Paper Number U017

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 and develop lineage tracing algorithms for relational views with aggregation. Our algorithms are efficient and provide consistent lineage tracing in a multi-source data warehousing environment. Using our results, we can build a tool that allows analysts to browse warehouse data, select view tuples of interest, then "drill-down" to examine the source data that produced them.

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 [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 are the views themselves 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 needs facilities for data exploration at different levels. The ability to select a portion of relevant view data and "drill-down" 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.

*This work was supported by Rome Laboratories under Air Force Contract F30602-96-1-0312, and by the Advanced Research and Development Committee of the Community Management Staff as a project in the MDDS Program.

- 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.
- 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 8.2, 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 certain 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 focus on the lineage problem for relational Select-Project-Join views with aggregation (*ASPJ views*) in a data warehousing environment. Our results extend to additional relational operators as we show in [CWW98]. In summary, we:

- Formulate the view data lineage problem. We give a declarative definition of *tuple derivations* for relational operators, and inductively define view tuple derivations based on the tuple derivations for the operators (Section 4).
- Develop derivation tracing algorithms, including proofs of their correctness (Sections 5 and 6).
- Discuss issues of derivation tracing in a warehousing environment, and show how to trace tuple derivations for views defined on distributed, legacy sources consistently and efficiently (Section 7).

We first discuss related work in Section 2. We then motivate the lineage problem using detailed examples in Section 3. After defining the lineage problem and presenting our solutions as summarized above, in Section 8 we revisit some related issues (e.g., the view update problem) in detail. Conclusions and future work are covered in Section 9. All proofs are provided in the Appendix.

2 Related Work

There has been a great deal of work in view maintenance and related problems in data warehousing, but to the best of our knowledge the lineage problem has not been addressed. Overviews of research directions and results in data warehousing can be found in [CD97, Wid95, WB97]. [GM95] specifically covers view maintenance problems in data warehousing. Incremental view maintenance algorithms have been presented for relational algebra views [QW91], for aggregation [Qua96], and for recursive views [GMS93]. View "self-maintainability" issues are addressed in [QGMW96]. Warehouse view consistency is studied in [ZGMW96, ZWGM97], to ensure that views in the warehouse are consistent with each other and reflect consistent states of the sources. All of these papers consider computing warehouse views but do not address the reverse problem of view data lineage.

OLAP systems, usually sitting on top of a data warehouse, allow users to perform analysis and make predictions based on the warehouse view information [Col96, HCC98]. The *data cube* is a popular OLAP structure that facilitates multi-dimensional aggregation over source data [GBLP96]. Cube "rolling-up" and "drilling-down" enable the user to browse the view data at any level and any dimension of the aggregation [HRU96, MQM97]. However, data cubes are based on a restricted form of relational views, and usually only allow drilling down within the warehouse, not to the original data sources.

Metadata for warehouse views can be maintained to record lineage information about a particular view column [CM89, HQGW93]. However, this approach only provides schema-level lineage tracing, while many applications require lineage at a finer (instance-level) granularity. Some scientific databases use tuple-level annotations to keep track of lineage [HQGW93], which can introduce high storage overhead in warehousing applications.

[WS97] introduces "weak inversion" to compute fine-grained data lineage. However, not all views have an inverse or weak inverse. Also, the system in [WS97] requires users to provide a view's inverse function in order to compute lineage, which we feel is not always practical. Our algorithms trace tuple-level lineage automatically for the user and maintain the necessary auxiliary information to ensure view invertibility.

The problem of reconstructing base data from summary data is studied in [FJS97]. Their statistical approach estimates the base data using only the summary data and certain constraints; it does not guarantee accurate lineage tracing. In this paper, we focus on accurate lineage tracing with the base data available either in remote sources or stored locally in the warehouse.

The view update problem [DB78, Kel86] is to translate updates against a view into updates against the relevant base tables, so that the updated base tables will derive the updated view. View data lineage can be used to help solve the view update problem, employing a different approach from previous techniques; see Section 8.

Finally, Datalog can 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. However, we take an approach that is very different from rule-goal unification; a detailed comparison is presented in Section 8.

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

Figure 1: `item` table

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 2: `store` 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

3 Motivating Examples

In this section, we provide examples that motivate the definition of data lineage and how lineage tracing can be useful. Consider a data warehouse with retail store data over the following base tables:

```
item(item_id, item_name, category)
store(store_id, store_name, city, state)
sales(store_id, item_id, price, num_sold)
```

The `item` and `store` tables are self-explanatory. The `sales` table contains sales information, including the number and price of each product sold by each store. Example table contents are shown in Figures 1, 2, and 3.

Example 3.1 (Lineage of SPJ View) Suppose a sales department wants to study the selling patterns of California stores. A materialized view `Calif` can be defined in the data warehouse for the analysis. The SQL definition of the view is:

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

The view definition also can be expressed using the relational algebra tree in Figure 4. 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 `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. □

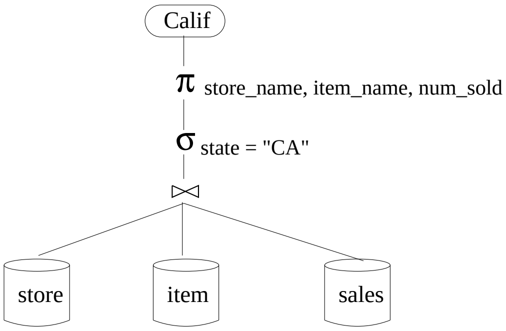


Figure 4: View definition for `Calif`

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

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

Example 3.2 (Lineage of Aggregation View) Now let's consider another warehouse view `Clothing`, for analyzing the total clothing sales of the large stores (which have sold more than 5000 clothing items).

```
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
```

The relational algebra definition of the view is shown in Figure 7. We extend 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>, as shown in Figure 8.

The analyst may wish to learn more about the origins of this tuple, and asks question Q2: "Which base data produced tuple <5400> in `Clothing`?" Not surprisingly, due to the more complex view definition, this question is more difficult to answer than Q1. We develop the appropriate algorithms in Section 6, and Figure 9 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 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, 6, and 7 then present algorithms to compute view tuple derivations.

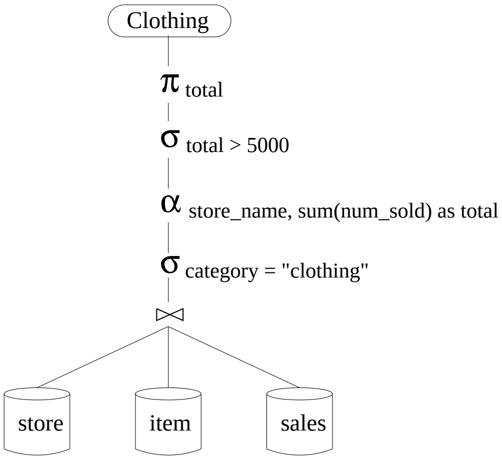


Figure 7: View definition for `Clothing`

total
5400

Figure 8: `Clothing` table

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 9: `Clothing` lineage for `<5400>`

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 introduces the views on which we focus in this paper. Tuple derivations for operators and views are then defined in Sections 4.2 and 4.3, respectively.

We assume that a table (or relation) R with schema $\mathbf{R}$ contains a set of tuples $\{t_1, \ldots, t_n\}$, with no duplicates. (Thus, we consider *set semantics* in this paper. We have adapted our work to *bag semantics* as well; please see [CWW98].) 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 $V = v(R_1, \ldots, R_m)$.

4.1 Views

We consider a class of views defined over base relations using the relational algebra operators *selection* (σ), *projection* (π), *join* ($\bowtie$), and *aggregation* (α). Our framework applies as well to *set union* ($\cup$), *set intersection* ($\cap$), and *set difference* ($-$), however these operators are omitted in this paper due to space constraints. Please see [CWW98] for details.

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*: $\pi_A(V_1) = \{t.A \mid t \in V_1\}$
- *Natural Join*: $V_1 \bowtie V_2 = \{t.[\mathbf{V_1} \cup \mathbf{V_2}] \mid \exists t_1 \in V_1,\ t_2 \in V_2 :$
  $t_1.[\mathbf{V_1} \cap \mathbf{V_2}] = t_2.[\mathbf{V_1} \cap \mathbf{V_2}] \text{ and } t = \langle t_1,\ t_2\rangle\}$.
  We consider the multi-way natural join $V_1 \bowtie \cdots \bowtie V_m = (((V_1 \bowtie V_2)\ldots) \bowtie V_m)$.
- *Aggregation*: $\alpha_{G,aggr(B)}(V_1) = \{\langle T.G,\ aggr(T.B)\rangle \mid T \subseteq V_1 \text{ and } \forall t,\ t' \in T,\ t'' \notin T:$
  $t'.G = t.G\ \wedge\ t''.G \neq t.G\}$

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

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

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 to apply 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 it upwards. We begin by focusing on each operator, defining derivations of the operator's result tuples based on its input tuples.

According to relational semantics, each operator generates its result tuple-by-tuple based on its operand tables. Intuitively, given a tuple t in the result of operator Op, only a subset of the input tuples produce t. We say that the 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 10 illustrates the derivation of a view tuple. In the figure, operator Op is applied to tables T_1 and T_2, which may be base tables or temporary results from other operators. (In general, we use R's to denote base tables and T's to denote tables that may be base or derived.) Table T is the operation result. Given tuple t in T, only subsets T_1^* and T_2^* of T_1 and T_2 contribute to t. $\langle T_1^*, T_2^*\rangle$ is called t's derivation. The formal definition of tuple derivation for an operator is given next, followed by additional explanation.

Definition 4.1 (Tuple Derivation for an Operator) Let Op be any of our relational operator (σ, π, $\bowtie$, α) 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:

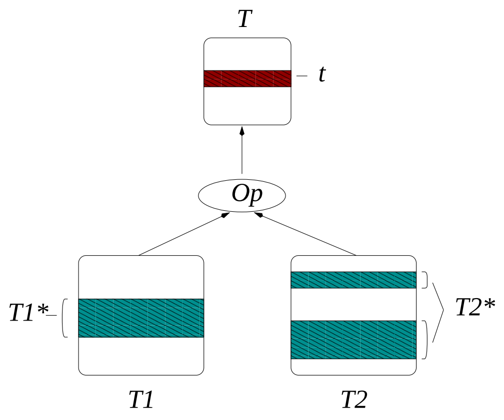


Figure 10: Derivation of tuple t

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

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

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 $i = 1..m$. Op^{-1} can be extended to represent the derivations 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.[1] $\square$

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 view tuple will not appear in any view 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 view. Theorem 4.2 shows that there is a unique derivation for any given view tuple. Recall that all proofs are provided in the Appendix.

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. $\square$

Example 4.3 (Tuple Derivation for Aggregation) Given table R in Figure 11(a), and tuple $t = \langle 2, 8\rangle \in \alpha_{X,sum(Y)}(R)$ in Figure 11(b), the derivation of t is

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

[1] $\langle R_1, \ldots, R_m\rangle \cup \langle S_1, \ldots, S_m\rangle = \langle (R_1 \cup S_1), \ldots, (R_m \cup S_m)\rangle$

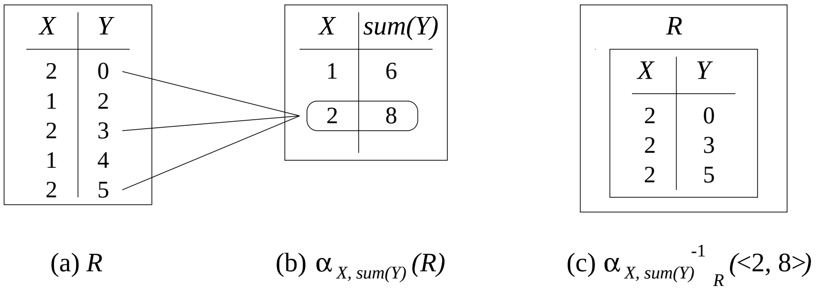


Figure 11: Tuple derivations for aggregation

shown in Figure 11(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 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 and t be a result tuple.

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

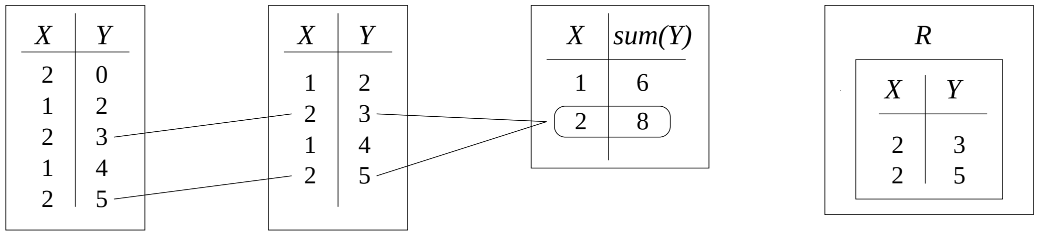
4.3 Tuple Derivations for Views

Now that we have defined tuple derivations for the operators, we proceed to define tuple derivations for views. As mentioned earlier, a view definition can be expressed as a query tree evaluated bottom-up. 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 query tree to the derivation tuple sets, and presenting the intermediate results for each operator in the evaluation.

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.*



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

Figure 12: Tuple derivation for a view

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 $v^{-1}{}_{R_i}(t)$. Finally, 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 a view tuple's derivation is unique.

Example 4.6 (Tuple Derivation for a View) Given base table R in Figure 12(a), view $V = \alpha_{X,sum(Y)}(\sigma_{Y \neq 0}(R))$ in Figure 12(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 12(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 12(d). $\Box$

We now state some properties of 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 $\bigodot_{j=1..k}$ represents the multi-way concatenation of relation lists. [2] $\Box$

Theorem 4.7 is a result of Definition 4.5. It shows that given a view V with a complex definition tree, we can break down its definition query tree into intermediate views, and compute a tuple's derivation by recursively tracing the hierarchy of intermediate views.

[2]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 relations are renamed so that the same relation never appears twice.

Since we define tuple derivations inductively based on the view query tree, an interesting question arises: Are the derivations of 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 are also equivalent.

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

According to Theorem 4.8, we can transform an SPJ view to a simple canonical form before tracing tuple derivations. Unfortunately, views with aggregation do not have this nice property, as shown in the following example.

Example 4.9 (Tuple Derivations for Equivalent 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 11(a) and 12(a), Figures 11(b) and 12(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 11(c)) is different from that according to v_2 (shown in Figure 12(d)). $\Box$

Given Definition 4.5, a straightforward way to compute a view tuple derivation is to compute the intermediate results for all operators in the view definition and store them 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 next two sections, we separately consider SPJ views and views with aggregation (ASPJ views). We show in Section 5 that one relational query over the base tables suffices to compute tuple derivations for SPJ views. A recursive algorithm for ASPJ view derivation tracing that requires a modest amount of auxiliary information is described in Section 6.

5 SPJ View Derivation Tracing

Derivations for tuples in SPJ views can be computed using a single relational query over the base data. In this section, we first define the general concept of a *derivation tracing query*, which can be applied directly to the base tables to compute a view tuple's derivation. We then specify tracing queries for SPJ views, and discuss optimization issues for tracing 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 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 with base tables $R_1, \ldots, R_m$. Given view definition v over $R_1, \ldots, R_m$, and tuple $t \in v(R_1, \ldots, R_m)$, $TQ_{t,v}$ 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 over 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 $TQ_{T,v}(D)$. $\Box$

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 SPJ 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, so that its tuple derivations can be computed systematically using a single query. 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 $A_i \subseteq \mathbf{T}$, $i = 1..m$.

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

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 $v(D) = \pi_A(\sigma_C(R_1 \bowtie \cdots \bowtie R_m))$ be an SPJ view over D. Given tuple $t \in v(D)$, 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 $T \subseteq v(D)$, 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 semi-join operator. $\Box$

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 13(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. $\Box$

5.3 Tracing Query Optimization

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 13(b) shows the optimized tracing query for the `Calif` tuple.

If sufficient key information is present, the tracing query is even simpler:

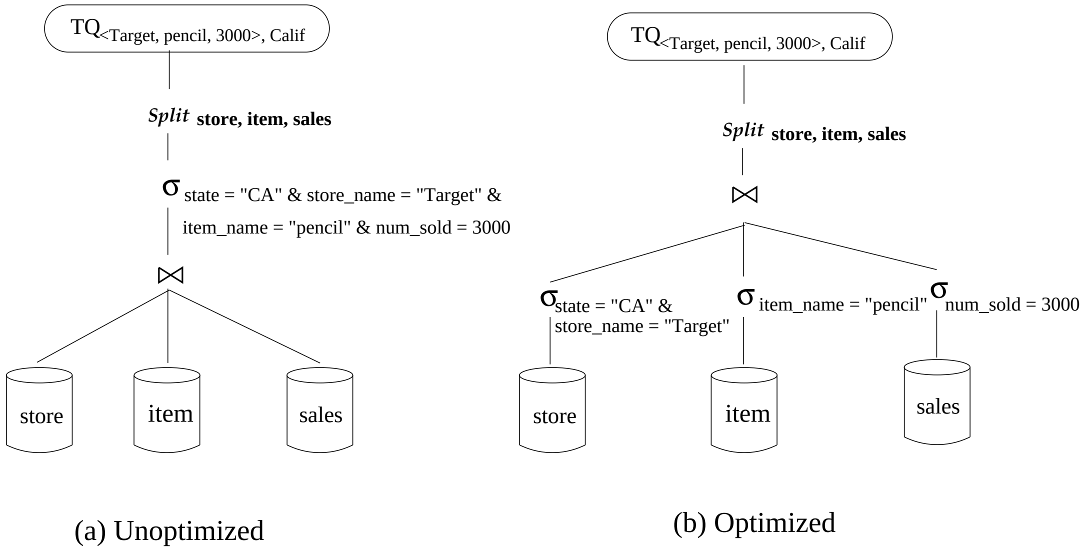


Figure 13: Derivation tracing query for `<Target, pencil, 3000>`

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 the base keys (i.e., $K_i \in A$, $i = 1..m$). View tuple t's derivation is $\langle \sigma_{K_1 = t.K_1}(R_1), \ldots, \sigma_{K_m = t.K_m}(R_m) \rangle$. $\Box$

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 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 efficiently. For more complex views with aggregations we cannot compute tuple derivations by a single query over the base tables. In the next section, we present a recursive tracing algorithm for these views.

6 Derivation Tracing Algorithm 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 certain intermediate results. For example, in Q2 in Example 3.2 the user 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 $t.$`total` $= 5400$. In order to trace the derivation correctly, we need tuple ⟨Macy's, 5400⟩ 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 the derivation tracing query for a simple one-level ASPJ view. 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. These intermediate results can either be recomputed from the base tables when needed, or they can be stored as materialized auxiliary views in a warehouse; this issue is further discussed in Section 7.1. 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. 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 a general ASPJ view query tree into a form composed of $\alpha\pi\sigma\bowtie$ operator sequences, which we call *ASPJ segments*. Each segment in the query tree except the topmost must include a non-trivial aggregation operator. We call this form the *ASPJ canonical form*.

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, $j = 1..k$. $\square$

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 compute a tuple derivation for a one-level ASPJ view.

Theorem 6.2 (Derivation Tracing Query for a One-Level ASPJ View) Given a one-level ASPJ view $V = v(T_1, \ldots, T_m) = \alpha_{G,aggr(B)}(\pi_A(\sigma_C(T_1 \bowtie \cdots \bowtie T_m)))$, and given tuple $t \in V$, t's derivation in $T_1, \ldots, T_m$ according to v can be computed by applying the following query to the base tables:

$$TQ^{\alpha}_{t,v} = Split_{\mathbf{T_1},\ldots,\mathbf{T_m}}(\sigma_{C \wedge G=t.G}(T_1 \bowtie \cdots \bowtie T_m))$$

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

$$TQ^{\alpha}_{T,v} = Split_{\mathbf{T_1},\ldots,\mathbf{T_m}}(\sigma_C(T_1 \bowtie \cdots \bowtie T_m) \ltimes T) \qquad \square$$

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

6.3 Recursive 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 14. $\square$

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 view or base tables at the next level below. Details follow.

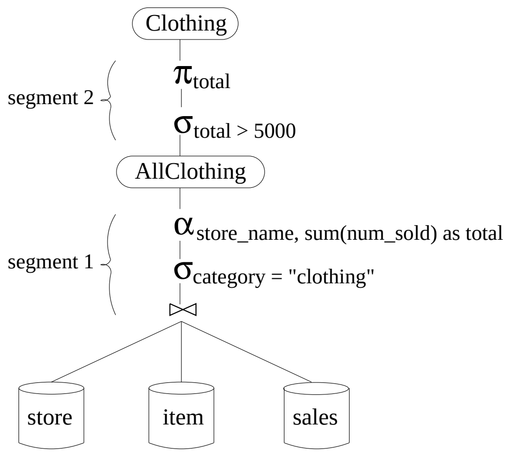


Figure 14: ASPJ segments and intermediate views for `Clothing`

6.3.1 Algorithm

Figure 15 presents our recursive derivation tracing algorithm for a general ASPJ view. Given a view definition v in ASPJ canonical form, and tuple $t \in v(D)$, procedure `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 16). 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 9. $\square$

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 7.1.

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.

```
procedure TupleDerivation(t, v, D)
begin
      return (TableDerivation({t}, v, D));
end
```

```
procedure TableDerivation(T, v, D)
begin
      if v = R ∈ D then return (⟨T⟩);
      // else 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)
begin
      D* ← ∅;
      for j ← 1 to k do
         D* ← D* ∘ TableDerivation(T_j, v_j, D);
      return (D*);
end
```

Figure 15: Algorithm for ASPJ view tuple derivation tracing

store_name	total
Target	1400
Macy's	5400

Figure 16: `AllClothing` table

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.

Our recursive derivation tracing algorithm can be used to trace the derivation of any tuple in any ASPJ view in a conventional database. However, certain additional issues arise when performing derivation tracing in a multi-source warehousing environment, which we proceed to discuss in the next section.

7 Derivation Tracing in a Warehousing Environment

In Sections 5 and 6 we presented algorithms to trace view tuple derivations. Our algorithms assume that all of the base tables as well as the intermediate views are accessible, and that they are consistent with the view being traced. These assumptions may not hold when we are tracing derivations for a warehouse view defined on remote distributed sources. The following problems may arise:

1. Efficiency problem: Querying remote sources and performing selections and joins (in the tracing queries) over them for each tuple derivation trace can be very inefficient. Also, recomputing intermediate views for tracing multi-level ASPJ views can be expensive.

2. Consistency problem: The warehouse may not refresh its views in real time, which means that warehouse views can become out of date. Thus, we cannot always compute the derivation of tuples in the "old" view from the "new" source base tables. For example, if a base tuple in the derivation of a view tuple has just been deleted from the source, but the change has not yet been propagated to the view, then the user sees the view tuple but cannot correctly trace its derivation since a relevant base tuple is gone.

3. Legacy source problem: Views defined on inaccessible legacy sources are not traceable because the base tables are not available.

Storing auxiliary views in the warehouse to reduce computation cost and to avoid querying the sources [QGMW96] is a solution that solves all three of the above problems. The price is extra storage and view maintenance costs. In Section 7.1, we consider the trade-offs between materializing and recomputing intermediate results, and propose to store intermediate aggregation results to improve overall performance. In Section 7.2, we introduce *derivation views*, which store information about sources so that view derivations always can be computed without querying the sources.

7.1 Materializing vs. Recomputing Intermediate Aggregate Results

In Section 6, we saw that intermediate aggregation results are needed for derivation tracing in the general case. There are two ways to obtain such information. One is to recompute the intermediate result during the tracing process. This approach requires no permanent extra storage, but the tracing process takes longer, especially when the recomputation may require querying the sources. The other way is to maintain materialized *auxiliary views* containing the intermediate results. In this case, less computation is required at tracing time, but the auxiliary views must be stored and kept up-to-date. Due to the characteristics of warehousing environments, we suggest warehouses maintain the intermediate aggregation results as materialized auxiliary views rather than recomputing them [QGMW96].

Example 7.1 (Materialized View for `AllClothing`) To improve the efficiency of the tracing process in Example 6.4, we materialize auxiliary view `AllClothing` (in Figure 14) with the contents shown in Figure 16. □

Note that when materializing `AllClothing`, tuple ⟨Target, 1400⟩ is not used when tracing tuple derivations for `Clothing`. In fact it is filtered out by the selection condition $\sigma_{total>5000}$ in `Clothing`'s definition. In this case, materializing the result of $V' = \sigma_{total>5000}(\texttt{AllClothing})$ instead of storing `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 `AllClothing` in any case in order to maintain V'. Therefore, materializing V' is not actually likely to be an improvement. In general, given a view definition tree where selections are pushed down as far as possible, all selection conditions above an aggregate must be on the summary attributes, and therefore are not incrementally maintainable without storing the entire aggregation results.

7.2 Storing Derivation Views in a Multi-Source Warehouse

The problems described earlier (e.g., inefficiency, inconsistency) in warehouse view derivation tracing arise when we apply our tracing queries to remote sources. We therefore may prefer to store auxiliary information about the sources in the warehouse, in order to avoid source queries during each derivation trace. Note that these auxiliary views may be in addition to the intermediate views for aggregate view tracing discussed in Section 7.1.

There are various strategies for storing such auxiliary views. A simple extreme solution is to store a copy of each source table in the warehouse. Our algorithm will then query the base views 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 base views 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 much higher. We propose an intermediate scheme that achieves low tracing query cost with modest extra storage and maintenance cost.

After adding auxiliary views as described in Section 7.1, the view definition is broken down into multiple ASPJ segments. Only views defined by the lowest-level segments are directly over the source tables, and it is these views that are of concern. Let $V = v(R_1, \ldots, R_m)$ be such a view. Based on V, 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 7.2 given next. Theorem 7.3 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 7.2 (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 the source tables. The *derivation view for* V, denoted as $DV(v)$, is

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

Theorem 7.3 (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 7.2. Given a tuple $t \in V$, t's derivation can be computed using $DV(v)$ as follows:

$$TQ^{\alpha}_{t,v} = Split_{\mathbf{R_1,\ldots,R_m}}(\sigma_{G=t.G}(DV(v)))$$

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

$$TQ^{\alpha}_{T,v} = Split_{\mathbf{R_1,\ldots,R_m}}(DV(v) \ltimes T). \qquad \square$$

Example 7.4 (Derivation View for `AllClothing`) The view `AllClothing` in Example 7.1 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 17 shows the derivation view definition, and Figure 18 shows its contents. The derivation tracer need only to query `DV_AllCothing` to compute view `AllClothing`'s tuple derivations, as shown in Figure 17. $\square$

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

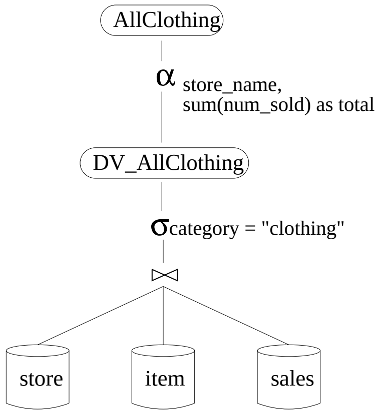


Figure 17: View definition for `DV_AllClothing`

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 18: `DV_AllClothing` table

7.3 A Warehousing System Supporting Derivation Tracing

Figure 19 illustrates an overall warehouse structure that supports tuple derivation tracing with materialized intermediate aggregation results and derivation views. Recall question Q2 from Example 3.2. The query tree on the left side of Figure 19 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 7.1). Furthermore, to trace tuples in `AllClothing`, derivation view `DV_AllClothing` is maintained (as discussed in Section 7.2). 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]. Solid arrows on the right side of Figure 19 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 over D as specified in Theorem 7.3. T_2 is t's derivation over D.

There are alternative derivation views to the one we described that trade tracing query cost for

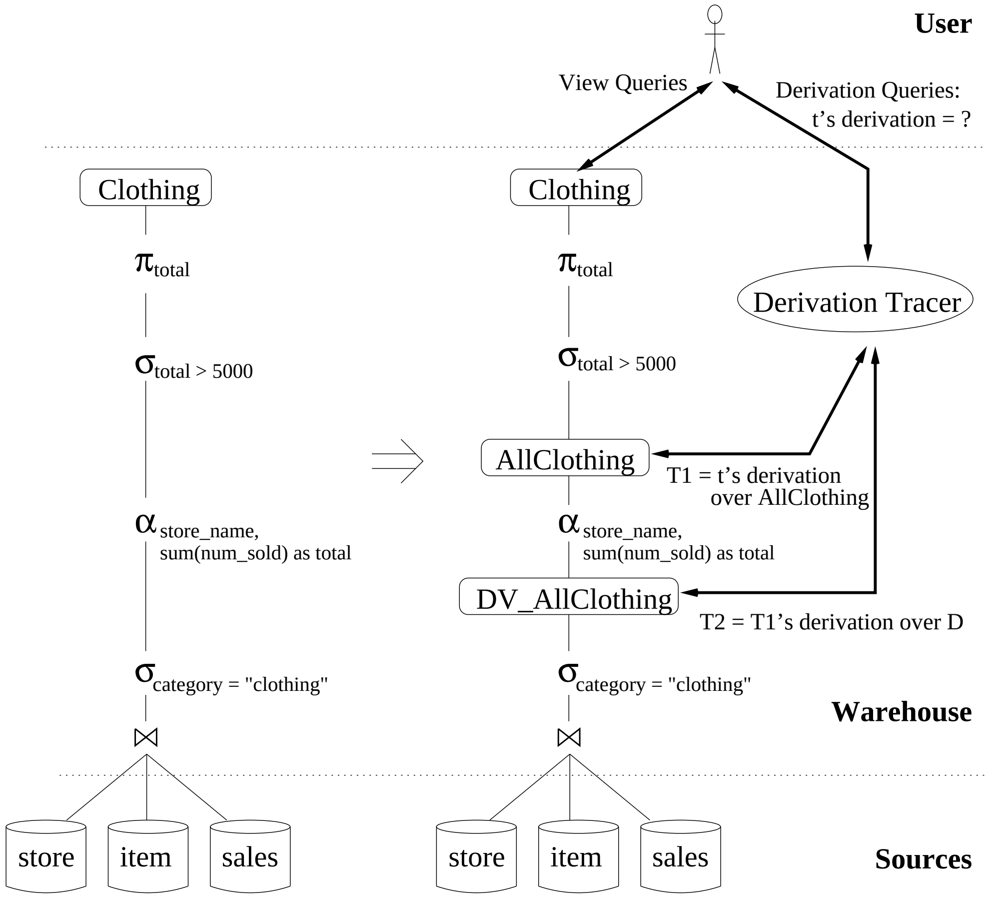


Figure 19: Derivation tracing in the warehouse system

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 query the sources and recompute information during 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].

8 Related Work Revisited

In this section, we revisit some related topics, including top-down Datalog query processing and the view update problem, and examine the differences between those problems and ours. We also show how lineage tracing can be applied to the view update problem.

8.1 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), and we support lineage tracing for aggregation views, 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.

8.2 View Update Problem

The well-known view update problem is to transform updates on views into updates against the base tables, so that the new base tables will continue to derive the updated view. The problem was first formulated in [DB78].

A view update can be an insertion, deletion, or modification of a view tuple. [Mas84] deduced a set of *view update translation rules* for different view update commands against select, project, and join views. Given a view update command, in some cases more than one set of base table updates can achieve the same view update effect. Much effort has been focused on finding appropriate translations for specific cases [BS81, CM89, Kel86, LS91]. In general, extra semantic information is needed to choose a translation.

View update algorithms cannot be used directly to compute view tuple derivations. First, none of the algorithms we know of consider aggregation. Second, although the algorithms do identify a set of base tuples that can affect a view tuple, the base tuples identified may not even derive the view tuple, and therefore do not satisfy our Definition 4.5 of a view tuple derivation. In general, the view update approach and the derivation tracing approach answer two different questions: "Which base tuples can affect a view tuple?" and "Which base tuples exactly derive a view tuple?", respectively. The two questions are not equivalent, but they are related to each other. Our derivation tracing algorithms can be used to guide the view update process to find an appropriate view update translation in some cases. For deletions and modifications, our derivation tracing algorithms can directly identify an appropriate set of base relation tuples to modify, as shown in the following example.

Example 8.1 (View Update: Deletion) In Example 4.6 (Figure 12), 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 also to delete $\langle 2,0\rangle$, 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 in the "guessing" process in certain cases, as shown in the following example.

Example 8.2 (View Update: Insertion) Suppose view update "`insert` $\langle 3,2\rangle$ `into` V" is issued to the view in Example 4.6 (Figure 12). 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 8.2 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 incorporated with base table constraints can be very useful in reducing the number of possible translations.

9 Conclusions and Future Work

We formulated the *view data lineage* problem and presented lineage tracing algorithms for relational views with aggregation. Our algorithms identify the exact subset 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-down" to the source data that produced warehouse data of interest. Follow-on and future work includes the following.

- We have extended the results in this paper to view definitions that use bag (instead of set) semantics, and to additional relational operators including $\cup$ and $-$. Please see [CWW98]. We also plan to extend our work to handle recursive views.

- 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 7.1 and 7.2 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 will apply our derivation tracing techniques to the view schema evolution problem, as motivated in Section 1.
- We will further study how lineage tracing can be incorporated with existing techniques to help solve the view update problem. As seen in Example 8.2, one interesting problem is to extend our derivation tracing algorithms to handle tuples not yet in the view.
- Most importantly, we are implementing a lineage tracing package within the *WHIPS* data warehousing prototype at Stanford[WGL+96]. Once the basic algorithms are completed, we plan to experiment with appropriate user interface tools through which an analyst can obtain and browse derivation information. For example, the analyst may wish to see not only the base derivation data itself, but also a representation of the process by which the view data item was derived.

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, 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.

[CM89] I. Chen and D. McLeod. Derived data update in semantic databases. In *Proc. of the Fifteenth International Conference on Very Large Data Bases*, pages 225–235, Amsterdam, The Netherlands, 1989.

[Col96] G. Colliat. OLAP, relational, and multidimensional database systems. *SIGMOD Record*, 25(3):64–69, September 1996.

[CWW98] Y. Cui, J. Widom, and J.L. Wiener. A complete solution for tracing the lineage of relational view data. Technical report, Stanford University, Database Group, August 1998. Available at http://www-db.stanford.edu/pub/papers/lineage-full.ps.

[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, Berlin, 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, sub-totals. In *Proc. of the Twelfth International Conference on Data Engineering*, pages 152–159, New Orleans, Louisiana, February 1996.

[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.

[GM95] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Engineering Bulletin, Special Issue on Materialized View and Data Warehousing*, 18(2):3–19, June 1995.

[GMS93] A. Gupta, I. S. Mumick, and V. Subrahmanian. Maintaining views incrementally. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 157–166, Washington, DC, May 1993.

[Gup97] H. Gupta. Selection of views to materialize in a data warehouse. In *Proc. of the Sixth International Conference on Database Theory*, pages 98–112, Delphi, Greece, January 1997.

[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.

[HRU96] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 205–216, Montreal, Canada, June 1996.

[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.

[Kel86] A. M. Keller. Choosing a view update translator by Dialog at view definition time. In *Proc. of the Twelfth International Conference on Very Large Data Bases*, pages 467–476, Kyoto, Japan, August 1986.

[LQA97] W.J. Labio, D. Quass, and B. Adelberg. Physical database design for data warehousing. In *Proc. of the Thirteenth International Conference on Data Engineering*, pages 277–288, Birmingham, UK, April 1997.

[LS91] J. A. Larson and A. P. Sheth. Updating relational views using knowledge at view definition and view update time. *Information Systems*, 16(2):145–168, 1991.

[Mas84] Y. Masunaga. A relational database view update translation mechanism. In *Proc. of the Tenth International Conference on Very Large Data Bases*, pages 309–320, Singapore, August 1984.

[MQM97] I. Mumick, D. Quass, and B. Mumick. Maintenance of data cubes and summary tables in a warehouse. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 100–111, Tucson, Arizona, May 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.

[Qua96] D. Quass. Maintenance expressions for views with aggregation. In *Proc. of the Workshop on Materialized Views, Techniques and Applications*, pages 110–118, Montreal, Canada, June 1996.

[QW91] X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *IEEE Transactions on Knowledge and Data Engineering*, 3(3):337–341, 1991.

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

[WB97] M. Wu and A. Buchmann. Research issues in data warehousing. In *Proc. of the Datenbanksysteme in Buero, Technik und Wissenschaft*, Ulm, Germany, March 1997.

[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 assume set semantics in this paper. 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, or π: $m = 1$.

(a) Since T'_1 and T''_1 satisfy Definition 4.1(a), $Op(T^*_1) = Op(T'_1 \cup T''_1) = Op(T'_1) \cup Op(T''_1) = \{t\}$

(b) $\forall t^* \in T^*_1$: $t^* \in T'_1$ or $t^* \in T''_1$. Since T'_1 and T''_1 satisfy Definition 4.1(b), $Op(\{t^*\}) \neq \varnothing$

If Op is $\bowtie$:

(a) Since $\langle T'_1, \ldots, T'_m \rangle$ 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, $Op(T^*_1, \ldots, \{t^*\}, \ldots, T^*_m) \supseteq Op(T'_1, \ldots, \{t^*\}, \ldots, T'_m)$

$\Rightarrow$ $Op(T^*_1, \ldots, \{t^*\}, \ldots, T^*_m) \neq \varnothing$

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. $\square$

A.2 Proof of Theorem 4.4

Theorem 4.4 (Tuple Derivations for Operators) Let $T, T_1, \ldots, T_m$ be tables and t be a result tuple.

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

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: $\langle {T_1}'',\ldots,{T_m}''\rangle \subseteq \langle {T_1}',\ldots,{T_m}'\rangle$.

A.2.1 ${\sigma_C}^{-1}{}_{\langle T\rangle}(t) = \langle\{t\}\rangle$, for $t \in \sigma_C(T)$.

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

1. $t \in \sigma_C(T)$
$\Rightarrow$ $t \in T$
$\Rightarrow$ $T' \subseteq T$

2. $t \in \sigma_C(T)$
$\Rightarrow$ t satisfies C
$\Rightarrow$ $\sigma_C(T') = \{t\}$

3. Consider an arbitrary $t' \in T'$
$t' = t$
$\Rightarrow$ $\sigma_C(\{t'\}) = \{t\} \neq \varnothing$

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

A.2.2 ${\pi_A}^{-1}{}_{\langle T\rangle}(t) = \langle\sigma_{A=t}(T)\rangle$, for $t \in \pi_A(T)$.

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

1. $T' \subseteq T$ according to the definition of σ

2. $\forall t' \in T'$: $t'.A = t$
$\Rightarrow$ $\pi_A(T') = \{t\}$

3. Consider an arbitrary $t' \in T'$

$t'.A = t$

$\Rightarrow$ $\pi_A(\{t'\}) = \{t\} \neq \varnothing$

4. Consider an arbitrary T'' that satisfies 1, 2, 3.

$\pi_A(T'') = \{t\}$

$\Rightarrow$ $\forall t'' \in T''$: $t''.A = t$

$\Rightarrow$ $\forall t'' \in T''$: $t'' \in T'$

$\Rightarrow$ $T'' \subseteq T'$ $\square$

A.2.3 $\bowtie^{-1}{}_{\langle T_1,\ldots,T_m\rangle}(t) = \langle \{t.\mathbf{T_1}\}, \ldots, \{t.\mathbf{T_m}\}\rangle$, 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 $\forall t'' \in T_i''$: $T_1{}'' \bowtie \ldots \{t''\} \bowtie T_m{}'' \neq \varnothing$

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

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

$\Rightarrow$ $T_i{}'' \subseteq T_i{}'$ $\square$

A.2.4 $\alpha_{G,aggr(B)}{}^{-1}{}_{\langle T\rangle}(t) = \langle \sigma_{G=t.G}(T)\rangle$, 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.
$\alpha_{G,aggr(B)}(T'') = \{t\}$
$\Rightarrow$ $\forall t'' \in T''$: $t''.G = t.G$
$\Rightarrow$ $\forall t'' \in T''$: $t'' \in T'$
$\Rightarrow$ $T'' \subseteq T'$ $\square$

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 $\bigodot_{j=1..k}$ 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 $j = 1..k$, $\forall t \in V$: $\forall t^* \in R_i \in D$:

$$t^* \text{ contributes to } t \text{ according to } v \iff \begin{array}{l} \exists j \in 1..k, t' \in V_j \text{ such that} \\ t^* \text{ contributes to } t' \text{ according to } v_j \text{ and} \\ t' \text{ contributes to } t \text{ according to } v'. \end{array} \quad \square$$

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. $\square$

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[3] 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, $t^* \in v^{-1}{}_{R_i}(t)$ $\square$

A.4 Proof of Theorem 4.8

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

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 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. $\square$

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

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. $\Box$

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 our 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 $t \in v_1(D) = v_2(D)$, $\forall t^* \in v_1^{-1}(t)$: 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 $R'_i = \emptyset$, 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, ${v_1^{-1}}_D(t) = {v_2^{-1}}_D(t)$. $\Box$

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 $v(D) = \pi_A(\sigma_C(R_1 \bowtie \cdots \bowtie R_m))$ be an SPJ view over D. Given tuple $t \in v(D)$, 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 $T \subseteq v(D)$, 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 semi-join operator. $\Box$

Proof: We need to prove:

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

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:

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

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

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 the base keys (i.e., $K_i \in A$, $i = 1..m$). View tuple t's derivation is $\langle \sigma_{K_1=t.K_1}(R_1), \ldots, \sigma_{K_m=t.K_m}(R_m)\rangle$. $\Box$

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 $i = 1..m$. $\Box$

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 $V = v(T_1, \ldots, T_m) = \alpha_{G,aggr(B)}(\pi_A(\sigma_C(T_1 \bowtie \cdots \bowtie T_m)))$, and given tuple $t \in V$, t's derivation in $T_1, \ldots, T_m$ according to v can be computed by applying the following query to the base tables:

$$TQ^{\alpha}_{t,v} = Split_{\mathbf{T_1},\ldots,\mathbf{T_m}}(\sigma_{C\wedge G=t.G}(T_1 \bowtie \cdots \bowtie T_m))$$

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

$$TQ^{\alpha}_{T,v} = Split_{\mathbf{T_1},\ldots,\mathbf{T_m}}(\sigma_C(T_1 \bowtie \cdots \bowtie T_m) \ltimes T) \qquad \Box$$

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:

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

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. $\Box$

A.8 Proof of Theorem 7.3

Theorem 7.3 (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 7.2. Given a tuple $t \in V$, t's derivation can be computed using $DV(v)$ as follows:

$$TQ^{\alpha}_{t,v} = Split_{\mathbf{R_1},\ldots,\mathbf{R_m}}(\sigma_{G=t.G}(DV(v)))$$

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

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

Proof: The proof is obvious according to the definition of the derivation view (Definition 7.2) and tracing queries for one-level ASPJ views (Theorem 6.2). $\Box$