Tracing the Lineage of View Data in a Warehousing Environment*

Yingwei Cui, Jennifer Widom, Janet L. Wiener

Computer Science Department, Stanford University
{cyw, widom, wiener}@db.stanford.edu

Abstract

We consider the *view data lineage* problem in a warehousing environment: For a given data item in a materialized warehouse view, we want to identify the set of source data items that produced the view item. We formally define the lineage problem, develop lineage tracing algorithms for relational views with aggregation, and propose mechanisms for performing consistent lineage tracing in a multi-source data warehousing environment. Our results can form the basis of a tool that allows analysts to browse warehouse data, select view tuples of interest, then "drill-through" to examine the exact source tuples that produced the view tuples of interest.

1 Introduction

In a *data warehousing* system, *materialized views* over source data are defined, computed, and stored in the warehouse to answer queries about the source data (which may be stored in distributed and legacy systems) in an integrated and efficient way [CD97, Wid95]. Typically, *on-line analytical processing and mining* (OLAP and OLAM) systems operate on the data warehouse, allowing users to perform analysis and predictions [CD97, HCC98]. In many cases, not only is the view data itself useful for analysis, but knowing the set of source data that produced specific pieces of view information also can be useful. Given a data item in a materialized view, determining the source data that produced it and the process by which it was produced is termed the *data lineage* problem. Some applications of view data lineage are:

- OLAP and OLAM: Effective data analysis and mining requires facilities for data exploration at different levels. The ability to select a portion of relevant view data and "drill-through" to its origins can be very useful. In addition, an analyst may want to check the origins of suspect or anomalous view data, to verify the reliability of the sources or even repair the source data.

- Scientific Databases: Scientists apply algorithms to commonly understood and accepted source data to derive their own views and perform specific studies. As in OLAP, it can be useful for the scientist to focus on specific view data, then explore how it was derived from the original raw data.

- On-line Network Monitoring and Diagnosis Systems: From anomalous view data computed by the diagnosis system, the network controller can use data lineage to identify the faulty data within huge volumes of data dumped from the network monitors.

*This work was supported by DARPA and the Air Force Rome Laboratories under contracts F30602-95-C-0119 and F30602-96-1-0312.

- Cleansed Data Feedback: Information centers download raw data from data sources and "cleanse" the data by performing various transformations on it. Data lineage helps locate the origins of data items, allowing the system to send reports about the cleansed data back to their sources, and even link the cleansed items to the original items.

- Materialized View Schema Evolution: In a data warehouse, users may be permitted to change view definitions (e.g., add a column to a view) under certain circumstances. View data lineage can help retrofit existing view contents to the new view definition without recomputing the entire view.

- View Update Problem: Not surprisingly, tracing the origins of a given view data item is related to the well-known *view update* problem [BS81]. In Section 10.2, we discuss this relationship, and show how lineage tracing can be used to help translate view updates into corresponding base data updates.

In general, a view definition provides a mapping from the *base data* to the *view data*. Given a state of the base 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 base 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 base data and some additional information.

The warehousing environment introduces some additional challenges to the lineage tracing problem, such as how to trace lineage when the base data is distributed among multiple sources, and what to do if the sources are inaccessible or not consistent with the warehouse views. At the same time, the warehousing environment can help the lineage tracing process by providing facilities to merge data from multiple sources, and to store auxiliary information in the warehouse in a consistent fashion.

In this paper, we provide a complete solution for tracing the lineage of relational view data in a warehousing environment. In summary, we:

- Formulate the view data lineage problem by giving a declarative, inductive definition of lineage for arbitrarily complex relational views, including aggregation.

- Develop lineage tracing algorithms for relational views with aggregation, including proofs of correctness. We separately consider the problem under set semantics and bag (multiset) semantics.

- Address issues of lineage tracing in a warehousing environment, and show how to perform lineage tracing consistently and efficiently for views defined on distributed, legacy sources.

The remainder of this paper is organized as follows. Section 2 surveys related work. Section 3 motivates the data lineage problem using detailed examples, and Section 4 formalizes the problem. Sections 5, 6, and 7 present set-semantics lineage tracing algorithms for Select-Project-Join (SPJ) views, views with aggregation, and views with union and difference operators, respectively. Section 8 extends our results to bag semantics, and Section 9 discusses additional issues for lineage tracing in a warehousing environment. Section 10 revisits some related problems (e.g., the view update problem) to further clarify their relationship with the data lineage problem. Section 11 concludes and discusses future extensions to our work. All proofs are provided in the Appendix.

store_id	store_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

Figure 1: `store` table

item_id	item_name	category
0001	binder	stationery
0002	pencil	stationery
0003	shirt	clothing
0004	pants	clothing
0005	pot	kitchenware

Figure 2: `item` table

store_id	item_id	price	num_sold
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 3: `sales` table

2 Related Work

The problem of tracing view data lineage is related to work in several areas. Deductive database techniques perform top-down recursive rule-goal unification to provide proofs for a goal proposition [Ull89]. The provided proofs find the supporting facts for the goal proposition, and therefore also can be thought of as providing the proposition's lineage. In this paper, we take a different approach that designs relational queries for lineage tracing. In addition, we allow views with aggregation, which are not considered by deductive databases, and we consider lineage tracing in multi-source warehousing environments.

[Sto75] provides an algorithm to translate updates on SPJ views to updates on their base tables. *Data cube* "rolling-up" and "drilling-down" enable a user to browse a summary view and underlying detailed data at any level and any dimension of the aggregation [GBLP96]. Although both papers address problems that roughly include retrieving lineage information for specific types of relational views, neither of them formally define the view data lineage problem or tackle it in the general case.

Scientific databases support view data lineage using metadata and annotations [HQGW93]. This approach is useful for providing schema-level lineage tracing. However, for applications that require lineage at a finer (instance-level) granularity, their techniques may introduce high storage overhead. [WS97] proposes a framework to compute instance-level data lineage lazily using the view's *weak inverse mapping*. However, the system requires the view definer to provide the view's weak inverse, an expectation that may not always be practical. Our algorithms trace instance-level lineage automatically for the user and maintain the necessary auxiliary information to ensure that views have inverses.

The lineage problem also relates somewhat to the problem of reconstructing base data from summary data as in [FJS97], which uses a statistical approach and certain constraint knowledge. However, that approach provides only estimated lineage information, and does not ensure accuracy.

3 Motivating Examples

In this section, we provide examples that motivate a precise definition of data lineage and show how lineage tracing can be useful. Consider a data warehouse with retail store data over three source tables, whose schema and sample contents are shown in Figures 1–3. The `store` and `item`

```
CREATE VIEW Calif AS
SELECT store_name, item_name, num_sold
FROM   store, item, sales
WHERE  sales.store_id = store.store_id AND
       sales.item_id = item.item_id AND
       store.state = "CA"
```

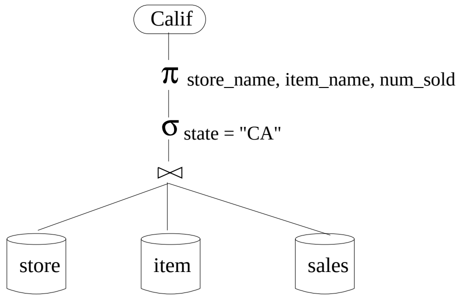


Figure 4: View definition for `Calif`

store_name	item_name	num_sold
Target	binder	1000
Target	pencil	3000
Target	pants	600
Macy's	shirt	1500
Macy's	pants	600

Figure 5: `Calif` table

store				item			sales			
s_id	s_name	city	state	i_id	i_name	category	s_id	i_id	price	num_sold
001	Target	Palo Alto	CA	0002	pencil	stationery	001	0002	1	3000

Figure 6: `Calif` lineage for `<Target, pencil, 3000>`

tables are self-explanatory. The `sales` table contains sales information, including the price and number of each product sold at that price by each store.

Example 3.1 (Lineage of SPJ View) 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 4 shows the SQL and relational algebra definitions of `Calif`. The materialized view for `Calif` over our sample data is shown in Figure 5.

The analyst browses the view table and is interested in the second tuple `<Target, pencil, 3000>`. He would like to see the relevant detailed information and asks question Q1: "Which base data produced tuple `<Target, pencil, 3000>` in view `Calif`?" Using the algorithms we present in Section 5, we obtain the answer in Figure 6. The answer tells us that the Target store in Palo Alto sold 3000 pencils at a price of 1 dollar each. □

Example 3.2 (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 7. We extend traditional relational algebra with an aggregation operator, denoted $\alpha_{G,aggr(B)}$, where G is a list of groupby attributes, and $aggr(B)$ abbreviates a list of aggregate functions over attributes. (Details are given in Section 4.1.) The materialized view contains one tuple, `<5400>`.

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

```
CREATE VIEW Clothing AS
SELECT   sum(num_sold) as total
FROM     item, store, sales
WHERE    sales.store_id = store.store_id AND
         sales.item_id = item.item_id AND
         item.category = "clothing"
GROUP BY store_name
HAVING   total > 5000
```

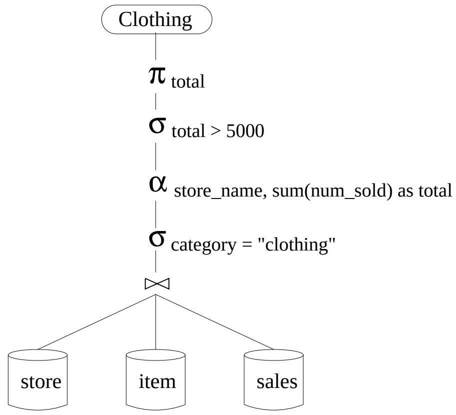


Figure 7: View definition for `Clothing`

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_sold
003	0003	45	1500
003	0004	60	600
004	0003	50	2100
004	0004	70	1200

Figure 8: `Clothing` lineage for `<5400>`

appropriate algorithms in Section 6, and Figure 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 the next section, we formally define the concept of derivation. Sections 5–9 then present algorithms to compute view tuple derivations for different view and warehouse scenarios.

4 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. Section 4.1 first reviews relational view semantics. Tuple derivations for operators and views are then defined in Sections 4.2 and 4.3, respectively.

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 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 this section as well as in Sections 5, 6, and 7. We adapt our work to *bag semantics* (duplicates permitted) in Section 8.

4.1 Views

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:

- Base case: $R = \{t \mid t \in R\}$
- *Selection*: $\sigma_C(V_1) = \{t \mid t \in V_1 \text{ and } t \text{ satisfies } C\}$
- *Projection*: $\pi_A(V_1) = \{t.A \mid t \in V_1\}$[1]
- *Join*: $\bowtie_\theta(V_1, \ldots, V_m) = \{\langle t_1, \ldots, t_m\rangle \mid t_i \in V_i \text{ for } i = 1..m \text{ and the } t_i\text{'s satisfy condition } \theta\}$

  We use infix notation and the special case of natural join $\bowtie$ in most of the paper, although all results and algorithms hold directly for the general case of theta join.
- *Aggregation*: $\alpha_{G,aggr(B)}(V_1) = \{\langle T.G,\ aggr(T.B)\rangle \mid T \subseteq V_1 \text{ and } \forall t,\ t' \in T,\ t'' \notin T\text{:}\ t'.G = t.G \ \wedge\ t''.G \neq t.G\}$
- *Set Union*: $V_1 \cup \cdots \cup V_m = \{t \mid t \in V_i \text{ for some } i \in 1..m\}$
- *Set Difference*: $V_1 - V_2 = \{t \mid t \in V_1 \text{ and } t \notin V_2\}$

Thus, the grammar of our view definition language is as follows:

$$V :- \ R \mid \sigma_C(V_1) \mid \pi_A(V_1) \mid V_1 \bowtie \cdots \bowtie V_m \mid \alpha_{G,aggr(B)}(V_1) \mid V_1 \cup \cdots \cup V_m \mid V_1 - V_2$$

where R is a base table, $V_1, \ldots, V_m$ are views, C is a selection condition (any boolean expression) on attributes of V_1, A is a projection attribute list from V_1, G is a groupby attribute list from V_1, and $aggr(B)$ abbreviates a list of aggregation functions applied to attributes of V_1.

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 4 and 7 are examples of query trees.

4.2 Tuple Derivations for Operators

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 generates its result table based on the results of its children nodes, and passes its result table upwards. We begin by focusing on individual operators, defining derivations of the operator's result tuples based on its input tuples.

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 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

[1]Note that π is duplicate-eliminating here, and we generally abuse notation by writing $t.A$ for $\langle t.A_1, \ldots, t.A_n\rangle$ where $A = \{A_1, \ldots, A_n\}$.

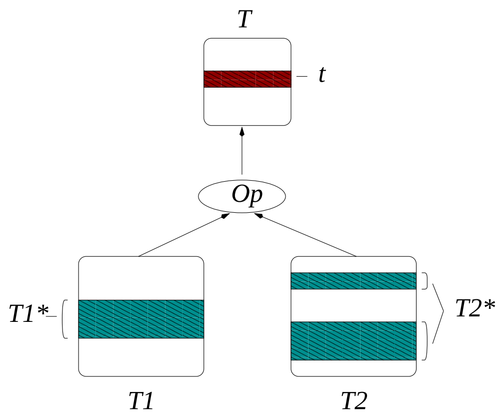


Figure 9: Derivation of tuple t

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 4.1 (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 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 $i = 1..m$. $\square$

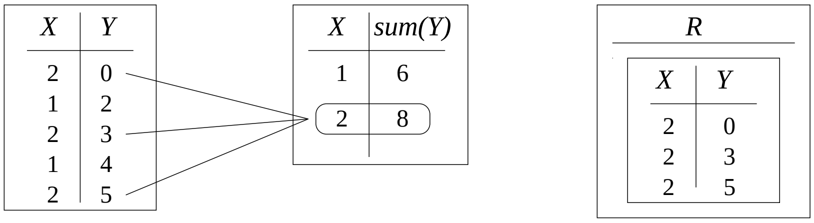
In Definition 4.1, requirement (a) says that the derivation tuple sets (the $T_i{}^*$'s) derive exactly t. From relational semantics, we know that for any result tuple t, there must exist such tuple sets. Requirement (b) says that each tuple in the derivation does in fact contribute something to t. For 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]

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 4.2 shows that there is a unique derivation for any operator and result tuple. Note that all proofs are provided in the Appendix.

[2]By Definition 4.1, 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 7.



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

Figure 10: Tuple derivation for aggregation

Theorem 4.2 (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. $\Box$

Example 4.3 (Tuple Derivation for Aggregation) Given table R in Figure 10(a), and tuple $t = \langle 2, 8\rangle \in \alpha_{X,sum(Y)}(R)$ in Figure 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 10(c). Notice that R's subset $\{\langle 2, 3\rangle, \langle 2, 5\rangle\}$ also satisfies requirements (a) and (b) in Definition 4.1, 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. $\Box$

From Definition 4.1 and the semantics of the operators introduced in Section 4.1, we now specify the actual tuple derivations for each of our operators.

Theorem 4.4 (Tuple Derivations for Operators) Let $T, T_1, \ldots, T_m$ be tables. Recall that $\mathbf{T_i}$ denotes the schema of T_i.

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

4.3 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.

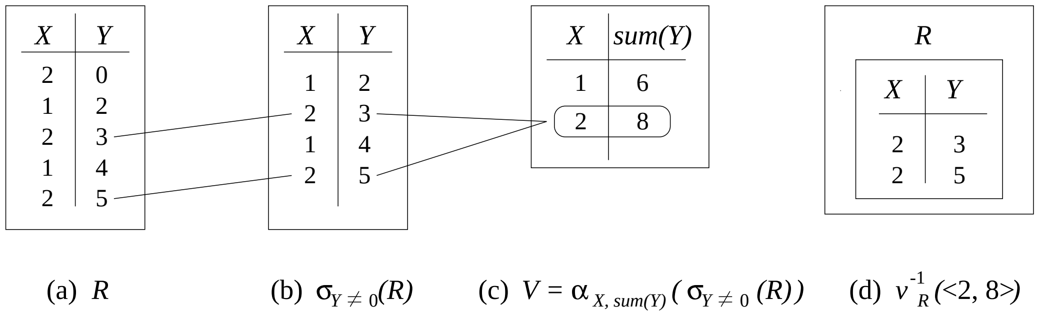


Figure 11: Tuple derivation for a view

Definition 4.5 (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. $v = R_i$: 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 4.1), 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:

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

Theorem 4.2 can be applied inductively in the obvious way to show that view tuple derivations are always unique.

Example 4.6 (Tuple Derivation for a View) Given base table R in Figure 11(a), view $V = \alpha_{X,sum(Y)}(\sigma_{Y \neq 0}(R))$ in Figure 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 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 11(d). $\Box$

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

Theorem 4.7 (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] $\square$

Theorem 4.7 follows from Definition 4.5. 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 $\forall D$: $v_1(D) = v_2(D)$ [Ull89]. We prove in Theorem 4.8 that given any two equivalent Select-Project-Join (SPJ) views, their tuple derivations also are equivalent.

Theorem 4.8 (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, $\forall D$: $\forall t \in v_1(D) = v_2(D)$: ${v_1^{-1}}_D(t) = {v_2^{-1}}_D(t)$. $\square$

Thus, according to Theorem 4.8, we can transform an 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 4.9 (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 10(a) and 11(a), Figures 10(b) and 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 10(c)) is different from that according to v_2 (shown in Figure 11(d)). $\square$

Given Definition 4.5, 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, views with aggregation (ASPJ views), and more general views with set operators. Section 5 shows that one relational query over the base tables suffices to compute tuple derivations for SPJ views. Sections 6 and 7 present recursive algorithms that require a modest amount of auxiliary information for ASPJ and more general view derivation tracing.

5 SPJ View Derivation Tracing

Derivations for tuples in Select-Project-Join (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.

5.1 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 4.5). We

[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.

call such a query a *derivation tracing query (or tracing query) for t and v*. More formally, we have:

Definition 5.1 (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 $TQ_{T,v}(D)$. $\square$

5.2 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 [Ull89]. We call this form the *SPJ canonical form*. From Theorem 4.8, 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 5.2 (*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 $A_i \subseteq \mathbf{T}$, $i = 1..m$.

$$Split_{A_1,\ldots,A_m}(T) = \langle \pi_{A_1}(T), \ldots, \pi_{A_m}(T) \rangle \qquad \square$$

Theorem 5.3 (Derivation 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:

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

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

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

where $\ltimes$ is the relational semijoin operator. $\square$

Example 5.4 (Tracing Query for `Calif`) Recall Q1 over view `Calif` in Example 3.1, where we asked about the derivation of tuple `<Target, pencil, 3000>`. Figure 12(a) shows the tracing query for `<Target, pencil, 3000>` in `Calif` according to Theorem 5.3. The reader may verify that by applying the tracing query to the source tables in Figures 1, 2, and 3, we obtain the derivation result in Figure 6. $\square$

5.3 Tracing Query Optimizations

The derivation tracing queries in Section 5.2 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 12(b) shows the optimized tracing query for the `Calif` tuple. If sufficient base table key information is present in the view, the tracing query can be even simpler:

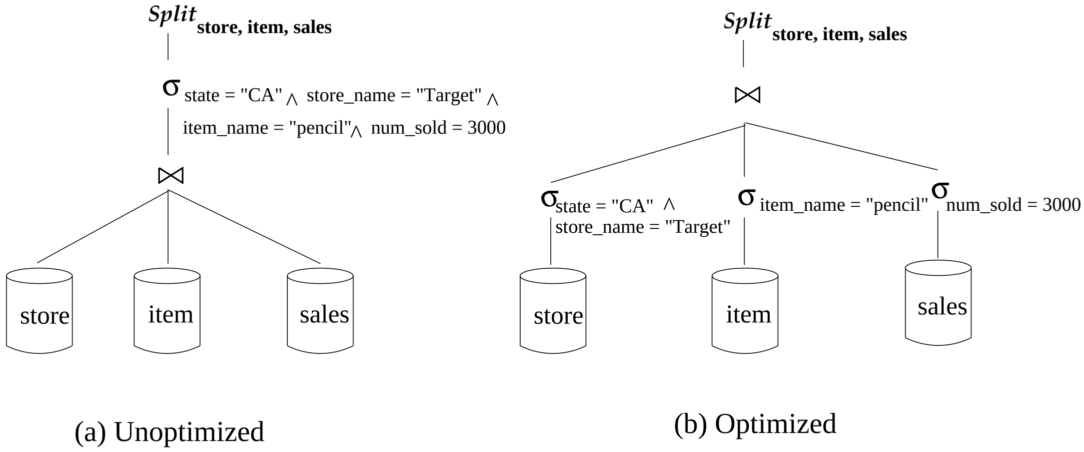


Figure 12: Derivation tracing queries for `<Target, pencil, 3000>` in view `Calif`

Theorem 5.5 (Derivation Tracing using Key Information) Let R_i be a base table with key attributes K_i, $i = 1..m$, and let view $V = \pi_A(\sigma_C(R_1 \bowtie \cdots \bowtie R_m))$ include all base table keys (i.e., $K_i \in A$, $i = 1..m$). View tuple t's derivation is $\langle \sigma_{K_1=t.K_1}(R_1), \ldots, \sigma_{K_m=t.K_m}(R_m) \rangle$. $\square$

According to Theorem 5.5, 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.

6 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 3.2 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 $t.\mathtt{total} = 5400$. In order to trace the derivation correctly, we need tuple $\langle$Macy's, 5400$\rangle$ in the intermediate aggregation result to serve as a "bridge" that connects the base tables and the view table.

We introduce a canonical form for ASPJ views in Section 6.1. In Section 6.2, we specify a derivation tracing query for simple "one-level" ASPJ views. We then develop a recursive tracing algorithm for complex ASPJ views and justify its correctness in Section 6.3. 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 further discussed in Section 9.1. In the remainder of this section we simply assume that all intermediate aggregation results are available.

6.1 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 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 [Ull89], 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 6.1 (ASPJ Canonical Form) Let v be an ASPJ view definition 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 non-trivial topmost aggregation operator, $j = 1..k$. $\Box$

As mentioned in Section 4, 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.

6.2 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 6.2 (Derivation Tracing Query for a One-Level ASPJ View) Given a 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 given tuple $t \in V$, t's derivation in $R_1, \ldots, R_m$ according to v can be computed by applying the following query to the base tables:

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

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

$$TQ_{T,v} = Split_{\mathbf{R_1},\ldots,\mathbf{R_m}}(\sigma_C(R_1 \bowtie \cdots \bowtie R_m) \ltimes T) \qquad \Box$$

Here too, evaluation of the tracing query can be optimized in various ways as discussed in Section 5.3.

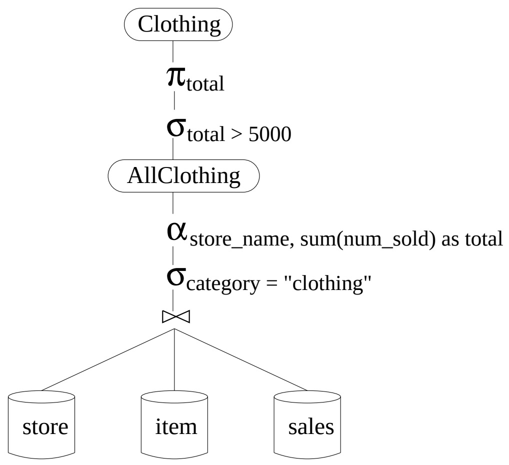


Figure 13: ASPJ segments and intermediate view for `Clothing`

6.3 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 6.3 (ASPJ Segments and Intermediate Views for `Clothing`) Recall the view `Clothing` in Example 3.2. We can rewrite its definition in ASPJ canonical form with two segments, and introduce an intermediate view `AllClothing` as shown in Figure 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. Details follow.

6.3.1 Algorithm

Figure 14 presents our basic 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 `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 6.2. It then calls procedure `TableListDerivation`$(\{V_1^*, \ldots, V_k^*\}, \{v_1, \ldots, v_k\}, D)$, 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 6.4 (Recursive Derivation Tracing) We divided the view `Clothing` into two segments in Example 6.3. We assume that the contents of the intermediate view `AllClothing` are available (shown in Figure 15). According to our algorithm, we first compute the derivation T_1^* of `<5400>` in `AllClothing` to obtain $T_1^* = \{$`<Macy's, 5400>`$\}$, then trace T_1^*'s derivation to the base tables to obtain the derivation result in Figure 8. □

```
procedure TupleDerivation(t, v, D)
input:    a tracing tuple t, a view definition v, and a database D
output:   t's derivation in D according to v
begin
          return (TableDerivation({t}, v, D));
end
```

```
procedure TableDerivation(T, v, D)
input:    a tracing tuple set T, a view definition v, and a database D
output:   T's derivation in D according to v
begin
          if v = R ∈ D then return (⟨T⟩);
          // otherwise v = v'(v_1, ..., v_k) where v' is a one-level ASPJ view,
          // V_j = v_j(D) is an intermediate view or a base table, j = 1..k
          ⟨V_1*, ..., V_k*⟩ ← TQ(T, v', {V_1, ..., V_k});
          return (TableListDerivation(⟨V_1*, ..., V_k*⟩, {v_1, ..., v_k}, D));
end
```

```
procedure TableListDerivation(⟨T_1, ..., T_k⟩, {v_1, ..., v_k}, D)
input:    a list of tuple sets T_1, ..., T_k to be traced,
          a list of view definitions v_1...., v_k, and a database D
output:   the concatenation of T_i's derivations according to v_i, i = 1..k
begin
          D* ← ∅;
          for j ← 1 to k do
              D* ← D*∘ TableDerivation(T_j, v_j, D);
          return (D*);
end
```

Figure 14: Derivation tracing algorithm for ASPJ views

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 discussed in Section 9.1.

6.3.2 Correctness

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

1. Transforming a view into ASPJ canonical form does not affect its derivation.

   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 [Ull89]. Theorem 4.8 shows that derivations are unchanged by SPJ transformations.

2. It is correct to trace derivations recursively down the view definition tree.

   From Theorem 4.7, 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.

store_name	total
Target	1400
Macy's	5400

Figure 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.

7 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 7.1, we briefly review tuple derivations for set operators, and provide an example justifying the definition for the difference operator. In Section 7.2, 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 7.3 we present a recursive algorithm that traces tuple derivations for general views.

7.1 Tuple Derivations for Set Operators

Recall from Theorem 4.4 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 4.1 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 $\neg\exists t \in T_2$. 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 7.1 further explains the necessity of including T_2 in the derivation of $t \in T_1 - T_2$.

Example 7.1 (Tuple Derivation for Difference) Consider tables R, S, T in Figure 16(a), and view $V = R - (S - T)$ in Figure 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 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 lineage 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. □

7.2 Canonical Form for General Views

We now define an extended canonical form that accommodates views with union and difference operators, as well as aggregate, select, project, and join operators. The extended canonical form

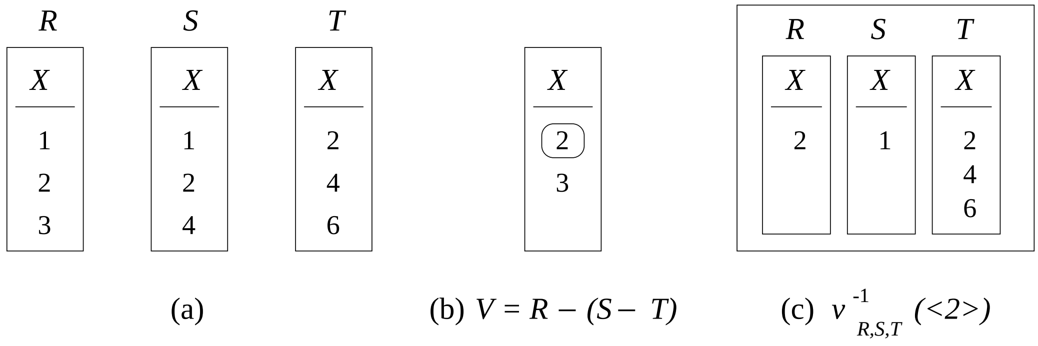


Figure 16: Tuple derivation for set difference

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 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 aggregate 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 6.1. In Figure 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 $\pi_A(R) \bowtie_\theta \pi_B(S) \equiv \pi_{A \cup B}(R \bowtie_\theta S)$[4]
3. pulling up σ: $\sigma_C(R) \bowtie \sigma_{C'}(S) \equiv \sigma_{C \wedge C'}(R \bowtie S)$

As in Figure 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 sometimes push selection and projection operators back below the join to reduce the cost of the tracing query, as discussed in Section 5.3.

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

[4] Although we use natural join throughout most of the paper, all results carry over directly to theta join. For this equivalence to hold in general, it is necessary to make the theta explicit.

```
procedure Canonicalize(v0)
input:    a general view definition tree v0
output:   a canonicalized view definition tree v with an intermediate view
          associated with each segment in v
begin
          copy v0 to v;
          pull union operators above selections and projections in v;
          pull projection operators above joins and selections in v;
          pull selection operators above joins in v;
          merge the same algebra operators adjacent to each other in v;
          divide v into segments;
          define an intermediate view over each segment in v;
          return (v);
end
```

Figure 17: Canonicalizing general view definitions

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 7.2 (Extended Canonical Form) Procedure Canonicalize in Figure 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. □

Example 7.3 (Canonical Form for a View with Set Operators) Consider the general view definition in Figure 18(a). Figure 18(b) shows the canonical form and its segments. Notice that σ_{11} is pulled up and merged with σ_1 to obtain $\sigma_{1\wedge 11}$, and π_8 is subsumed by π_5. □

7.3 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 7.4 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 4.4.

Theorem 7.4 (Derivation 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:

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

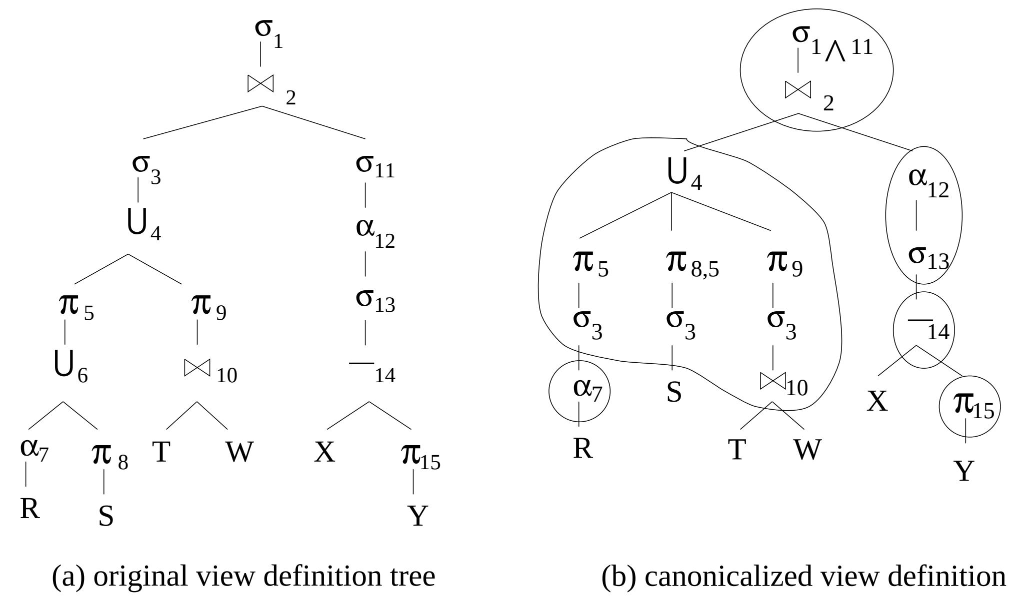


Figure 18: Canonical form for a general view

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

$$TQ_{T,v} = \bigodot_{j=1..k} Split_{\mathbf{R^j_1},\ldots,\mathbf{R^j_{l_j}}}(\sigma_C(R^j_1 \bowtie \cdots \bowtie R^j_{l_j}) \ltimes T)$$

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:

$$TQ_{\mathbf{ALL},v} = \bigodot_{j=1..k} Split_{\mathbf{R^j_1},\ldots,\mathbf{R^j_{l_j}}}(\sigma_C(R^j_1 \bowtie \cdots \bowtie R^j_{l_j})) \qquad \Box$$

The recursive tracing algorithm for ASPJ views (Figure 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 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, $i = 1..k$; (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 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 7.4 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 7.1. Recall from Theorem 7.4 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.

```
procedure TableDerivation(T, v, D)
input:    a tracing tuple set T (or the special symbol ALL),
          a view definition v, and a database D
output:   T's derivation in D according to v
begin
          case v = R ∈ D:
              if T = ALL then return (⟨R⟩);
              else return (⟨T⟩);

          case v = v'(v_1, ..., v_k):
              // v' is a one-level ASPJ view,
              // V_j = v_j(D) is an intermediate view or a base table, j = 1..k
              ⟨V_1*, ..., V_k*⟩ ← TQ(T, v', {V_1, ..., V_k});
              return (TableListDerivation(⟨V_1*, ..., V_k*⟩, {v_1, ..., v_k}, D));

          case v = v_1 − v_2:
              // V = v(D),
              // V_j = v_j(D) is an intermediate view or a base table, j = 1,2
              if T = ALL then T ← V;
              return (TableListDerivation(⟨T, ALL⟩, ⟨v_1, v_2⟩, D));
end
```

Figure 19: Derivation tracing algorithm for general views

8 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 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 ($\dot{-}$). 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 Figure 20. 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 5.3. 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.

8.1 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 N or more ways to derive t from the T_i's, possibly because of duplicates in the T_i's,

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

Figure 20: Base table and view scenarios

projection on non-key attributes, or because Op is a bag union. Thus, there may no longer be a unique derivation for t according to Definition 4.1. If Op is any operator except difference, there are exactly N derivations for t, each of which derives a copy of t. If Op is a difference operator, the number of derivations may exceed the number of t's in the result.

Similarly, given a view v, a tuple $t \in v(D)$ may have multiple derivations. 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 8.2 and 8.3 to cover both cases. First, we redefine tuple derivations for a view under bag semantics, because our previous definition (Definition 4.5), although clearer, does not generalize to bag semantics. Definition 8.1 subsumes Definition 4.5.

Definition 8.1 (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 4.1), 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 $\langle R_1^{1*}, \ldots, R_{l_k}^{k\,*} \rangle$ 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. $\Box$

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 8.2) can be impractical, and we may prefer to trace the tuple set's derivation pool (Definition 8.3).

Definition 8.2 (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 $v^{-S}{}_D(t)$.[5] The derivation set of a tuple set $T \subseteq Op(T_1, \ldots, T_m)$ is the set of all derivations of T, denoted $v^{-S}{}_D(T)$. $\Box$

Definition 8.3 (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 8.1). $\langle R_1{}^*, \ldots, R_m{}^* \rangle$ 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 $v^{-P}{}_D(T)$. $\Box$

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

store_name	item_name
Target	binder
Target	pencil
Target	binder
Target	pencil

Figure 21: `Stationary` 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_sold
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_sold
002	0002	2	2000

Figure 22: Multiple derivations of ⟨`Target, pencil`⟩

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_sold
001	0002	1	3000
002	0002	2	2000

Figure 23: Derivation pool for ⟨`Target, pencil`⟩

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 8.4 (Multiple Derivations, Derivation Set, Derivation Pool) Consider a view `Stationary` defined over our original tables `store`, `item`, and `sales` from Figures 1–3: `Stationary` $= \pi_{\texttt{store\_name},\texttt{item\_name}}(\sigma_{\texttt{category}=\text{“}\texttt{stationary}\text{”}}(\texttt{store} \bowtie \texttt{item} \bowtie \texttt{sales}))$, where π now does not eliminate duplicates. Figure 21 shows `Stationary`'s contents. A single tuple $t =$ ⟨`Target, pencil`⟩ in `Stationary` has two derivations, D_1^* and D_2^*, as shown in Figure 22, so t's derivation set is $\{D_1^*, D_2^*\}$. Figure 23 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 7.2) 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 4.7) 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 8.5 (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 $v^{-1}{}_D(t) = v^{-P}{}_D(t)$. □

Theorem 8.6 (Derivation Pool of a Tuple Set with 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 $v^{-P}{}_D(\sigma_{\mathbf{V}=t}(v(D))) = v^{-P}{}_D(t)$. $\Box$

Theorem 8.7 (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 $v' = \sigma_C(v)$. $\Box$

In Sections 8.2, 8.3, and 8.4, we now 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.

8.2 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 5–7. We begin by specifying the derivation pools for all of the relational operators, using bag semantics. Note that except for $\dot{-}$, the only real change from Theorem 4.4 is to replace $\{t\}$ with $\sigma_{\mathbf{T}=t}(T)$ in several places.

Theorem 8.8 (Derivation Pools for Operators with Bag Semantics) Let T, $T_1, \ldots, T_m$ be tables. Recall that $\mathbf{T_i}$ denotes the schema of T_i.

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

We show in Theorem 8.10 that the derivation pool tracing query for SPJ views can be obtained by replacing the *Split* operator in Theorem 5.3 with a *targeted split (TSplit)* operator.

Definition 8.9 (*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:

$$TSplit_{T_1,\ldots,T_m}(T) = \langle T_1 \ltimes T, \ldots, T_m \ltimes T\rangle \quad \Box$$

Theorem 8.10 (Derivation Pool Tracing Query for an SPJ View) Let D be a database with base tables $R_1, \ldots, R_m$, and let $V = v(D) = \pi_A(\sigma_C(R_1 \bowtie \ldots \bowtie R_m))$ be an SPJ view over D. Given tuple $t \in V$, t's derivation pool in D according to v can be computed by applying the following query to the base tables:

$$TQ_{t,v} = TSplit_{R_1,\ldots,R_m}(\sigma_{C\wedge A=t}(R_1 \bowtie \ldots \bowtie R_m))$$

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

$$TQ_{T,v} = TSplit_{R_1,\ldots,R_m}(\sigma_C(R_1 \bowtie \ldots \bowtie R_m) \ltimes T) \qquad \Box$$

```
procedure TableDerivation(T, v, D)
input:   a tracing tuple set T, a view definition v, and a database D
output:  T's derivation pool in D according to v
begin
         case v = R ∈ D:
            if T = ALL then return (⟨R⟩);
            else return (⟨R ⋉ T⟩);

         case v = v'(v_1, ..., v_k):
            // v' is a one-level AUSPJ view,
            // V_j = v_j(D) is an intermediate view or a base table, j = 1..k
            ⟨V_1^*, ..., V_k^*⟩ ← TQ(T, v', {V_1, ..., V_k});
            return (TableListDerivation(⟨V_1^*, ..., V_k^*⟩, {v_1, ..., v_k}, D));

         case v = v_1 ∸ v_2:
            // V = v(D),
            // V_j = v_j(D) is an intermediate view or a base table, j = 1,2
            if T = ALL then T ← V;
            if T contains only tuples with value t
            then return (TableListDerivation(⟨T, ALL⟩, ⟨v_1, σ_{V_2≠t}(v_2)⟩, D));
            else return (TableListDerivation(⟨T, ALL⟩, ⟨v_1, v_2⟩, D));
end
```

Figure 24: Derivation pool tracing algorithm

We can similarly construct the derivation pool tracing query for one-level AUSPJ views (which subsumes the tracing query for one-level ASPJ views) by replacing the *Split* operator in Theorem 7.4 with *TSplit*. Notice that since *TSplit* (which performs semijoins) is less efficient than *Split* (which performs projections), we recommend using the original tracing query when the base tables are known to have no duplicates.

The derivation pool tracing procedure is obtained by making only small modifications to procedure `TableDerivation` from Figure 19, shown as the underlined portions in Figure 24. 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* from Theorem 8.10 instead of using *Split* from Theorem 7.4. 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 8.8. 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.

8.3 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 25 traces the derivation set of a tuple t according to a general view v. We assume that v is already in canonical form (Section 7.2). 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 non-trivial aggregation node. These cases map to the five cases in Figure 25, in order.

For case (1), according to Definition 8.1 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 8.11:

Theorem 8.11 (Derivation Set for an SPJ View) Given an SPJ view $V = v(T_1, \ldots, T_m) = \pi_A(\sigma_C(T_1 \bowtie \cdots \bowtie T_m))$ and a tuple $t \in V$, t's derivation set according to v is

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

We initialize a *Pending* set to be $\sigma_{C\wedge A=t}(v_1(D) \bowtie \cdots \bowtie v_k(D))$. According to Theorem 8.11, $\forall t' \in Pending$: $D^* = \langle\{t'.\mathbf{V_1}\}, \ldots, \{t'.\mathbf{V_k}\}\rangle$ 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 8.1, 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 non-trivial aggregation operator. Since v has no duplicates due to the aggregation, by Theorem 8.5 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 25 is called to trace the derivation set of a tuple set T according to a view v. Recall from Definition 8.1 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 24). In case (4), if v does not contain any difference operators, then by Theorem 8.5 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**.

```
procedure DerivationEnum(t, v, D)
input:   a tracing tuple t, a view definition v, and a database D
output:  the set of all t's derivations in D according to v
begin
        DS ← ∅;
        case v = R ∈ D:
           for each tuple in σ_{R=t}(R) do
              insert ⟨{t}⟩ into DS;
        case v = π_A(σ_C(v_1 ⋈ ··· ⋈ v_k)):
           Pending ← σ_{C∧A=t}(v_1(D) ⋈ ··· ⋈ v_k(D));
           while Pending ≠ ∅ do
           begin
              pick a tuple t' in Pending;
              for j ← 1 to k do
                 DS_j ← DerivationEnum(t'.V_j, v_j, D);
              insert all elements of DS_1 × ··· × DS_k into DS;
              remove t' and all its duplicates from Pending;
           end
        case v = v_1 ⊎ ··· ⊎ v_k:
           for j ← 1 to k do
              insert all elements of DerivationEnum(t, v_j, D) into DS;
        case v = v'(v_1, ... v_k) where v' is an AUSPJ segment with aggregation:
           ⟨V_1*, ..., V_k*⟩ = TQ({t}, v', ⟨V_1, ..., V_k⟩);
           for j ← 1 to k do
              DS_j ← TableDerivEnum(V_j*, v_j, D);
           DS ← DS_1 × ··· × DS_k;
        case v = v_1 ∸ v_2:
           DS ← DerivationEnum(t, v_1, D) × TableDerivEnum(ALL, σ_{V_2≠t}(v_2), D);
        return (DS);
end
```

```
procedure TableDerivEnum(T, v, D)
input:   a tracing tuple set T, a view definition v, and a database D
output:  the set of all T's derivations in D according to v
begin
        let T = {t_1, ..., t_n};
        for i ← 1 to n do
           DS_i ← DerivationEnum(t_i, v, D);
        DS ← ∅;
        for each element D_1, ..., D_n in DS_1 × ··· × DS_n do
           if ∀i ≠ j : D_i ≠ D_j then insert D_1 ∪ ··· ∪ D_n into DS;
        return (DS);
end
```

Figure 25: Derivation enumeration algorithm

8.4 Using Key Information

We now propose an alternative approach to tracing view tuple derivations in the presence of duplicates for multi-level ASPJ views. Suppose for now that view v's base tables all have keys, i.e., case (3) in Figure 20. 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 8.12 (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 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 ${v^{-S}}_D(t) = \{{v'^{-1}}_D(t_j),\ j = 1..n\}$. $\square$

According to Theorem 8.12, 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 5.5 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 8.3.

Example 8.13 (SPJ View Extended with Keys) Consider again the problem of tracing the derivation of ⟨`Target, pencil`⟩ according to view `Stationary` in Example 8.4. Suppose table `store` has key `store_id`, `item` has key `item_id`, and `sales` has key ⟨`store_id, item_id`⟩. We first define a view `ExtStationary` $= \pi_{\texttt{store\_name,item\_name,store\_id,item\_id}}(\sigma_{\texttt{category=``stationary''}}(\texttt{store} \bowtie \texttt{item} \bowtie \texttt{sales}))$. Figure 26 shows the view contents. Let us map the traced tuple t = ⟨`Target, pencil`⟩ in `Stationary` to t' = ⟨`Target, pencil, 001, 0002`⟩ in `ExtStationary`. We retrieve t''s derivation according to `ExtStationary` and obtain the result in Figure 27. We can similarly map another copy of ⟨`Target, pencil`⟩ in `Stationary` to t'' = ⟨`Target, pencil, 002, 0002`⟩ in `ExtStationary`, and retrieve the second derivation, shown in Figure 28. Notice that Figures 27 and 28 are identical to the two derivations in Figures 22, as desired. $\square$

Given a multi-level 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 8.5). 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 8.12. (For the cases where T_i is an intermediate aggregation view, its key is the groupby 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 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. We leave the extension as an exercise for the reader.

store_name	item_name	store_id	item_id
Target	binder	001	0001
Target	pencil	001	0002
Target	binder	002	0001
Target	pencil	002	0002

Figure 26: `ExtStationary` contents

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_sold
001	0002	1	3000

Figure 27: Derivation of $\langle$`Target`, `pencil`, `001`, `0002`$\rangle$

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_sold
002	0002	2	2000

Figure 28: Derivation of $\langle$`Target`, `pencil`, `002`, `0002`$\rangle$

8.5 Algorithms Summary

We have presented four major derivation tracing algorithms:

1. The *basic tracing algorithm* (Sections 5–7) traces view tuple derivations under set semantics for general views.

2. The *pool tracing algorithm* (Section 8.2) 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 8.3) 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 8.4) 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 multi-level ASPJ views and may require additional machinery to generate tuple IDs.

9 Derivation Tracing in a Warehousing Environment

In a distributed, multi-source data warehousing environment, querying the sources for lineage information can be difficult or impossible: sources may be inaccessible, expensive to access, expensive to transfer data from, and/or inconsistent with the views at the warehouse. In Section 9.1, we consider the trade-offs between materializing and recomputing intermediate results in the view definition tree, and conclude that storing intermediate aggregation results improves overall lineage tracing and view maintenance performance. In Section 9.2, we discuss how we can reduce or entirely avoid source queries, and perform efficient and consistent lineage tracing, by storing modest amounts of additional auxiliary data from the source tables in the warehouse.

9.1 Materializing Intermediate Aggregate Results

In Sections 6 and 7, we saw that intermediate aggregation results are needed for derivation tracing in the general case. There are two approaches to obtaining the necessary results. One approach is to recompute the required intermediate data during the tracing process. This approach requires no extra storage or maintenance cost, but the tracing process takes longer, especially when the recomputation may require querying the sources. The other approach is to maintain materialized *auxiliary views* containing the intermediate results. In this approach, less computation is required at tracing time, but the auxiliary views must be stored and kept up-to-date. Because efficient incremental maintenance of multi-level aggregate views generally requires materializing the same intermediate views we need for lineage tracing [QGMW96], we take the materialized approach.

Example 9.1 (Materialized View for `AllClothing`) To improve the efficiency of the tracing process in Example 6.4, we materialize auxiliary view `AllClothing` (in Figure 13) with the contents shown in Figure 15. □

Note that when materializing `AllClothing`, the tuple ⟨Target, 1400⟩ is never used when tracing tuple derivations for `Clothing` since it is filtered out by the selection condition $\sigma_{total>5000}$ in `Clothing`'s definition. In this case, materializing the result of $V' = \sigma_{total>5000}$(`AllClothing`) instead of `AllClothing` seems to be a better choice. However, notice that V' is not incrementally maintainable without storing `AllClothing`. Thus, we would either need to recompute V' for each relevant update, which would incur a high maintenance cost, or we need to store all of `AllClothing` in order to maintain V'. Therefore, materializing V' is not actually an improvement. In general, given a view definition tree where selections are pushed down as far as possible, all selection conditions above an aggregate are selections on the aggregated attributes, and therefore are not incrementally maintainable without storing the entire aggregation results.

9.2 Derivation Views

By storing auxiliary data based on the source tables in the warehouse, we can reduce or entirely avoid expensive source queries during lineage tracing. There is a wide variety of possible auxiliary views to maintain, with different performance tradeoffs in different settings. A simple extreme solution is to store a complete copy of each source table in the warehouse. Our tracing procedures will then query these source table copies as if they are the sources. However, this solution can be costly and wasteful if a source is large, especially if much of its data does not contribute to the view. Also, computing selections and joins over large source table copies each time a tuple's derivation is traced can be expensive. Other solutions can in some cases store much less information and still enable derivation tracing without accessing the sources, but the maintenance cost is higher. We propose an intermediate scheme that achieves low tracing query cost with modest extra storage and maintenance cost. For the following discussion, we focus on general ASPJ views with set semantics.

After adding auxiliary views for intermediate aggregation results as described in Section 9.1, the views of concern are those defined by the lowest-level segments, which are directly over the

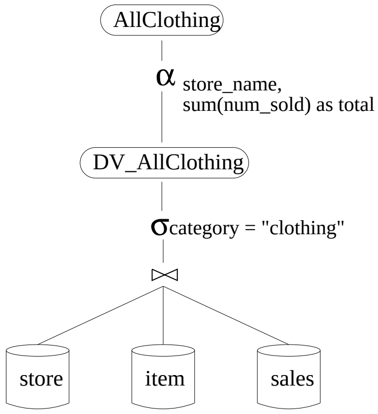


Figure 29: View definition for `DV_AllClothing`

source tables. Let $V = v(R_1, \ldots, R_m)$ be such a view. We introduce an auxiliary view, called the *derivation view for v*. It contains information about the derivation of each tuple in V over $R_1, \ldots, R_m$, as specified in Definition 9.2 given next. Theorem 9.3, which follows directly from Theorem 6.2, then shows that any V tuple's derivation in $R_1, \ldots, R_m$ can be computed with a simple selection and split operation over the derivation view.

Definition 9.2 (Derivation View) Let $V = v(R_1, \ldots, R_m) = \alpha_{G,aggr(B)}(\pi_A(\sigma_C(R_1 \bowtie \cdots \bowtie R_m)))$ be a one-level ASPJ view over source tables $R_1, \ldots, R_m$. The *derivation view for v*, denoted $DV(v)$, is

$$DV(v) = \sigma_C(R_1 \bowtie \cdots \bowtie R_m) \qquad \Box$$

Theorem 9.3 (Derivation Tracing using the Derivation View) Let V be a one-level ASPJ view over base tables: $V = v(R_1, \ldots, R_m) = \alpha_{G,aggr(B)}(\pi_A(\sigma_C(R_1 \bowtie \cdots \bowtie R_m)))$. Let $DV(v)$ be v's derivation view as defined in Definition 9.2. Given a tuple $t \in V$, t's derivation in $R_1, \ldots, R_m$ according to v can be computed by applying the following query to $DV(v)$:

$$TQ_{t,v} = Split_{\mathbf{R_1,\ldots,R_m}}(\sigma_{G=t.G}(DV(v)))$$

Given tuple set $T \subseteq V$, T's derivation tracing query using $DV(v)$ is:

$$TQ_{T,v} = Split_{\mathbf{R_1,\ldots,R_m}}(DV(v) \ltimes T). \qquad \Box$$

Example 9.4 (Derivation View for `AllClothing`) The view `AllClothing` in Example 9.1 is defined on base tables `store`, `item`, and `sales`. Suppose these base tables are located in remote sources that we cannot or do not wish to access. In order to trace tuple derivations for `AllClothing`, we maintain a derivation view `DV_AllClothing`. Figure 29 shows the derivation view definition, and Figure 30 shows its contents. Then, computing derivations for tuples in view `AllClothing` only requires accessing view `DV_AllClothing`, and not the source tables. $\Box$

Using previously devised techniques for data warehousing [ZGMW96, ZWGM97], the auxiliary intermediate views and derivation views can be maintained consistently with each other, and with other views in the warehouse. Note that in cases of warehousing environments where the sources are inaccessible, the auxiliary views themselves need to be made *self-maintainable*. Previously devised techniques can be used here as well [GJM96, QGMW96].

store_id	store_name	city	state	item_id	item_name	category	price	num_sold
001	Target	Palo Alto	CA	0004	pants	clothing	30	600
002	Target	Albany	NY	0004	pants	clothing	35	800
003	Macy's	San Francisco	CA	0003	shirt	clothing	45	1500
003	Macy's	San Francisco	CA	0004	pants	clothing	60	600
004	Macy's	New York City	NY	0003	shirt	clothing	50	2100
004	Macy's	New York City	NY	0004	pants	clothing	70	1200

Figure 30: `DV_AllClothing` table

There are alternative derivation views to the one proposed here that trade tracing query cost for storage or maintenance cost. One simple option is to split the derivation view into separate tables that contain the base tuples of each source relation that contribute to the view. This scheme may reduce the storage requirement, but tracing queries must then recompute the join. Of course if accessing the sources is cheap and reliable, then it may be preferable to query the sources directly. However, a *compensation log* [HZ96] may be needed to keep the tracing result consistent with the warehouse views. Determining whether it is better to materialize the necessary information for derivation tracing or to query the sources and recompute information at tracing time in a given setting (based on query cost, update cost, storage constraints, source availability, etc.) is an interesting question left open for future work, and is closely related to results in [Gup97, LQA97].

9.3 A System Supporting Derivation Tracing

Figure 31 illustrates an overall warehouse structure for our example that supports tuple derivation tracing with materialized intermediate aggregation results and derivation views. The query tree on the left side of Figure 31 is the original definition of view `Clothing`. In order to trace `Clothing`'s tuple derivations, an auxiliary view `AllClothing` is maintained to record the intermediate aggregation results (as discussed in Section 9.1). Furthermore, to trace tuples in `AllClothing`, derivation view `DV_AllClothing` is maintained (as discussed in Section 9.2). The final set of materialized views is:

$$\begin{aligned}
\texttt{Clothing} &= \pi_{\texttt{total}}(\sigma_{\texttt{total}>5000}(\texttt{AllClothing})) \\
\texttt{AllClothing} &= \alpha_{\texttt{store_name, sum(num_sold) as total}}(\texttt{DV_AllClothing}) \\
\texttt{DV_AllClothing} &= \sigma_{\texttt{category="clothing"}}(\texttt{store} \bowtie \texttt{item} \bowtie \texttt{sales})
\end{aligned}$$

Each view can be computed and maintained based on the views (or base tables) directly beneath it using warehouse view maintenance techniques [QGMW96, ZWGM97]. Bold arrows on the right side of Figure 31 show the query and answer data flow. Ordinary view queries are sent to the view `Clothing`, while derivation queries are sent to the *Derivation Tracer* module. The tracer takes a request for the derivation of a tuple t in `Clothing` and queries auxiliary view `AllClothing` for t's derivation T_1 in `AllClothing` as specified in Theorem 5.3. The tracer then queries `DV_AllClothing` for the derivation T_2 of T_1 in D as specified in Theorem 9.3. T_2 is t's derivation in D.

10 Related Work Revisited

In this section, we revisit top-down Datalog query processing and the view update problem, and examine the differences between those problems and ours. We also show how our lineage tracing

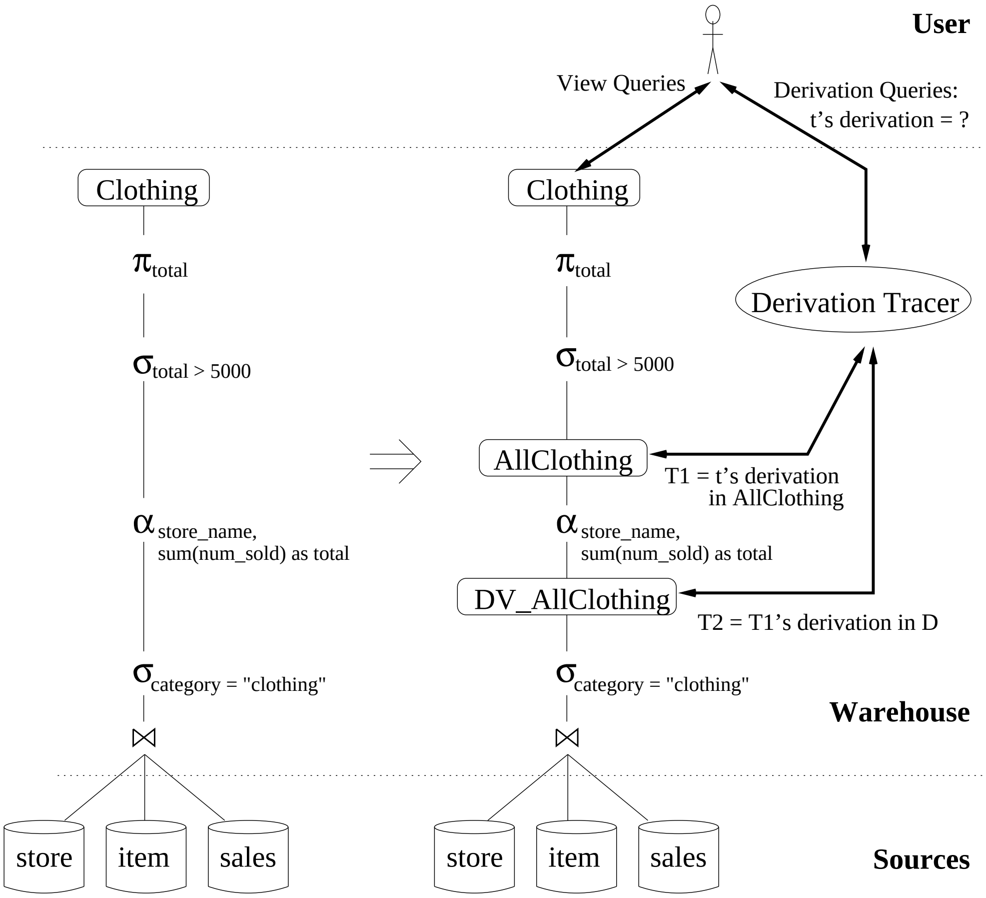


Figure 31: Derivation tracing in the warehouse system

algorithms can be applied to facilitate solutions to the view update problem.

10.1 Top-down Datalog Query Processing

In Datalog, relations are represented as *predicates*, tuples are *atoms* (or *facts*), and queries or views are represented by logical *rules*. Each rule contains a *head* (or *goal*) and a *body* with some *subgoals* that can (possibly recursively) derive the head [Ull89].

There are two modes of reasoning in Datalog: the bottom-up (or forward-chaining) mode and the top-down (or backward-chaining) mode. The top-down mode proves a goal by constructing a *rule-goal graph* with the goal as the top node, scanning the graph top-down, and recursively applying *rule-goal unification* and atom matching until finding an instantiation of all of the subgoals in the base data. Backtracking is used if a dead-end is met in the searching (proving) process.

Top-down evaluation of a Datalog goal thus provides information about the facts in the base data that yield the goal; in other words, it provides the lineage of the goal tuple. Our approach to tracing tuple lineage is obviously different from Datalog top-down processing. Instead of performing rule-goal unification and atom matching one tuple at a time, we generate a single query to retrieve all lineage tuples of a given tuple (or tuple set) in an SPJ or one-level ASPJ view. Our approach is better suited to tracing query optimization (as described in Section 5.3). Also, we support lineage tracing for aggregation views and bag semantics, which are not handled in Datalog. We do not handle recursion in this paper, although we believe our approach can be extended to recursive views while maintaining efficiency.

10.2 View Update Problem

The well-known view update problem is to transform updates on views into updates against the base tables on which the view is defined, so that the new base tables will continue to derive the updated view. The problem was first formulated in [Sto75] and [DB78], and solutions were based generally on the idea of view data lineage. In fact, [Sto75] provides a view update algorithm that is similar to our algorithm for tracing the lineage of SPJ views. However, neither paper formally defines the notion of data lineage, nor does it consider complex view scenarios (e.g., views with aggregation) or the warehousing environment.

Our derivation tracing algorithm for ASPJ views can guide the view update process to find an appropriate update translation for views with aggregation in many cases. For deletions (or modifications), our derivation tracing algorithms can directly identify an appropriate set of base relation tuples to delete (or modify), as shown in the following example.

Example 10.1 (View Update: Deletion) In Example 4.6 (Figure 11), we illustrated the derivation for $\langle 2, 8\rangle$ in view $V = \alpha_{X,sum(Y)}(\sigma_{Y\neq 0}(R))$. When the view update command "`delete` $\langle 2, 8\rangle$ `from` V" is issued, we can use the tuple derivation to determine that $\langle 2, 3\rangle$ and $\langle 2, 5\rangle$ should be deleted, and these should be the only changes. The updated base table will be $R = \{\langle 1, 2\rangle, \langle 1, 4\rangle, \langle 2, 0\rangle\}$, which derives the updated view $\{\langle 1, 6\rangle\}$. Note that without using tuple derivation tracing, a more naive algorithm might choose to delete $\langle 2, 0\rangle$ also, which maintains "correctness" but deletes more than necessary. $\square$

For insertions, the problem is harder, since the view tuple being inserted as well as its derivation do not currently exist. Our derivation tracing algorithms can be adapted to identify some components of the possible derivations of a view tuple being inserted, thereby guiding the base tuple insertions. Any attribute that is not projected into the view must be guessed using extra semantics, such as user instructions or base table constraints, or left null. Even here, derivation tracing can help the "guessing" process in certain cases, as shown in the following example.

Example 10.2 (View Update: Insertion) Suppose view update "`insert` $\langle 3, 2\rangle$ `into` V" is issued to the view in Example 4.6 (Figure 11). Since $\langle 3, 2\rangle$ is not in view V, we cannot ask for its current derivation. We only know that after we update R, R should produce a new view with $\langle 3, 2\rangle$ in it. According to the tracing query for V (as specified in Theorem 6.2), we can guess that after the update, the derivation R^* of $\langle 3, 2\rangle$ must satisfy the condition: $\forall t \in R^*$: $t.X = 3 \ \wedge \ t.Y \neq 0$. Assuming a constraint that $R.Y$ attributes are positive integers, and considering the requirement $sum(R^*.Y) = 2$, we can also assert that $\forall t \in R^*$: $t.Y \leq 2$. By further assuming a constraint that R has no duplicates, we can assert that $\forall t_1, t_2 \in R^*$: $t_1.Y \neq t_2.Y$. Putting all these assertions together, the only potential derivation of $\langle 3, 2\rangle$ is $\langle 3, 2\rangle$, so an appropriate base table update is to insert $\langle 3, 2\rangle$ into R. $\square$

Notice that the inserted tuple in Example 10.2 was carefully chosen. If $\langle 3, 8\rangle$ were inserted instead, we would have to randomly pick a translation from the reasonable ones or ask the user to choose. Even in this case, lineage tracing techniques together with base table constraints can be very useful in reducing the number of possible translations.

11 Conclusions and Future Work

We formulated the *view data lineage* problem and presented lineage tracing algorithms for relational views with aggregation using both set and bag semantics. Our algorithms identify the exact set of base data that produced a given view data item. We also presented techniques for efficient and consistent lineage tracing in a multi-source warehousing system. Our results can form the basis of a tool by which an analyst can browse warehouse data, then "drill-through" to the source data that produced certain warehouse data of interest. The basic data lineage problem and solutions presented in this paper lead to the following additional research issues.

- Tuple derivations as defined in this paper explain how certain base relation tuples cause certain view tuples to exist. As such, derivation tracing is a useful technique for investigating the origins of potentially erroneous view data. However, in some cases a view tuple may be erroneous not (only) because the base tuples that derive it are erroneous, but because base relation tuples that should appear in the derivation are missing. For example, a base tuple may contribute to the wrong group in an aggregate view because its grouping value is incorrect. We plan to explore how this "missing derivation data" problem can be addressed in our lineage framework.
- In Sections 9.1 and 9.2 we discussed trade-offs associated with materializing versus recomputing intermediate and derivation views, and we mentioned briefly self-maintainability of auxiliary views. We are in the process of conducting a comparative performance study of the various options.
- We believe that lineage tracing can be used to help solve related problems such as view schema evolution and view update. Some initial ideas for view update were presented in Section 10.2.
- In some situations, an analyst may wish to see not only the base data that derive a given view data item, but also a representation of the process by which the view data item was derived. Appropriate user interfaces need to be explored.

In summary, data lineage is a rich problem with many interesting applications. In this paper we provide an initial practical solution for lineage tracing in data warehouses. We are implementing a lineage tracing package within the *WHIPS* data warehousing prototype at Stanford [WGL+96], and we plan to extend our work in the many directions outlined above.

Acknowledgements

We are grateful to Sudarshan Chawathe, Himanshu Gupta, Jeff Ullman, Vasilis Vassalos, Yue Zhuge, and all of our WHIPS group colleagues for helpful and enlightening discussions.

References

[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.

[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.

[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.

[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.

[GJM96] A. Gupta, H. Jagadish, and I. S. Mumick. Data integration using self-maintainable views. In *Proc. of Fifth International Conference on Extending Database Technology*, pages 140–144, Avignon, France, March 1996.

[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.

[HCC98] J. Han, S. Chee, and J.Y. Chiang. Issues for on-line analytical mining of data warehouses. In *Proc. of the Workshop on Research Issues on Data Mining and Knowledge Discovery*, Seattle, Washington, June 1998.

[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.

[HZ96] R. Hull and G. Zhou. A framework for supporting data integration using the materialized and virtual approaches. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 481–492, Montreal, Canada, June 1996.

[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.

[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.

[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.

[Ull89] J. D. Ullman. *Database and Knowledge-base Systems (Vol 2).* Computer Science Press, 1989.

[WGL+96] J.L. Wiener, H. Gupta, W.J. Labio, Y. Zhuge, H. Garcia-Molina, and J. Widom. A system prototype for warehouse view maintenance. In *Proc. of the Workshop on Materialized Views: Techniques and Applications*, pages 26–33, Montreal, Canada, June 1996.

[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.

[ZGMW96] Y. Zhuge, H. Garcia-Molina, and J. L. Wiener. The Strobe algorithms for multi-source warehouse consistency. In *Proc. of the Fourth Conference on Parallel and Distributed Information Systems*, pages 146–157, Miami Beach, Florida, December 1996.

[ZWGM97] Y. Zhuge, J. L. Wiener, and H. Garcia-Molina. Multiple view consistency for data warehousing. In *Proc. of the Thirteenth International Conference on Data Engineering*, pages 289–300, Birmingham, UK, April 1997.

A Proofs

A.1 Proof of Theorem 4.2

Theorem 4.2 (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. $\square$

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 aggregate operator, we know that T_1' contains exactly all tuples in T_1 that have the same groupby attributes as t. So does T_1''. Therefore, $T_1' = T_1''$.

For SPJ operator Op, we first prove that $D^* = \langle T_1^*, \ldots, T_m^* \rangle = D' \cup D''$ also satisfies requirements (a) $Op(T_1^*, \ldots, T_m^*) = \{t\}$; (b) $\forall i = 1..m$: $\forall t^* \in T_i^*$: $Op(T_1^*, \ldots, \{t^*\}, \ldots, T_m^*) \neq \varnothing$ in Definition 4.1.

If Op is σ, π, or $\cup$.

(a) Since D' and D'' satisfy Definition 4.1(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 Definition 4.1(b), $Op(T_1^*, \{t^*\}, T_m^*) \supseteq Op(T_1', \{t^*\}, T_m') \neq \varnothing$

If Op is $\bowtie$:

(a) $\langle T_1', \ldots, T_m' \rangle$ satisfies Definition 4.1(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 $t' \in T_i'$ since $\langle T_1', \ldots, T_m' \rangle$ also satisfies Definition 4.1(b)
$\Rightarrow$ $\varnothing \subsetneq Op(T_1', \ldots, \{t'\}, \ldots, T_m') \subseteq Op(T_1', \ldots, T_i', \ldots, T_m') = \{t\}$
$\Rightarrow$ $Op(T_1', \ldots, \{t'\}, \ldots, T_m') = \{t\}$
$\Rightarrow$ $t' = t.\mathbf{T_i}$
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 Definition 4.1(b), $Op(T_1', \ldots, \{t^*\}, \ldots, T_m') \neq \varnothing$
$\Rightarrow$ According to monotonicity of Op, $Op(T_1^*, \ldots, \{t^*\}, \ldots, T_m^*) \supseteq Op(T_1', \ldots, \{t^*\}, \ldots, T_m')$
$\Rightarrow$ $Op(T_1^*, \ldots, \{t^*\}, \ldots, T_m^*) \neq \varnothing$

If Op is $-$:

(a) $\langle T_1', T_2' \rangle$ satisfies Definition 4.1(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 Definition 4.1(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, $\forall t^* \in T_2^*$: $t^* \neq t \Rightarrow T_1^* - \{t^*\} = \{t\} \neq \varnothing$

Above, we have shown that $D^* = D' \cup D''$ also satisfies requirements (a) and (b) in

Definition 4.1. According to the maximality of tuple derivation, 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. $\Box$

A.2 Proof of Theorem 4.4

Theorem 4.4 (Tuple Derivations for Operators) Let $T, T_1, \ldots, T_m$ be tables. Recall that $\mathbf{T_i}$ denotes the schema of T_i.

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

Proof: To prove $Op^{-1}{}_{\langle T_1,\ldots,T_m\rangle}(t) = \langle {T_1}', \ldots, {T_m}'\rangle$, according to Definition 4.1 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: $\langle {T_1}'', \ldots, {T_m}''\rangle \subseteq \langle {T_1}', \ldots, {T_m}'\rangle$.

${\sigma_C}^{-1}{}_{\langle T\rangle}(t) = \langle\{t\}\rangle$, for $t \in \sigma_C(T)$.

Let $T' = \{t\}$.

1. $t \in \sigma_C(T) \Rightarrow t \in T \Rightarrow T' = \{t\} \subseteq T$
2. $t \in \sigma_C(T) \Rightarrow t$ satisfies $C \Rightarrow \sigma_C(T') = \{t\}$
3. Consider an arbitrary $t' \in T'$, $t' = t$. Therefore, $\sigma_C(\{t'\}) = \{t\} \neq \varnothing$
4. Consider an arbitrary T'' that satisfies 1, 2, 3.

$\Rightarrow$ $\forall t'' \in T''$: $\sigma_C(\{t''\}) \neq \varnothing$ and $\sigma_C(T'') = \{t\}$
$\Rightarrow$ $\forall t'' \in T''$: $\sigma_C(\{t''\}) = \{t''\} = \{t\}$
$\Rightarrow$ $\forall t'' \in T''$: $t'' = t \in T'$
$\Rightarrow$ $T'' \subseteq T'$ $\Box$

${\pi_A}^{-1}{}_{\langle T\rangle}(t) = \langle\sigma_{A=t}(T)\rangle$, for $t \in \pi_A(T)$.

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

1. $T' \subseteq T$ according to the definition of σ
2. $\forall t' \in T'$: $t'.A = t$. Therefore, $\pi_A(T') = \{t\}$
3. Consider an arbitrary $t' \in T'$, $t'.A = t$. Therefore, $\pi_A(\{t'\}) = \{t\} \neq \varnothing$

4. Consider an arbitrary T'' that satisfies 1, 2, 3.
   $\pi_A(T'') = \{t\}$
$\Rightarrow$ $\forall t'' \in T''$: $t''.A = t$
$\Rightarrow$ $\forall t'' \in T''$: $t'' \in T'$
$\Rightarrow$ $T'' \subseteq T'$ $\Box$

$\bowtie^{-1}{}_{\langle T_1,\ldots,T_m\rangle}(t) = \langle\{t.\mathbf{T_1}\},\ldots,\{t.\mathbf{T_m}\}\rangle$, 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.
$\Rightarrow$ $T_1{}'' \bowtie \cdots \bowtie T_m{}'' = \{t\}$ and $\forall t'' \in T_i''$: $T_1{}'' \bowtie \ldots \{t''\} \bowtie T_m{}'' \neq \varnothing$
$\Rightarrow$ $\forall t'' \in T_i''$: $T_1{}' \bowtie \ldots \{t''\} \bowtie T_m{}' = \{t\}$
$\Rightarrow$ $\forall t'' \in T_i''$: $t'' = t.\mathbf{T_i} \in T_i{}'$
$\Rightarrow$ $T_i{}'' \subseteq T_i{}'$ $\Box$

$\alpha_{G,aggr(B)}{}^{-1}{}_{\langle T\rangle}(t) = \langle\sigma_{G=t.G}(T)\rangle$, 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.
   $\alpha_{G,aggr(B)}(T'') = \{t\}$
$\Rightarrow$ $\forall t'' \in T''$: $t''.G = t.G$
$\Rightarrow$ $\forall t'' \in T''$: $t'' \in T'$
$\Rightarrow$ $T'' \subseteq T'$ $\Box$

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

Let $T_i' = \sigma_{\mathbf{T_i}=t}(T_i)$.

1. $T_i' \subseteq T_i$ according to the definition of σ
2. $T_1 \cup \cdots \cup T_m = \{t\}$ according to the definition of $\cup$
3. Consider an arbitrary $t' \in T_i'$, $T_1' \cup \ldots \{t'\} \cdots \cup T_m' \subseteq \{t'\} \neq \varnothing$

4. Consider an arbitrary $\langle T_1'', \ldots, T_m'' \rangle$ that satisfies 1, 2, 3.
$\Rightarrow$ $T_1'' \cup \cdots \cup T_m'' = \{t\}$
$\Rightarrow$ $\forall t'' \in T_i''$: $t'' = t \in T_i'$
$\Rightarrow$ $T_i'' \subseteq T_i'$ $\square$

$-^{-1}{}_{\langle T_1,T_2 \rangle}(t) = \langle \{t\}, T_2 \rangle$**, 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. $t \in T_1 - T_2$
$\Rightarrow$ $t \notin T_2$
$\Rightarrow$ $T_1' - T_2' = \{t\} - T_2 = \{t\}$
3. Consider an arbitrary $t' \in T_1'$, $t' = t$
$\Rightarrow$ $\{t'\} - T_2' = \{t\} \neq \varnothing$
Consider an arbitrary $t' \in T_2'$, $t' \neq t$
$\Rightarrow$ $T_1' - \{t'\} = \{t\} \neq \varnothing$
4. Consider an arbitrary $\langle T_1'', T_2'' \rangle$ that satisfies 1, 2, 3.
$\Rightarrow$ $T_1'' - T_2'' = \{t\}$ and $\forall t'' \in T_1''$: $\{t''\} - T_2'' \neq \varnothing$
$\Rightarrow$ $\forall t'' \in T_1''$: $t'' = t$. Otherwise, if $t'' \notin T_2''$ then $T_1'' - T_2'' = \{t\}$; if $t'' \in T_2''$ then $\{t''\} - T_2'' = \varnothing$
$\Rightarrow$ $T_1'' \subseteq T_1'$
Also, $T_2'' \subseteq T_2' = T_2$ $\square$

A.3 Proof of Theorem 4.7

Theorem 4.7 (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. $\square$

Lemma A.1 (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 $v \Longleftrightarrow$
$\exists j \in 1..k, t' \in V_j$ such that t^* contributes to t' according to v_j and t' contributes to t according to v'.

Proof of Lemma A.1: We use induction on the height of the query tree for v', denoted $h(v')$, where the v_i's are the leaves.

Base: For $h(v') = 0$, $v = v_1$. Lemma A.1 holds trivially.
Induction hypothesis: Suppose for $0 \leq h(v') \leq n-1$, Lemma A.1 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 4.5, $\exists p \in 1..q, \exists t'' \in V_p{}'$ 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 4.5, 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 4.5, $\exists p \in 1..q, \exists t'' \in V_p{}'$ 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 4.1, t^* contributes to t.

By induction, Lemma A.1 holds for v' of height n, and therefore for any v' and v. $\Box$

Proof of Theorem 4.7: Now, we prove Theorem 4.7 using Lemma A.1. We need to prove that $\forall i = 1..m$: all tuples in the R_i component[6] of $v^{-1}{}_D(t)$ are also in the R_i component of $\bigodot_{j=1..k} v_j{}^{-1}{}_D(V_j^*)$, and vise versa.

1. Consider an arbitrary R_i and $t^* \in v^{-1}{}_{R_i}(t)$
$\Rightarrow$ According to Definition 4.5, t^* contributes to t
$\Rightarrow$ According to Lemma A.1, $\exists j \in 1..k, t' \in V_j$ such that
t' contributes to t according to v', and t^* contributes to t' according to v_j
$\Rightarrow$ $t' \in V_j^*$ and $t^* \in v_j{}^{-1}{}_{R_i}(t')$
$\Rightarrow$ $t^* \in v_j{}^{-1}{}_{R_i}(V_j^*)$, 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 $\bigodot_{j=1..k} v_j{}^{-1}{}_D(V_j^*)$
$\Rightarrow$ $\exists j \in 1..k$, such that v_j is defined over R_i
$\Rightarrow$ $t^* \in v_j{}^{-1}{}_{R_i}(V_j^*)$
$\Rightarrow$ $\exists t' \in V_j^*$, such that $t^* \in v_j{}^{-1}{}_{R_i}(t')$
$\Rightarrow$ t' contributes to t according to v', and t^* contributes to t' according to v_j
$\Rightarrow$ According to Lemma A.1, t^* contributes to t
$\Rightarrow$ According to Definition 4.5, $t^* \in v^{-1}{}_{R_i}(t)$ $\Box$

A.4 Proof of Theorem 4.8

Theorem 4.8 (Derivation Equivalence after SPJ Transformation) Tuple derivations of equivalent SPJ views are equivalent. That is, given two equivalent SPJ views v_1 and v_2, $\forall D$: $\forall t \in v_1(D) = v_2(D)$: $v_1^{-1}{}_D(t) = v_2^{-1}{}_D(t)$. $\Box$

To prove this theorem, we first give a new definition of tuple derivation for SPJ views, which is equivalent to Definition 4.5 when only SPJ views are considered. We then prove Theorem 4.8

[6] 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 $v^{-1}{}_{R_i}(t)$.

based on the new definition.

Definition A.2 (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 $\exists R'_j \subseteq R_j$ for $j = 1..m$, such that

(a) $t \in v(R'_1, \ldots, R'_{i-1}, \{t_i\}, R'_{i+1}, \ldots, R'_m)$

(b) $t \notin v(R'_1, \ldots, R'_{i-1}, \varnothing, R'_{i+1}, \ldots, R'_m)$

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. $\square$

Definition A.2 says that a base tuple $t^* \in R_i$ contributes to t in the view if there exist some subsets of base tables $\langle R'_1, \ldots, R'_m \rangle$, such that they produce t with $R'_i = \{t^*\}$, and cannot produce t if $R'_i = \varnothing$. We first prove that Definition A.2 is equivalent to Definition 4.5 for SPJ views.

Lemma A.3 (Definition Equivalence) Definition A.2 is equivalent to Definition 4.5 for SPJ views. In other words, the derivation of t defined by Definition 4.5 is the same as that defined by Definition A.2 for every SPJ view and every traced tuple. $\square$

Proof of Lemma A.3: 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 4.5 and Definition A.2.

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

$\Rightarrow$ According to Definition 4.5 and the monotonicity of V, $\varnothing \subsetneq v(\{t^*\}) \subseteq v(R^*) = \{t\}$

$\Rightarrow$ $v(\{t^*\}) = \{t\}$

$\Rightarrow$ $v(\varnothing) = \varnothing$ and $v(\{t^*\}) = \{t\}$

$\Rightarrow$ t^* contributes to t according to Definition A.2

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

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

$\Rightarrow$ According to Definition A.2, $v(\varnothing) = \varnothing$ and $v(\{t^*\}) = \{t\}$

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

$\Rightarrow$ $R^* \cup \{t^*\}$ also satisfies requirements (a) and (b) in Definition 4.5

$\Rightarrow$ 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 4.5 and Definition A.2.

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

$\Rightarrow$ According to Definition 4.5 and the monotonicity of V, $\varnothing \subsetneq \{t^*\} \bowtie S^* \subseteq R^* \bowtie S^* = \{t\}$

$\Rightarrow$ $\{t^*\} \bowtie S^* = \{t\}$ and $\varnothing \bowtie S^* = \varnothing$

$\Rightarrow$ t^* contributes to t according to Definition A.2

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

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

$\Rightarrow$ According to Definition A.2, $\exists S' \subseteq S$ such that $\varnothing \bowtie S' = \varnothing$ and $\{t^*\} \bowtie S' = \{t\}$

$\Rightarrow$ $\exists t' \in S'$ 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 requirements (a) and (b) in Definition 4.5.

$\Rightarrow$ 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 4.7 that tuple derivation as defined by Definition 4.5 is transitive. We can also prove this property for derivation as defined by Definition A.2. After proving that Definition 4.5 and Definition A.2 are equivalent for views with a single SPJ operator, due to the transitive property of tuple derivation, we can easily prove that Definition 4.5 and Definition A.2 are equivalent for any SPJ view by induction. □

Proof of Theorem 4.8: Having proved that Definition A.2 is equivalent to Definition 4.5 for SPJ views, we now prove Theorem 4.8 based on Definition A.2 (instead of Definition 4.5, which is a more general definition originally used in the paper).

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 A.2, there exists subsets of the base tables $\langle R'_1, \ldots, R'_m\rangle$, such that they produce t with $R'_i = \{t^*\}$, and cannot produce t if $R'_i = \varnothing$, according to v_1. Since these subsets produce exactly the same result using v_2, we know that t^* must also contribute to t according to v_2. 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, ${v_1^{-1}}_D(t) = {v_2^{-1}}_D(t)$. □

A.5 Proof of Theorem 5.3

Theorem 5.3 (Derivation 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:

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

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

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

where $\ltimes$ is the relational semijoin operator. □

Proof: We need to prove:

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

Let:

$$V_2 = R_1 \bowtie \cdots \bowtie R_m$$
$$V_1 = \sigma_C(V_2)$$
$$V = \pi_A(V_1)$$

According to Theorem 4.7, we have:

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

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

A.6 Proof of Theorem 5.5

Theorem 5.5 (Derivation Tracing using Key Information) Let R_i be a base table with key attributes K_i, $i = 1..m$, and let view $V = \pi_A(\sigma_C(R_1 \bowtie \cdots \bowtie R_m))$ include all base table keys (i.e., $K_i \in A$, $i = 1..m$). View tuple t's derivation is $\langle \sigma_{K_1=t.K_1}(R_1), \ldots, \sigma_{K_m=t.K_m}(R_m)\rangle$. $\square$

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 5.3, we know that

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

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 $i = 1..m$. $\square$

A.7 Proof of Theorem 6.2

Theorem 6.2 (Derivation Tracing Query for a One-Level ASPJ View) Given a 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 given tuple $t \in V$, t's derivation in $R_1, \ldots, R_m$ according to v can be computed by applying the following query to the base tables:

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

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

$$TQ_{T,v} = Split_{\mathbf{R_1},\ldots,\mathbf{R_m}}(\sigma_C(R_1 \bowtie \cdots \bowtie R_m) \ltimes T) \qquad \square$$

Proof: We need to prove:

$$\begin{aligned} v^{-1}{}_D(t) &= Split_{\mathbf{T_1},\ldots,\mathbf{T_m}}(\sigma_{G=t.G \wedge C}(T_1 \bowtie \ldots \bowtie T_m)) \\ v^{-1}{}_D(T) &= Split_{\mathbf{T_1},\ldots,\mathbf{T_m}}(\sigma_C(T_1 \bowtie \ldots \bowtie T_m) \ltimes T) \end{aligned}$$

Let:

$$\begin{aligned} V_1 &= \pi_A(\sigma_C(T_1 \bowtie \cdots \bowtie T_m)) \\ V &= \alpha_{G,aggr(B)}(V_1) \end{aligned}$$

According to Theorem 4.7 and Theorem 5.3, we have:

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

$v^{-1}{}_D(T)$ can be computed from $v^{-1}{}_D(t)$ similar as in Section A.5. $\square$

A.8 Proof of Theorem 7.2

Theorem 7.2 (Extended Canonical Form) Procedure `Canonicalize` in Figure 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. $\square$

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: $- \rightarrow \cup$, $\alpha \rightarrow \pi$, $\cup \rightarrow \pi$, $\pi \rightarrow \pi$, 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 learn from [Ull89] 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 prove the derivation equivalence after applying `Canonicalize`. Alternatively, we can prove equivalent SPJ views with set unions have equivalent derivations. We have proven that two equivalent SPJ views have equivalent derivations in Appendix A.4. We can easily extend the results to views with set unions. □

A.9 Proof of Theorem 7.4

Theorem 7.4 (Tracing Queries 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:

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

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

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

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:

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

Proof: We first prove:

$$v^{-1}{}_D(t) = \bigodot_{j=1..k} Split_{\mathbf{T_{j,1}},\ldots,\mathbf{T_{j,l_j}}}(\sigma_{G=t.G \wedge C}(T_{j,1} \bowtie \ldots \bowtie T_{j,l_j}))$$

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

$$v^{-1}{}_D(\mathbf{ALL}) = \bigodot_{j=1..k} Split_{\mathbf{T_{j,1}},\ldots,\mathbf{T_{j,l_j}}}(\sigma_C(T_{j,1} \bowtie \ldots \bowtie T_{j,l_j}))$$

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 4.7, Proof A.7, 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 $v^{-1}{}_D(t)$ similar as in Section A.5. We now prove $v^{-1}{}_D(V)$ based on $v^{-1}{}_D(T)$.

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

A.10 Proof of Theorem 8.5

Theorem 8.5 (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 $v^{-1}{}_D(t) = v^{-P}{}_D(t)$. $\square$

Proof: According to Definition 8.1, 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 derive a tuple with value t in the view table. These derived tuples cannot be removed by other base tuples due to the view 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, $v^{-P}{}_D(t) = v^{-1}{}_D(t)$. $\square$

A.11 Proof of Theorem 8.6

Theorem 8.6 (Derivation Pool of a Tuple Set with Tuples with the Same Value) 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 $v^{-P}{}_D(\sigma_{\mathbf{V}=t}(v(D))) = v^{-P}{}_D(t)$. $\square$

Proof: $\forall t^* \in v^{-1}{}_D(T)$: $\exists t' \in T$ such that t^* contributes to t'. Because T contains only tuples with value t, $t' = t$. So, t^* contributes to t, and $t^* \in v^{-1}{}_D(t)$. Therefore, $v^{-1}{}_D(T) \subseteq v^{-1}{}_D(t)$. Also, since $t \in T$, $v^{-1}{}_D(T) \supseteq v^{-1}{}_D(t)$. Therefore, $v^{-1}{}_D(T) = v^{-1}{}_D(t)$. $\square$

A.12 Proof of Theorem 8.7

Theorem 8.7 (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 $v' = \sigma_C(v)$. $\square$

Proof: Let $v' = \sigma_C(v)$,

$$\begin{aligned} v'^{-1}{}_D(v'(D)) &= v^{-1}{}_D(\sigma_C{}^{-1}{}_{\langle V \rangle}(\sigma_C(v(D))) \\ &= v^{-1}{}_D(V \ltimes \sigma_C(v(D))) \\ &= v^{-1}{}_D(\sigma_C(v(D))) \qquad \square \end{aligned}$$

A.13 Proof of Theorem 8.8

Theorem 8.8 (Derivation Pools for Operators with Bag Semantics) Let T, $T_1, \ldots, T_m$ be tables. Recall that $\mathbf{T_i}$ denotes the schema of T_i.

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

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 4.4 as proved in Proof A.2. 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 $i = 1..m$. $\square$

$\dot{-}^{-1}{}_{\langle T_1,T_2 \rangle}(t) = \langle \sigma_{\mathbf{T_1}=t}(T_1), \sigma_{\mathbf{T_2}\neq t}(T_2) \rangle$**, 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 4.4 as proved in Proof A.2. 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 $t \in T_1 - T_2$. $\square$

A.14 Proof of Theorem 8.10

Theorem 8.10 (Derivation Tracing Query for an SPJ View) Let D be a database with base tables $R_1, \ldots, R_m$, and let $V = v(D) = \pi_A(\sigma_C(R_1 \bowtie \ldots \bowtie R_m))$ be an SPJ view over D. Given tuple $t \in V$, t's derivation pool in D according to v can be computed by applying the following query to the base tables:

$$TQ_{t,v} = TSplit_{R_1,\ldots,R_m}(\sigma_{C \wedge A=t}(R_1 \bowtie \ldots \bowtie R_m))$$

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

$$TQ_{T,v} = TSplit_{R_1,\ldots,R_m}(\sigma_C(R_1 \bowtie \ldots \bowtie R_m) \ltimes T) \qquad \square$$

Proof: We need to prove:

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

From Theorem 4.4 with bag semantics, we know that

$$\begin{aligned} \pi_A{}^{-1}{}_{\langle T_1\rangle}(T) &= \langle T_1 \ltimes T\rangle, \text{ for } T \subseteq \pi_A(T_1) \\ \sigma_C{}^{-1}{}_{\langle T_1\rangle}(T) &= \langle T_1 \ltimes T\rangle, \text{ for } T \subseteq \sigma_C(T_1) \\ \bowtie^{-1}{}_{\langle T_1,\ldots,T_m\rangle}(T) &= \langle T_1 \ltimes T, \ldots, T_m \ltimes T\rangle, \text{ for } T \subseteq T_1 \bowtie \cdots \bowtie T_m \\ &= TSplit_{T_1,\ldots,T_m}(T) \end{aligned}$$

Let:

$$\begin{aligned} V_2 &= R_1 \bowtie \cdots \bowtie R_m \\ V_1 &= \sigma_C(V_2) \\ V &= \pi_A(V_1) \end{aligned}$$

According to Theorem 4.7, we have:

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

$$\begin{aligned} v^{-1}{}_D(t) &= v_1{}^{-1}{}_D(\pi_A{}^{-1}{}_{V_1}(T)) \\ &= v_1{}^{-1}{}_D(V_1 \ltimes T) \\ &= v_2{}^{-1}{}_D(\sigma_C{}^{-1}{}_{V_2}(V_1 \ltimes T)) \\ &= v_2{}^{-1}{}_D(V_2 \ltimes (V_1 \ltimes T)) \\ &= TSplit_{\mathbf{R_1},\ldots,\mathbf{R_m}}(V_1 \ltimes T) \\ &= TSplit_{\mathbf{R_1},\ldots,\mathbf{R_m}}(\sigma_C(R_1 \bowtie \ldots \bowtie R_m) \ltimes T) \qquad \square \end{aligned}$$

A.15 Proof of Theorem 8.11

Theorem 8.11 (Derivation Set for an SPJ View) Given an SPJ view $V = v(T_1, \ldots, T_m) = \pi_A(\sigma_C(T_1 \bowtie \cdots \bowtie T_m))$ and a tuple $t \in V$, t's derivation set according to v is

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

Proof: $\forall D^*$,

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

$\Leftrightarrow$ $D^* \in DS_v(t)$ $\Box$

A.16 Proof of Theorem 8.12

Theorem 8.12 (Extending SPJ View Definition Using 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 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 $v^{-S}{}_D(t) = \{v'^{-1}{}_D(t_j),\ j = 1..n\}$. $\Box$

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 8.1, 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 is conflicting to the fact that $v'(D^*)$ contains a single tuple.

According to Theorem 4.8, 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 4.7, $v'^{-1}{}_D(t_j)$ 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. $\Box$

A.17 Proof of Theorem 9.3

Theorem 9.3 (Derivation Tracing using the Derivation View) Let V be a one-level ASPJ view over base tables: $V = v(R_1, \ldots, R_m) = \alpha_{G,aggr(B)}(\pi_A(\sigma_C(R_1 \bowtie \cdots \bowtie R_m)))$. Let $DV(v)$ be v's derivation view as defined in Definition 9.2. Given a tuple $t \in V$, t's derivation in $R_1, \ldots, R_m$ according to v can be computed by applying the following query to $DV(v)$:

$$TQ_{t,v} = Split_{\mathbf{R_1},\ldots,\mathbf{R_m}}(\sigma_{G=t.G}(DV(v)))$$

Given tuple set $T \subseteq V$, T's derivation tracing query using $DV(v)$ is:

$$TQ_{T,v} = Split_{\mathbf{R_1},\ldots,\mathbf{R_m}}(DV(v) \ltimes T). \qquad \Box$$

Proof: The proof is obvious according to the definition of the derivation view (Definition 9.2) and tracing queries for one-level ASPJ views (Theorem 6.2). $\Box$