dashed regions in Figure 5.25. We then trace lineage through the combined transformations on both paths. We vary the input scale factor from 0.01 to 1 and obtain the results shown in Figure 5.24. We see that as the input size grows, combining transformations provides a significant reduction in tracing time.

5.8 Related Work

There has been a significant body of work on data transformations in general, including aspects such as transforming data formats, models, and schemas, e.g., [ACM+99, BDH+95,

CR99, HMN+99, LSS96, RH01, Shu87, Squ95]. Often the focus is on data integration or warehousing, but none of these papers considers lineage tracing through transformations, or even addresses the related problem of transformation inverses. A recent paper on interactive data transformation and data cleaning [GFS+01] discusses how to use data lineage to develop effective data cleaning tools that allow navigation back and forth in a transformation graph.

As discussed briefly in Section 1.5, there has also been work on tracing schema-level lineage for data transformations providing information such as which transformations were involved in producing a given warehouse data item [BB99, LBM98], or which source attributes derive certain warehouse attributes [YMHR01, HQGW93, MBR01, RS98]. As seen earlier in this chapter, we can sometimes use such information (e.g., *schema mappings*) in support of our instance-level lineage tracing techniques.

Also as discussed briefly in Section 1.5, [WS97] proposed a general framework for computing fine-grained data lineage in a transformational setting. The paper defines and traces data lineage for each transformation based on a *weak inverse*, which must be specified by the transformation definer. Lineage tracing through a transformation graph proceeds by tracing through each path one transformation at a time. In our approach, the definition and tracing of data lineage is based on general transformation properties, and we specify an algorithm for combining transformations in a sequence or graph for improved tracing performance.

Finally, [LGMW00] considers an ETL setting like ours, and defines the concept of a *contributor* in order to enable efficient resumption of interrupted warehouse loads. Although similar in overall spirit, the definition of a contributor in [LGMW00] is different from our definition of data lineage, and does not capture all aspects of data lineage we consider in this chapter. In addition, we consider a more general class of transformations than those considered in [LGMW00].

5.9 Chapter Summary

We presented a complete set of techniques for lineage tracing when the warehouse data is loaded through a graph of general transformations. Our approach relies on a variety of

transformation properties that hold frequently in practice, and that can be specified easily by transformation authors. We also introduced some optimization techniques, including building indexes and combining transformations for the purpose of lineage tracing. All algorithms presented in this chapter have been implemented in a prototype lineage tracing system and performance results are reported.

The results in this chapter can be used to develop principles for creating transformations and transformation graphs that are amenable to lineage tracing. As a first step, transformation authors can be sure to specify the most restrictive properties that a transformation satisfies based on our property hierarchy (Figure 5.13), to ensure that the most efficient tracing procedure is selected. Second, a transformation might be modified slightly to improve its properties, for example retaining key values so that a dispatcher becomes a backward key-map. Third, in many cases complex black-box transformations can be avoided by splitting the transformation into simpler transformations with better properties for more refined or efficient lineage tracing. In general, for the purposes of lineage tracing it is better to specify smaller atomic transformations rather than larger ones, since the lineage tracing system will combine transformations automatically anyway when it is beneficial to do so.

Chapter 6

Conclusions and Future Work

This thesis addressed the general problem of lineage tracing in a data warehousing environment. We considered two data warehousing scenarios: warehouses defined by relational views and warehouses constructed through graphs of general data transformations. For each scenario, we provided a formal lineage definition, lineage tracing algorithms, and optimization techniques. Unlike most previous work on data lineage, which requires the warehousing system to annotate warehouse data items with lineage information when they are computed, our algorithms perform lineage tracing in a lazy fashion: we use automatically-generated tracing procedures to trace data lineage only when a lineage query is issued. Based on our results, we have implemented a lineage tracing system prototype for relational views within the WHIPS [HGMW+95] data warehousing project at Stanford, and we have implemented our algorithms for lineage tracing in the presence of general transformations. A variety of empirical results were reported in the thesis.

In Chapter 2, we formulated the data lineage problem and provided lineage tracing algorithms for a very general class of relational views. Ours was the first work we know of to provide a declarative definition of data lineage and detailed tracing algorithms. Although the concept of data lineage seems straightforward in many of our examples, finding a declarative and intuitive definition for data lineage turned out to be one of the biggest challenges in our work.

In Chapter 3, we addressed performance and implementation issues. We began by considering select-project-join (SPJ) views. We introduced a family of schemes for storing

auxiliary views at the warehouse to provide consistent and efficient lineage tracing in a distributed warehousing environment. In general, storing auxiliary views will incur additional warehouse loading and maintenance cost. However, some auxiliary views that are useful for lineage tracing can also support maintenance of the original view. To study the performance tradeoffs among lineage tracing and warehouse maintenance, we ran simulations on our auxiliary view schemes. The results showed that auxiliary views can improve performance dramatically, while different auxiliary view schemes offer different advantages and thus are suitable for different settings. We then considered arbitrary aggregate-select-project-join (ASPJ) views. Here we faced a combinatorial optimization problem, and we proposed four algorithms: *exhaustive*, *greedy*, *three-step*, and *naive*. Our experiments showed that although the greedy and three-step algorithms do not guarantee to always find an optimal auxiliary view set, they do find a near-optimal auxiliary view set quickly in most cases. We saw that even a naive selection of auxiliary views can again reduce overall cost dramatically. In general, our performance experiments underscored the importance of materializing auxiliary views for the dual purposes of lineage tracing and view maintenance in a warehousing environment. Chapter 3 also described the lineage tracing prototype we implemented in the context of the WHIPS data warehousing system at Stanford.

In Chapter 4, we studied the relationship between data lineage and the view update problem for deletions. We devised a fully automatic algorithm for translating deletions against SPJ views into deletions against the underlying database using techniques based on data lineage. Unlike most previous work on view update, our algorithm does not require user intervention or additional semantic information. Instead, it uses only the view definition at compile time and the base data at view-update time to find a translation that is guaranteed to be exact (side-effect free) whenever an exact translation exists.

In Chapter 5, we extended our lineage definition and provided new lineage tracing algorithms for data warehouses created through graphs of general transformations. Our approach relied on a variety of transformation properties that hold frequently in practice, and that can be specified easily by transformation authors. In cases where the transformations are relational operators, the new lineage definition is the same as the lineage definition for relational views from Chapter 2. We also presented optimization techniques such as building indexes and combining transformations for the purpose of lineage tracing. Our

results can guide the design of transformation-based data warehouses that enable efficient and accurate lineage tracing.

6.1 Future Work

We propose several possible avenues of future work related to the results in this thesis.

6.1.1 Annotation-Based Lineage Tracing

Throughout the thesis we have assumed that most of the work for lineage tracing should be done at tracing time. That is, we don't want to expend considerable extra computation or storage cost during the warehouse load or refresh process just for lineage tracing. An alternative approach is to *annotate* each warehouse data item with its lineage information at warehouse load time. Then, at tracing time we can obtain the lineage of any given warehouse data item directly from its precomputed annotation. There are a number of ways to compute and store lineage annotations, some proposed in previous work (e.g., [HQGW93]), and others that could exploit our lineage tracing algorithms.

The lineage annotation approach improves lineage tracing performance at the expense of warehouse load and refresh performance, and is preferred only when lineage queries are frequent and tracing performance is crucial. The "lazy" tracing approach used throughout this thesis and the lineage annotation approach actually delineate two extremes in terms of how much work should be done at warehouse load time versus lineage tracing time. It might be interesting to explore middle-ground approaches, which compute and store some amount of annotation information to support lineage tracing, but without incurring undue performance degradation at load time.

6.1.2 Missing Data

Data lineage as defined in this thesis explains how certain base data causes certain warehouse data to exist. As such, lineage tracing is a useful technique for investigating the origins of potentially erroneous warehouse data. However, in some cases a warehouse data item may be erroneous not (only) because some existing source data items that derive it

are erroneous, but because source data items that should appear in the lineage are missing. For example, a source tuple may contribute to the wrong group in a relational aggregate view because its grouping value is incorrect. It is also possible that some data items that should appear in the warehouse are missing because of erroneous (or missing) source data. One area of future work is to explore how both of these "missing data" problems can be addressed in our lineage framework.

6.1.3 Generalizing View Update

In Chapter 4, we considered the view update problem for select-project-join (SPJ) views. More general relational views (including, e.g., aggregation, set union, or difference) pose additional challenges to the view update problem. Our work on relational data lineage does cover a very general class of relational views. We suspect that, as we have shown with SPJ views, our lineage work on more general views (such as views with aggregation) can be used as the basis of new view update translation algorithms. Going one step further, we also addressed the data lineage problem for general transformation graphs, which appears to pose an even more challenging view update problem.

We have addressed the deletion-to-deletion view update translation problem only. Even extending to consider base insertions or modifications when translating view deletions (the "compensation" idea discussed in Section 4.1) is a significant step, particularly if we wish to retain our exactness guarantee. We might also consider extending our algorithm to provide translations for view insertions or modifications, although as discussed in Section 4.1 it appears that data lineage techniques may not be applicable in these cases.

Our algorithms in Chapter 4 did not attempt to produce any kind of *minimal* translation, e.g., one that deletes the fewest base tuples. Using cardinality as a minimality metric, modifying our algorithm is not difficult, but keeping it efficient is a challenge. We also did not take constraints on base data (or on base data updates) into account—again the challenge may be one primarily of efficiency. Finally, our view update solution considers each view in isolation. When there are multiple views, an update translation could induce side-effects on views other than the view being updated, and it might be interesting to take such "side-side-effects" into account.

6.1.4 Generalizing Transformation-Based Lineage Tracing

In Chapter 5, we assumed that properties of a transformation are provided to the lineage tracing system, either by the transformation author, or because the transformation is a prepackaged component with known properties. A separate line of research is that of inferring a transformation's properties, either by examining the specification (e.g., using program analysis techniques over the code), or by running sample data through the transformation and examining the results [YMHR01, MBR01].

As discussed in Section 5.4.2, we used a simple cost metric and greedy algorithm for normalizing transformation sequences. Although we already obtain good performance improvements (Section 5.7), clearly it would be interesting to develop a more detailed and accurate cost model and more sophisticated algorithms for exploring the space of transformation combinations. More generally, in the case of transformation graphs, we might benefit from normalizing the graph globally, rather than normalizing each path independently. Even more generally, in the presence of commutative transformations (which are not common but do occur), we might consider reorganizing a transformation sequence or graph to obtain a better normalized result.

Bibliography

[ACM+99] S. Abiteboul, S. Cluet, T. Milo, P. Mogilevsky, J. Simeon, and S. Zohar. Tools for data translation and integration. *IEEE Data Engineering Bulletin*, 22(1):3–8, March 1999.

[BB99] P. Bernstein and T. Bergstraesser. Meta-data support for data transformations using Microsoft Repository. *IEEE Data Engineering Bulletin*, 22(1):9–14, March 1999.

[BDH+95] P. Buneman, S.B. Davidson, K. Hart, G.C. Overton, and L. Wong. A data transformation system for biological data sources. In *Proc. of the Twenty-first International Conference on Very Large Data Bases*, pages 158–169, Zurich, Switzerland, September 1995.

[BS81] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Transaction on Database Systems*, 6(4):557–575, 1981.

[CD97] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Record*, 26(1):65–74, March 1997.

[Cle78] E.K. Clemons. *An external schema facility to support data base updates*. Academic Press, 1978.

[CR99] K.T. Claypool and E.A. Rundensteiner. Flexible database transformations: The SERF approach. *IEEE Data Engineering Bulletin*, 22(1):19–24, March 1999.

[CW00a] Y. Cui and J. Widom. Lineage tracing in a data warehousing system. In *Proc. of the Sixteenth International Conference on Data Engineering*, pages 683–684, San Diego, California, February 2000. Demonstration description.

[CW00b] Y. Cui and J. Widom. Practical lineage tracing in data warehouses. In *Proc. of the Sixteenth International Conference on Data Engineering*, pages 367–378, San Diego, California, February 2000.

[CW00c] Y. Cui and J. Widom. Storing auxiliary data for efficient maintenance and lineage tracing of complex views. In *Proc. of the 2nd International Workshop on Design and Management of Data Warehouses*, Stockholm, Sweden, June 2000.

[CW01] Y. Cui and J. Widom. Run-time translation of view tuple deletions using data lineage. Technical report, Stanford University Database Group, June 2001. Available at http://dbpubs.stanford.edu/pub/2001-24.

[CWW00] Y. Cui, J. Widom, and J.L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Transactions on Database Systems*, 25(2):179–227, 2000.

[DB78] U. Dayal and P.A. Bernstein. On the updatability of relational views. In *Proc. of the Fourth International Conference on Very Large Data Bases*, pages 368–377, Germany, September 1978.

[DB82] U. Dayal and P.A. Bernstein. On the correct translation of update operations on relational views. *ACM Transaction on Database Systems*, 8(3):381–416, 1982.

[DB2] IBM Corporation: DB2 OLAP Server. http://www.ibm.com/db2/.

[FJS97] C. Faloutsos, H.V. Jagadish, and N.D. Sidiropoulos. Recovering information from summary data. In *Proc. of the Twenty-Third International Conference on Very Large Data Bases*, pages 36–45, Athens, Greece, August 1997.

[GBLP96] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In *Proc. of the Twelfth International Conference on Data Engineering*, pages 152–159, New Orleans, Louisiana, February 1996.

[GFS+01] H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C. Saita. Improving data cleaning quality using a data lineage facility. In *Proc. of the Third International Workshop on Design and Management of Data Warehouses*, Interlaken, Switzerland, June 2001.

[GHQ95] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. In *Proc. of the Twenty-First International Conference on Very Large Data Bases*, pages 358–369, Zurich, Switzerland, September 1995.

[GMR95] A. Gupta, I.S. Mumick, and K.A. Ross. Adapting materialized views after redefinitions. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 211–222, San Jose, California, May 1995.

[GMS93] A. Gupta, I. S. Mumick, and V. Subrahmanian. Maintaining views incrementally. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 157–166, Washington, DC, May 1993.

[Gup97] H. Gupta. Selection of views to materialize in a data warehouse. In *Proc. of the Sixth International Conference on Database Theory*, pages 98–112, Delphi, Greece, January 1997.

[HGMW+95] J. Hammer, H. Garcia-Molina, J. Widom, W. Labio, and Y. Zhuge. The Stanford Data Warehousing Project. *IEEE Data Engineering Bulletin, Special Issue on Materialized Views and Data Warehousing*, 18(2):41–48, June 1995.

[HMN+99] L.M. Haas, R.J. Miller, B. Niswonger, M.T. Roth, P.M. Schwarz, and E.L. Wimmers. Transforming heterogeneous data with database middleware:

Beyond integration. *IEEE Data Engineering Bulletin*, 22(1):31–36, March 1999.

[HQGW93] N. I. Hachem, K. Qiu, M. Gennert, and M. Ward. Managing derived data in the Gaea scientific DBMS. In *Proc. of the Ninteenth International Conference on Very Large Data Bases*, pages 1–12, Dublin, Ireland, August 1993.

[HRU96] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 205–216, Montreal, Canada, June 1996.

[IK93] W.H. Inmon and C. Kelley. *Rdb/VMS: Developing the Data Warehouse.* QED Publishing Group, Boston, Massachussetts, 1993.

[Imp] Cognos: Impromptu Interactive Database Query and Reporting Tool. http://www.cognos.com/impromptu/.

[Inf] Informix Formation Data Transformation Tool. http://www.informix.com/informix/products/integration/formation/formation.html.

[Kel85] A.M. Keller. Algorithms for translating view updates to database updates for views involving selections, projections, and joins. In *Proc. of the Fourth International Conference on Principle of Database Systems*, pages 467–476, Stockholm, Sweden, March 1985.

[Kel86] A.M. Keller. Choosing a view update translator by dialog at view definition time. In *Proc. of the Twelfth International Conference on Very Large Data Bases*, pages 467–476, Kyoto, Japan, August 1986.

[KLM+97] A. Kawaguchi, D.F. Lieuwen, I.S. Mumick, D. Quass, and K.A. Ross. Concurrency control theory for deferred materialized views. In *Proc. of the Sixth International Conference on Database Theory*, pages 306–320, Delphi, Greece, January 1997.

[KU84] A.M. Keller and J.D. Ullman. On complementary and independent mappings of databases. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 143–148, Boston, Massachusetts, June 1984.

[LBM98] T. Lee, S. Bressan, and S. Madnick. Source attibution for querying against semi-structured documents. In *Proc. of the Workshop on Web Information and Data Management*, pages 33–39, Washington, DC, November 1998.

[LGMW00] W.J. Labio, H. Garcia-Molina, and J.L. Weiner. Efficient resumption of interrupted warehouse loads. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 46–57, Dallas, Texas, May 2000.

[LQA97] W.J. Labio, D. Quass, and B. Adelberg. Physical database design for data warehousing. In *Proc. of the Thirteenth International Conference on Data Engineering*, pages 277–288, Birmingham, UK, April 1997.

[LS91] J.A. Larson and A.P. Sheth. Updating relational views using knowledge at view definition and view update time. *Information Systems*, 16(2):145–168, 1991.

[LSS96] L. Lakshmanan, F. Sadri, and I.N. Subramanian. SchemaSQL – a language for interoperability in relational multi-database systems. In *Proc. of the Twenty-Second International Conference on Very Large Data Bases*, pages 239–250, Bombay, India, September 1996.

[LW95] D. Lomet and J. Widom, editors. *Special Issue on Materialized Views and Data Warehousing,* IEEE Data Engineering Bulletin 18(2), June 1995.

[Mas84] Y. Masunaga. A relational database view update translation mechanism. In *Proc. of the Tenth International Conference on Very Large Data Bases*, pages 309–320, Singapore, August 1984.

[MBR01] J. Madhavan, P. Bernstein, and E. Rahm. Generic schema matching with Cupid. In *Proc. of the Twenty-Seventh International Conference on Very Large Data Bases*, Rome, Italy, September 2001.

[Mic] Microsoft SQL Server 7.0, Data Transformation Services. http://msdn.microsoft.com/library/psdk/sql/dts_ovrw.htm.

[Ora] Oracle 8i. http://technet.oracle.com/products/oracle8i/.

[Pow] Cognos: PowerPlay OLAP Analysis Tool. http://www.cognos.com/powerplay/.

[PPD] PPD Informatics: TableTrans Data Transformation Software. http://www.belmont.com/tt.html.

[QGMW96] D. Quass, A. Gupta, I. S. Mumick, and J. Widom. Making views self-maintainable for data warehousing. In *Proc. of the Fourth International Conference on Parallel and Distributed Information Systems*, pages 158–169, Miami Beach, Florida, December 1996.

[RH01] V. Raman and J. Hellerstein. Potters Wheel: An interactive data cleaning system. In *Proc. of the Twenty-Seventh International Conference on Very Large Data Bases*, Rome, Italy, September 2001.

[RS79] L. Rowe and K.A. Schoens. Data abstractions, views, and updates in Rigel. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 71–81, Boston, Massachusetts, May 1979.

[RS98] A. Rosenthal and E. Sciore. Propagating integrity information among inter-related databases. In *Proc. of the Second Working Conference on Integrity and Internal Control in Information Systems*, pages 5–18, Warrenton, Virginia, November 1998.

[RSS96] K.A. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In *Proc. of the*

ACM SIGMOD International Conference on Management of Data, pages 447–458, Montreal, Canada, June 1996.

[Sag] Sagent Technology. http://www.sagent.com/.

[Shu87] N.C. Shu. Automatic data transformation and restructuring. In *Proc. of the Third International Conference on Data Engineering*, pages 173–180, Los Angeles, California, February 1987.

[Shu96] H. Shu. Formulating view update translation as constraint satisfaction. Technical report, University of Karlstad, Sweden, November 1996. http://www-b.informatik.uni-hannover.de/∼hs/.

[Squ95] C. Squire. Data extraction and transformation for the data warehouse. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 446–447, San Jose, California, May 1995.

[Sto75] M. Stonebraker. Implementation of integrity constraints and views by query modification. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 65–78, San Jose, California, May 1975.

[Tom94] A. Tomasic. Determining correct view update translations via query containment. In *Proc. of the Workshop on Deductive Databases and Logic Programming*, pages 75–83, Santa Margherita Ligure, Italy, June 1994.

[TPC96] Transaction Processing Performance Council. *TPC-D Benchmark Specification, Version 1.2*, 1996. http://www.tpc.org/.

[Ull89a] J. D. Ullman. *Database and Knowledge-base Systems (Vol 1)*. Computer Science Press, 1989.

[Ull89b] J. D. Ullman. *Database and Knowledge-base Systems (Vol 2)*. Computer Science Press, 1989.

[Wid95] J. Widom. Research problems in data warehousing. In *Proc. of the Fourth International Conference on Information and Knowledge Management*, pages 25–30, Baltimore, Maryland, November 1995.

[WS97] A. Woodruff and M. Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *Proc. of the Thirteenth International Conference on Data Engineering*, pages 91–102, Birmingham, UK, April 1997.

[YMHR01] L. Yan, R.J. Miller, L.M. Hass, and Fagin R. Data-driven understanding and refinement of schema mappings. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 485–496, Santa Barbra, California, May 2001.

[ZGMHW95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 316–327, San Jose, California, May 1995.