Practical Lineage Tracing in Data Warehouses*

Yingwei Cui and Jennifer Widom
Computer Science Department, Stanford University
{cyw, widom}@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 formalize the problem, and we present a lineage tracing algorithm for relational views with aggregation. Based on our tracing algorithm, we propose a number of schemes for storing *auxiliary views* that enable consistent and efficient lineage tracing in a multi-source data warehouse. We report on a performance study of the various schemes, identifying which schemes perform best in which settings. Based on our results, we have implemented a lineage tracing package in the WHIPS data warehousing system prototype at Stanford. With this package, users can select view tuples of interest, then efficiently "drill through" to examine the exact source tuples that produced the view tuples of interest.

1 Introduction

Data warehousing systems collect data from multiple distributed sources, integrate the information as *materialized views* in local databases, and keep the view contents up-to-date when the sources change [CD97, IK93, LW95, Wid95]. Users can then perform data analysis and mining based on the warehouse views. However, the view contents alone sometimes do not provide sufficient information for in-depth analysis. In many cases, it is useful to be able to "drill through" from interesting view data all the way to the original source data that derived the view data. For a given view data item, identifying the exact set of base data items that produced the view data item is termed the *view data lineage* problem [CWW97]. The primary motivation for supporting view data lineage is to enable further analysis of interesting or potentially erroneous view data, and this function is of clear benefit in OLAP and data mining environments. Other application domains include, e.g., scientific databases [HQGW93] and network monitoring systems. Our algorithms and results for lineage tracing also can be applied to the problems of *view update* [DB78], materialized view schema evolution [GMR95], and data cleansing. See [CWW97] for further discussion on these applications.

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

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

There has been growing interest in lineage tracing in industry (often referred to there as *drill-through*) although products so far support only primitive or special-purpose capabilities, based only on schema information or specific forms of aggregate views. The problem of how to enable and perform instance-level lineage tracing for general views in a consistent and efficient manner in a warehousing environment has not been studied to the best of our knowledge. In this paper, we provide an initial practical solution for tracing the lineage of tuples in set-based *aggregate-select-project-join (ASPJ)* views in relational data warehouses.[1] This class of views is quite powerful and includes many of the commonly-used warehouse data transformations. As future work we plan to extend our results to other, more general, data transformations.

The contributions of the paper include the following.

1. We formulate the view data lineage problem and develop a lineage tracing algorithm for relational ASPJ views. The tracing procedure can be generated automatically from a view definition, and all queries in the procedure can be optimized easily by a standard DBMS. We also discuss further optimizations of the tracing procedure in special-case scenarios.

2. For the restricted case of SPJ views, we introduce a family of schemes for storing *auxiliary views* to provide consistent and efficient lineage tracing in a distributed warehousing environment. The various schemes offer different advantages and tradeoffs, thus different schemes are suitable for different settings.

3. We present a performance study of our proposed auxiliary view schemes using a cost model that incorporates lineage tracing cost as well as overall view maintenance and storage costs. We identify which schemes present the best overall performance in which settings.

Based on these results, we have implemented a view tuple lineage tracing package as part of the WHIPS [HGMW$^+$95] data warehousing prototype at Stanford.

1.1 Related Work

The problem of tracing view data lineage is clearly related to work in the area of view update [Sto75, Kel86]. In [CWW97] we provide an in-depth discussion of the relationship, including showing how our work on lineage tracing can be used to improve the view update process. Our work also relates to some extent to work in deductive databases [Ull89], scientific databases [HQGW93], and multi-dimensional databases [GBLP96]. In each case, a problem is addressed that roughly includes retrieving lineage information for a specific type of views. However, none of these papers, including the previous view update work, develops a complete lineage tracing solution for general relational views with aggregation in a distributed warehousing environment. Nor has the performance issue been investigated in any depth considering lineage tracing, view maintenance, and storage costs. Note that our initial work on lineage tracing in warehousing environments [CWW97] forms the basis for the present paper.

[LBM98] presents a solution for explicitly storing lineage (which they call *attribution*) of data items in query results based on a mediation architecture. Coarse-grained lineage information is stored when queries are computed—it identifies which sources data items were derived from, along with additional information such as timestamps and source quality. In the data warehousing context,

[1] Our approach generalizes to include other relational operators such as *union* and *difference*, as well as to duplicate semantics, but we omit the extensions due to space limitations in this paper; please see [CWW97].

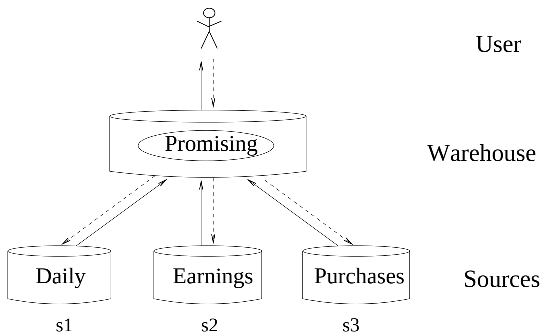

Figure 1: A simple warehousing scenario

[BB99] presents a scheme whereby the identifier of each warehouse data transformation is attached to all objects generated by the transformation, so that the user can trace which transformations produced each warehouse data object. Neither [LBM98] nor [BB99] can retrieve fine-grained lineage, such as identifying exactly which source data items produced a given view data item. [WS97] proposes a framework for computing and verifying the approximate lineage of any given view "grain" based on the view's *weak inverse*. The paper does not, however, provide a mechanism for generating the weak inverse given the view definition. Finally, [FJS97] uses a statistical approach to reconstruct base data from summary data and certain knowledge of constraints. This approach does not require access to the base data. However, it provides only estimated lineage information, and does not ensure accuracy. In contrast to all of the above approaches, we propose a solution that computes exact, fine-grained lineage for any tuple in any relational ASPJ view, using a tracing procedure automatically generated from the view definitio and a small amount of auxiliary information maintained together with the warehouse views.

Some other recent work has dealt with data warehouse design issues, and thus relates to our storage of auxiliary views to improve performance. [LQA97] and [RSS96] consider the problem of selecting auxiliary views and indexes to store in order to minimize total view maintenance cost. [Gup97] provides a theoretical solution that finds the warehouse design that minimizes warehouse query cost given certain constraints on the maintenance cost. All three papers develop heuristics and search algorithms for a large solution space. In this paper, we consider the problem of storing auxiliary views primarily to enable and optimize the lineage tracing process. Because lineage tracing queries are of a special form, we have a relatively small solution space. Therefore, we are able to conduct a comprehensive performance study of different schemes under different scenarios considering lineage tracing, view maintenance, and storage costs.

In industry, most OLAP systems support a "drill-down" capability for multidimensional warehouse data, allowing aggregated dimension data to be unrolled one level at a time within the warehouse. A few products further support "drilling through" to the original transactional data, also stored in the warehouse [DB2, Pow]. Some on-line reporting systems allow report-to-report "drilling" based on annotations, e.g., by creating hyperlinks from data in one report to lineage information in another report [Imp]. All of these products focus on specific types of views in specific settings, and do not enable lineage tracing through general relational views as we consider in this paper.

2 Motivating Example

In this section, we provide a simple example to motivate the lineage tracing problem and to introduce terminology used throughout the paper. Figure 1 shows a simplified diagram of a financial data

Daily

ticker	high	low	closing
AAA	29	25	26
BBB	89	87	89
CCC	75	74	74
DDD	120	100	120
EEE	72	67	70
FFF	12	10	12
...	...	...	...

Earnings

ticker	industry	earnings
AAA	automobile	0.5
BBB	computer	4.0
CCC	computer	1.2
DDD	medicine	2.0
EEE	retail	1.5
FFF	automobile	0.5
...	...	...

Purchases

tranID	ticker	date	price	shares
0021	AAA	1/5/98	40	100
0022	BBB	1/5/98	16	100
0023	CCC	3/24/98	70	300
0024	BBB	6/7/98	40	50
0025	DDD	6/11/98	80	200
0026	CCC	7/2/98	80	300
...	...	...	...	...

Figure 2: Sample data

warehousing system, in which the solid arrows represent data flow and the dashed arrows represent queries.

Warehouse sources. The warehouse is based on information from the following source tables:

```
s1.Daily(ticker, high, low, closing)
s2.Earnings(ticker, industry, earnings)
s3.Purchases(tranID, ticker, date, price, shares)
```

The `Daily` table at source `s1` (a stock market database, say) contains the latest stock price information, including the high, low, and closing price of each stock (identified by ticker symbol) for the most recent working day. The `Earnings` table at source `s2` (an analysis firm's database, say) lists the latest earnings per share of each stock, and the industry the stock belongs to. The `Purchases` table at source `s3` (a stockbroker's database, say) records all stock purchases, including the transaction ID, purchase date and price, as well as number of shares in each transaction. Sample table contents appear in Figure 2.

Warehouse materialized view. Suppose the warehouse user wants to monitor a list of all "promising" industries, where an industry is regarded as promising if some stock in that industry is gaining money over all purchases (i.e., the latest closing price is higher than the average share purchase price), and the stock has a price-earnings ratio below 40. To this end, the warehouse defines a materialized view `Promising`. The view definition is expressed as SQL and as a relational algebra tree in Figure 3, where the α operator represents group-by and aggregation [CWW97]. Over our sample data the view contains two tuples, $\langle$`computer`$\rangle$ and $\langle$`medicine`$\rangle$.

Data lineage. Suppose the user wishes to learn more about why an industry is listed in his view. He selects view tuple $\langle$`computer`$\rangle$ and traces its *lineage* to see the source tuples that produced the view tuple. The lineage result contains three tables, each of which contains those tuples in the corresponding source that produced the view tuple; see Figure 4. Section 3 provides a formal definition of view data lineage and a procedure to identify the lineage of any tuple in any given view.

Auxiliary views. We may want to store additional information to enable lineage tracing. In our example, intermediate results from the group-by/aggregate node $\alpha_{\texttt{ticker},\ldots}$ in `Promising`'s algebraic definition (Figure 3) are needed to trace the lineage of tuples in `Promising`. We can recompute the relevant portion of the aggregate when tracing a tuple's lineage, or we can define an *auxiliary materialized view* over this node specifically for lineage tracing. As mentioned in the introduction, we may also choose to store auxiliary views in the warehouse in order to perform lineage tracing without querying the sources. Section 4 describes a number of schemes for auxiliary information and

```
CREATE VIEW Promising AS
SELECT e.industry
FROM Purchases p, Daily d, Earnings e
WHERE p.ticker = d.ticker
  AND d.ticker = e.ticker
  AND d.closing/e.earnings < 40
GROUP BY p.ticker, e.industry
HAVING
  SUM(p.price*p.shares)/SUM(p.shares) < d.closing
```

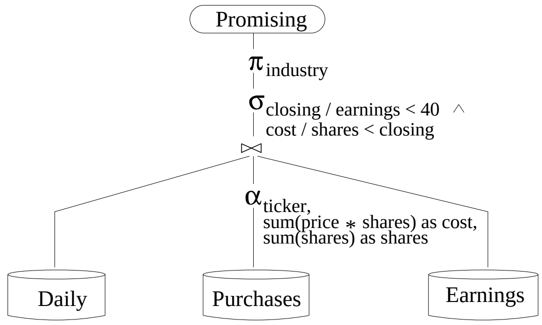

Figure 3: View definitions for `Promising`

Daily

ticker	high	low	closing
BBB	89	87	89

Purchases

tranID	ticker	date	price	shares
0022	BBB	1/5/98	16	100
0026	BBB	6/7/98	40	50

Earnings

ticker	industry	earnings
BBB	computer	4.0

Figure 4: Lineage of ⟨`computer`⟩ according to `Promising`

Section 5 analyzes their relative performance.

View maintenance. Both the user view being traced and any auxiliary views to support lineage tracing must be kept up-to-date when the sources change. In this paper, we assume a standard *incremental view maintenance* approach [GMS93, ZGMHW95]. Changes to each source table are recorded in a *delta table*. During view maintenance, changes to the view are computed using a predefined query called the *maintenance expression*. In our example, when tuples are inserted into `s1.Daily`, the insertions are recorded in a delta table `Daily-ins`. Insertions to the view `Promising` can then be computed using a query (maintenance expression) that is the same as in Figure 3, except that `Daily` is replaced by `Daily-ins`. Deletions to the view are computed similarly. We then *refresh* the view table by applying the changes, and the view becomes up-to-date. To compute the maintenance expressions, it is usually necessary to query the source tables, which can be problematic as discussed earlier. Prior work has addressed this problem by adding auxiliary views to ensure *self-maintainability*: a set of views is self-maintainable if they can be maintained using only the source changes and the view data, without querying the sources [QGMW96].

As we introduce auxiliary views in the warehouse to support lineage tracing, overall warehouse maintenance cost may increase since more views need to be maintained. However, the same auxiliary views that help lineage tracing often can help maintain the user view, sometimes even making the entire set of views self-maintainable.

3 Tracing View Data Lineage

Given a view data item I, the exact set of base data that produced I is called its *lineage*. In this section we provide a formal definition of the data lineage problem, and we present our lineage tracing algorithm for relational aggregate-select-project-join (ASPJ) views under set semantics. Extensions for additional operators (union and difference) and for duplicate semantics, as well as further details of our tracing solution, can be found in [CWW97].

3.1 View Data Lineage

To define the concept of view data lineage, we assume logically that the view contents are computed by evaluating an algebraic view definition query tree bottom-up. Each operator in the tree generates its result tuple-by-tuple based on the results of its children nodes, and passes the result upwards. For convenience in formulation, when a view references the same relation more than once, we consider each relation instance as a separate relation. (This approach allows view definitions to be expressed using an algebra tree instead of a graph, while not limiting the views we can handle.) We first focus on individual operators, defining the lineage of a tuple in an operator's result based on its input.

Definition 3.1 (Tuple Lineage for an Operator) Let Op be a relational operator (σ, π, $\bowtie$, or α), and let $T = Op(T_1, \ldots, T_m)$ be the table that results from applying Op to tables $T_1, \ldots, T_m$. Given a tuple $t \in T$, we define *t's lineage in $T_1, \ldots, T_m$ according to Op* to be $Op^{-1}{}_{\langle T_1,\ldots,T_m\rangle}(t) = \langle T_1^*, \ldots, T_m^* \rangle$, where $T_1^*, \ldots, T_m^*$ are **maximal** subsets of $T_1, \ldots, T_m$ such that:

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

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

Also, we say that $Op^{-1}{}_{T_i}(t) = T_i^*$ is *t's lineage in T_i*, and each tuple t^* in T_i^* *contributes to t*, for $i = 1..m$. Op^{-1} can be extended for the lineage 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 table lists, i.e., $\langle R_1, \ldots, R_m \rangle \cup \langle S_1, \ldots, S_m \rangle = \langle (R_1 \cup S_1), \ldots, (R_m \cup S_m) \rangle$. $\square$

In Definition 3.1, requirement (a) says that the lineage 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 lineage 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 lineage. By defining the T_i^*'s to be the maximal subsets that satisfy requirements (a) and (b), we make sure that the lineage contains exactly all the tuples that contribute to t. Thus, the lineage fully explains why a tuple exists in the view. (Further motivation for and discussion of this definition appears in [CWW97].)

Now that we have defined tuple lineage for the individual operators, we proceed to define tuple lineage for arbitrary 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 (logical) table corresponding to an intermediate node in the view definition tree, and t' further contributes to a view tuple t, then t^* contributes to t. More formally:

Definition 3.2 (Tuple Lineage for a View) Let D be a database with base tables $R_1, \ldots, R_m$, and let $V = v(D)$ be an ASPJ 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 defined over D, $j = 1..k$: Suppose $t' \in v_j(D)$ contributes to t according to operator Op (by Definition 3.1), and $t^* \in R_i$ contributes to t' according to view v_j (by this definition recursively). Then t^* *contributes to t according to v*.

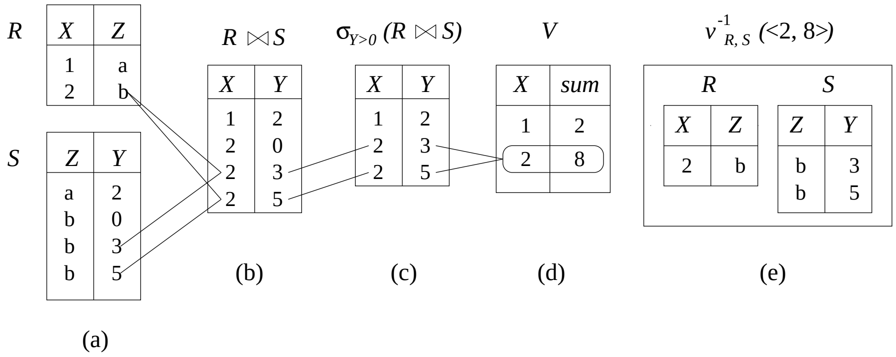

Figure 5: Tuple lineage for a view

Then *t's lineage in D according to v* is $v^{-1}{}_D(t) = \langle R_1^*, \ldots, R_m^* \rangle$, 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 $i = 1..m$. R_i^* is *t's lineage in* R_i *according to v*, denoted $v^{-1}{}_{R_i}(t)$. Finally, the lineage of a view tuple set T contains all base tuples that contribute to any view tuple in the set T: $v^{-1}{}_D(T) = \bigcup_{t \in T} v^{-1}{}_D(t)$. $\Box$

Example 3.3 (Tuple Lineage for a View) Consider tables $R(X, Z)$ and $S(Z, Y)$ in Figure 5(a), and view $V = \alpha_{X,sum(Y)}(\sigma_{Y>0}(R \bowtie S))$ in Figure 5(d). We are interested in the lineage of tuple $\langle 2, 8\rangle$ in V. It is easy to see that tuple $\langle 2, b\rangle$ in R and tuples $\langle b, 3\rangle$ and $\langle b, 5\rangle$ in S contribute to $\langle 2, 3\rangle$ and $\langle 2, 5\rangle$ in $(R \bowtie S)$, which contribute to the same tuples in $\sigma_{Y>0}(R \bowtie S)$, and further contribute to $\langle 2, 8\rangle$ in V. Thus $v^{-1}{}_{R,S}(\langle 2, 8\rangle) = \langle\{\langle 2, b\rangle\},\ \{\langle b, 3\rangle,\ \langle b, 5\rangle\}\rangle$ as shown in Figure 5(e). $\Box$

3.2 Lineage Tracing Procedure

Any aggregate-select-project-join (ASPJ) view v can be transformed into an equivalent form v' composed of $\alpha\pi\sigma\bowtie$ operator sequences by commuting and combining some select-project-join (SPJ) operators in the view definition tree. We call the resulting form v's *ASPJ canonical form*, and we call each $\alpha\pi\sigma\bowtie$ sequence an *ASPJ segment*. In ASPJ canonical form, each ASPJ segment except the outermost must include a non-trivial aggregation (α) operator (or it would be merged with an adjacent segment). For details see [CWW97].

An ASPJ view v and its canonical form v' are equivalent in the sense that $\forall D$: $v(D) = v'(D)$. In addition, lineage of tuples in v and v' are also equivalent, i.e., $\forall D$: $\forall t \in v(D) = v'(D)$: $v^{-1}{}_D(t) = v'^{-1}{}_D(t)$; see [CWW97] for a proof. Thus, when tracing lineage for an ASPJ view, we can first transform the view definition to its ASPJ canonical form, and then trace the view's lineage based on the canonical form. The uniformity of the canonical form makes it easier to generate view lineage tracing procedures automatically, and it can make the tracing process more efficient. In the remainder of the paper, we assume that the views we are tracing are in ASPJ canonical form. Note that our example view `Promising` in Figure 3 is already in ASPJ canonical form. It contains two ASPJ segments: (1) $\alpha_{\texttt{ticker},\ldots}$(`Purchases`) (2) $\pi_{\texttt{industry}}(\sigma_{\ldots}(\texttt{Daily} \bowtie \langle\text{segment}(1)\rangle \bowtie \texttt{Earnings}))$.

A view defined by one ASPJ segment is called a *one-level ASPJ view*, and SPJ views are special cases of one-level ASPJ views. The lineage of tuples in a one-level ASPJ view can be computed using a single relational query, called the *lineage tracing query*.

Definition 3.4 (Lineage Tracing Query) Let D be a database, and let v be a view over D. Given tuple $t \in v(D)$, $TQ_{t,v}$ is a *lineage tracing query for t and v* iff $TQ_{t,v}(D) = v^{-1}{}_D(t)$, where

```
procedure Lineage(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});
        D* ← ∅;
        for j ← 1 to k do
            // concatenate the lineage of each subview onto the result
            D* ← D*∘ Lineage(V_j*, v_j, D);
        return (D*);
end
```

Figure 6: Algorithm for ASPJ view tuple lineage tracing

${v^{-1}}_D(t)$ is t's lineage in D according to v, and $TQ_{t,v}$ is independent of database instance D. We can similarly define the tracing query for a view tuple set T, and denote it as $TQ_{T,v}(D)$. □

Theorem 3.5 (Lineage Tracing Query for One-level ASPJ Views) 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 lineage in $T_1, \ldots, T_m$ according to v can be computed with the following query:

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

where $Split$ is an operator that breaks a table into multiple tables, i.e., $Split_{A_1,\ldots,A_m}(T) = \langle \pi_{A_1}(T), \ldots, \pi_{A_m}(T) \rangle$. Given a tuple set $T \subseteq V$, T's lineage tracing query is:

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

Proof: See [CWW97]. □

To trace the lineage of a view defined by multiple levels of ASPJ segments, we logically define an intermediate view for each segment, and then recursively trace through the hierarchy of intermediate views top-down. At each level, we use the tracing query for a one-level ASPJ view to compute lineage for the current traced tuples with respect to the views or base tables at the next level below. As discussed earlier, we can either materialize and maintain the intermediate views for the purpose of lineage tracing, or we can recompute the relevant intermediate results at tracing time. Because efficient incremental maintenance of multi-level aggregate user views generally requires materializing the same intermediate views we use for lineage tracing [Qua96], let us assume that the intermediate views are materialized.

Figure 6 presents the recursive tracing procedure. Given a view v in ASPJ canonical form and tuple $t \in v(D)$, procedure `Lineage(`$T = \{t\}, v, D$`)` computes the lineage of t 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 lineage $\langle V_1^*, \ldots, V_k^* \rangle$ in $\langle V_1, \ldots, V_k \rangle$ using the one-level view tracing query `TQ(`$T, v', \langle V_1, \ldots, V_m \rangle$`)` as in Theorem 3.5. It then computes (recursively) the lineage of each tuple set V_j^* according to v_j, $j = 1..k$, and concatenates the results to form the lineage of the entire list of view tuple sets.

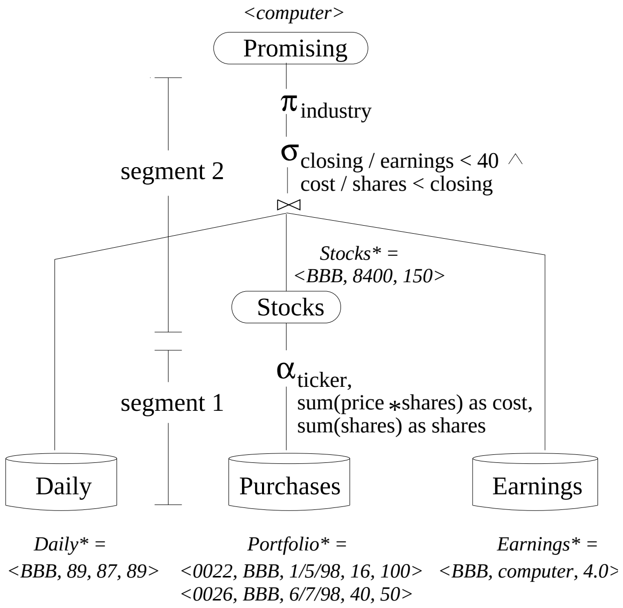

Figure 7: Tracing `Promising`

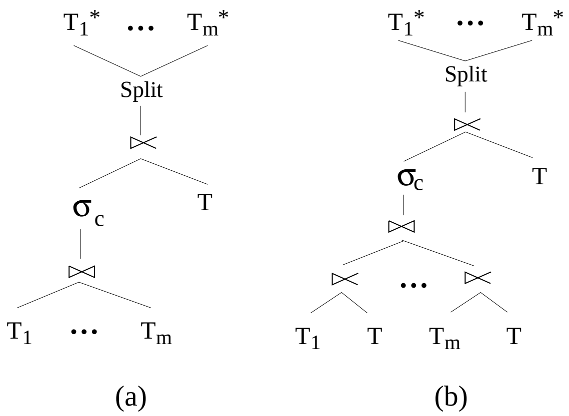

Figure 8: Tracing queries

Example 3.6 (Tracing the Lineage of `Promising`) Consider view `Promising` from Section 2. Recall that it has two ASPJ segments, illustrated in Figure 7. We define a materialized intermediate view `Stocks` corresponding to segment 1 as shown in Figure 7. Given view tuple $t = \langle \texttt{computer} \rangle$, we first compute t's lineage in tables `Daily`, `Stocks`, and `Earnings` using the lineage tracing query: $Split_{\texttt{Daily,Stocks,Earnings}}(\sigma_C(\texttt{Daily} \bowtie \texttt{Stocks} \bowtie \texttt{Earnings}))$ where the selection condition C is ($\texttt{price/earnings} < 40 \wedge \texttt{cost/shares} < \texttt{closing} \wedge \texttt{industry} =$ "`computer`"). The result of this first tracing query is $\langle \texttt{Daily}^*, \texttt{Stocks}^*, \texttt{Earnings}^* \rangle$ as shown in Figure 7. Since `Stocks` is an intermediate view, we then further trace the lineage of `Stocks`* in source table `Purchases`, obtaining `Purchases`* in Figure 7. Concatenating `Daily`*, `Purchases`*, and `Earnings`*, we obtain the final lineage result that was shown in Figure 4. □

3.3 Optimizations

There are some obvious improvements that we can make to our basic algorithm. For example, selection conditions or semijoins in the tracing query can be pushed below the join operator, which significantly reduces tracing cost in many cases. If the user view contains a key for each base table, we can use the keys in the traced view tuple to fetch its lineage directly from the base tables without performing any joins. Finally, although the contents of intermediate α results (e.g., `Stocks`) are needed in general to trace the lineage of a multi-level ASPJ view, in the case where the user view contains all group-by attributes of an intermediate α node, we can trace the view's lineage through that node without maintaining or recomputing the intermediate result.

4 Auxiliary Views for Lineage Tracing

Recall that in a distributed multi-source warehousing environment, querying the sources for lineage information can be difficult or impossible: sources may be inaccessible, expensive to access, expensive to transfer data from, and/or inconsistent with the views at the warehouse. By storing auxiliary views in the warehouse we can reduce or entirely avoid source queries during lineage tracing. In

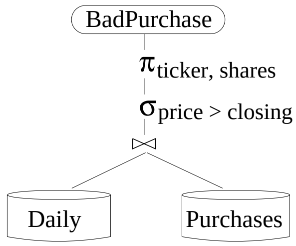

Figure 9: View definition for `BadPurchase`

ticker	shares
AAA	100
CCC	300

Figure 10: Contents of view `BadPurchase`

addition, as we saw in Section 3, auxiliary views corresponding to intermediate *ASPJ segments* in the view definition can be useful for efficient lineage tracing.

As will be seen in the following subsections, there is a wide variety of possible auxiliary views to maintain, with different performance tradeoffs in different settings. The remainder of this paper focuses on auxiliary view schemes and their relative performance for the restricted case of SPJ views. As future work we will first extend our results to one-level ASPJ views, which we expect to be relatively straightforward, and then to the full generality of multi-level views.

Given an SPJ view $V = v(D) = \pi_A(\sigma_C(T_1 \bowtie \cdots \bowtie T_m))$ and a tuple set $T \subseteq V$ to be traced, Figure 8(a) shows the generic form of its tracing query from Theorem 3.5.[2] We assume that all local selection conditions in the view—conditions that involve a single base table—are pushed down to the T_i's, so σ_C contains join conditions only. Since the size of T tends to be small, in some cases we also push down the semijoin and rewrite the tracing query as in Figure 8(b). The auxiliary views we consider are based on the forms of these two query trees. (Of course since the traced tuple set T is not available until tracing time, we cannot define or maintain auxiliary views on subqueries involving T.) We propose seven schemes for storing auxiliary views to support tracing the lineage of T according to v. For each scheme we specify the lineage tracing procedure, as well as the maintenance procedures for the auxiliary views and the original view, since they are all factors in overall performance.

In the maintenance procedures, we use δ to denote the delta tables (as discussed in Section 2), but with insertions and deletions combined, and we use $\uplus$ to denote application of the delta tables [GMS93]. We refer to the original view v as the *user view* when we need to distinguish it from the auxiliary views. As a running example, we use a simple SPJ view `BadPurchase` defined in Figure 9, where `Daily` and `Purchases` are the same source tables introduced in Section 2. `BadPurchase` contains all purchases where the purchase price was higher than the current closing price, including the stock ticker and the number of shares in the purchase. Figure 10 shows the view contents over our sample source data.

4.1 Store Nothing ($\varnothing$)

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

1. Auxiliary views: None
2. Lineage tracing: $TQ_{T,v} = Split_{\mathbf{T_1},\ldots,\mathbf{T_m}}(\sigma_C((T_1 \ltimes T) \bowtie \cdots \bowtie (T_m \ltimes T)) \ltimes T)$
3. Maintenance of auxiliary views: None
4. Maintenance of v [GMS93]: $\delta V = \pi_A(\sigma_C(\delta T_1 \bowtie (T_2 \uplus \delta T_2) \bowtie \cdots \bowtie (T_m \uplus \delta T_m) \uplus T_1 \bowtie \delta T_2 \bowtie (T_3 \uplus \delta T_3) \bowtie \cdots \bowtie (T_m \uplus \delta T_m) \uplus \cdots \uplus T_1 \bowtie \cdots \bowtie T_{m-1} \bowtie \delta T_m))$

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

ticker	high	low	closing	tranID	date	price	shares
AAA	29	25	26	0021	1/5/98	40	100
CCC	75	74	74	0026	7/2/98	80	300

Figure 11: Lineage view (LV)

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

4.2 Store Base Tables (BT)

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

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

Storing base tables can improve user view maintenance as well as lineage tracing, and maintaining the base tables is fairly easy. However, base tables can be large, even after applying local selections, and much of the source data may be irrelevant to any view tuple's lineage if joins are selective. In our example, base tables for view `BadPurchase` are simply copies of tables `Daily` and `Purchases`.

4.3 Store Lineage Views (LV)

An alternative way of improving tracing query performance is to store an auxiliary view based on the left subtree in Figure 8(a), which we call the *lineage view (LV)* for v, since it contains all lineage information for all tuples in the user view.

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

The LV scheme significantly simplifies the tracing query and thus reduces tracing query cost. However, lineage views can be large and are usually expensive to maintain. On the other hand, like base tables, lineage views can be helpful in maintaining the user view. Figure 11 shows the contents of the lineage view for `BadPurchase`.

4.4 Store Split Lineage Tables (SLT)

For views whose joins are many-to-many, lineage views as defined in Section 4.3 can be very large, and thus not efficient when performing the semijoin with T during lineage tracing. One solution is to split the lineage view and store a set of tables instead, which we call the *split lineage tables (SLTs)*. Note that we use lineage view LV as defined in Section 4.3 in the following definitions.

SLT-Daily

ticker	high	low	closing
AAA	29	25	26
DDD	75	74	74

SLT-Purchases

tranID	ticker	date	price	shares
0021	AAA	1/5/98	40	100
0026	CCC	7/2/98	80	300

Figure 12: Split lineage tables (SLTs)

PBT-Daily

ticker	high	low	closing
AAA	29	25	26
CCC	75	74	74

PBT-Purchases

tranID	ticker	date	price	shares
0021	AAA	1/5/98	40	100
0023	CCC	3/24/98	70	300
0026	CCC	7/2/98	80	300

Figure 13: Partial base tables (PBTs)

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

Split lineage tables contain no irrelevant source data, since every tuple in SLT_i, $i = 1..m$, contributes to some view tuples. Furthermore, the size of the split lineage tables can be much smaller than the lineage view. Their maintenance cost is similar to that of the lineage view. Note that although we do not materialize the lineage view LV in the SLT scheme, we still compute δLV, in order to maintain the user view V and auxiliary views SLT_i, $i = 1..m$. The disadvantage of SLT is that lineage tracing queries may be more expensive. Figure 12 shows the contents of the split lineage tables for `BadPurchase`. Since in our example the join is one-to-one, there is little advantage to the SLT scheme over LV in this case.

4.5 Store Partial Base Tables (PBT)

Reconsidering the BT scheme (Section 4.2), another way to reduce the size of the base tables is to store the semijoin of each source table T_i with the user view V; we call this semijoin result the *partial base table (PBT)* for T_i according to v.

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

For views with selective join conditions, the PBT scheme replicates much less source data than the BT scheme, with several benefits: It reduces the storage requirement, as well as the cost of refreshing the auxiliary views. It also reduces the tracing cost, because the tracing query operates on a much smaller table. However, partial base tables do not help with the maintenance of the user view. Instead, the user view needs to be maintained first. The partial base tables are then relatively cheap to maintain based on the user view's contents and changes. Figure 13 shows the contents of the partial base tables for `BadPurchase`.

4.6 Storing Base Table Projections (BP)

When source tables have known keys, we can store in our auxiliary views key attributes from the source tables together with other necessary attributes, which we call the *base table projections (BPs)*.

BP-Daily

ticker	closing
AAA	26
BBB	89
CCC	74
DDD	120
EEE	70
FFF	12
...	...

BP-Purchases

tranID	ticker	price	shares
0021	AAA	40	100
0022	BBB	16	100
0023	CCC	70	300
0024	BBB	40	50
0025	DDD	80	200
0026	CCC	80	300
...	...	...	...

Figure 14: Base table projections (BPs)

ticker	tranID	shares
AAA	0021	100
CCC	0026	300

Figure 15: Lineage view projection (LP)

This scheme improves tracing query performance (over storing nothing) while reducing view maintenance and storage costs (over storing full source replicas).

1. Auxiliary views: $BP_i = \pi_{A_i}(T_i)$, where A_i includes the key attributes K_i, attributes that are projected into V ($\mathbf{T_i} \cap \mathbf{V}$), and attributes involved in v's join conditions ($\mathbf{T_i} \cap \mathbf{C}$)
2. Lineage tracing: $T_i{}^* = T_i \ltimes (\sigma_C((BP_1 \ltimes T) \bowtie \cdots \bowtie (BP_m \ltimes T)) \ltimes T)$, $v^{-1}{}_D(T) = \langle T_1{}^*, \ldots, T_m{}^* \rangle$
3. Maintenance of auxiliary views: $\delta BP_i = \pi_{A_i}(\delta T_i)$, $i = 1..m$
4. Maintenance of v: Same as scheme $\varnothing$ replacing T_i with BP_i, $i = 1..m$

Note that the semijoins in the tracing procedure are key-based. This scheme can be especially useful when a source table has wide tuples but the view projects only a small fraction. During lineage tracing, the stored information identifies by key which source tuples really contribute to a given view tuple, then the detailed source information is fetched from the source using the key information. Maintenance of the user view is easy. However, in the BP scheme we do need to query the sources, which has its drawbacks as discussed earlier. Figure 14 shows the contents of the base table projections for `BadPurchase`.

4.7 Storing Lineage View Projections (LP)

Again assuming base tables with known keys, we can store a projection over the lineage view (Section 4.3) that includes only base table keys and user view attributes. We call this view the *lineage view projection (LP)*. Note that we use lineage view LV as defined in Section 4.3 in the following definitions.

1. Auxiliary views: $LP = \pi_{A \cup K_1 \cup \cdots \cup K_m}(LV)$, where A is the set of attributes in V, and K_i is the set of key attributes of table T_i, $i = 1..m$
2. Lineage tracing: $T_i{}^* = T_i \ltimes (LP \ltimes T)$, $v^{-1}{}_D(T) = \langle T_1{}^*, \ldots, T_m{}^* \rangle$
3. Maintenance of auxiliary views: $\delta LP = \pi_{A \cup K_1 \cup \cdots \cup K_m}(\delta LV)$, $i = 1..m$
4. Maintenance of v: $\delta V = \pi_A(\delta LP)$

Compared with the BP scheme, the LP scheme further simplifies the tracing query and improves tracing performance. However, the maintenance cost for the lineage view projection is higher than for the base table projections. LP also requires a source query as the last step of the tracing process, with the disadvantages previously discussed. Figure 15 shows the contents of the lineage view projection for `BadPurchase`.

4.8 Self-Maintainability and Self-Traceability

As mentioned in Section 4.2, self-traceable views can be traced correctly even if the sources are inaccessible or inconsistent with the warehouse views. Analogously, view self-maintainability as in-

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

Table 1: Scheme self-traceability and self-maintainability

troduced in Section 2 ensures that views can be maintained without querying the sources [QGMW96]. In cases where the sources are inaccessible, we must ensure that the user view together with our auxiliary views are both self-traceable and self-maintainable. Table 1 summarizes these properties with respect to the seven schemes introduced so far. We also consider self-maintainable extensions of three of the schemes, LV, SLT, and PBT, calling the extensions LV-S, SLT-S, and PBT-S.

5 Performance Study

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

5.1 System Model

Table 2 summarizes the configuration parameters for a simple warehousing system with the architecture in Figure 1. We again study SPJ views only, and consider two types of operations on the view: lineage tracing queries and view maintenance. Most entries in the table are self-explanatory. We assume all local selection conditions are pushed down to corresponding source tables and have been incorporated in the source table size. For simplicity, we assume that all source tables in the view have the same statistics. We also assume that the warehouse and the source databases have the same data block size and the same disk access cost. Finally, we assume that all sources can perform simple SPJ operations, and all join operations are nested-loop index joins. We consider table scans as well as key-based index lookups. We use the base values in the table as the baseline setting for our experiments, varying relevant parameters one at a time.

5.2 Cost Model

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

lineage tracing (resp. view maintenance) time
= disk_cost * # of disk I/Os in lineage tracing (resp. in view change computation and refresh)
+ trans_cost * # of bytes transmitted in lineage tracing (resp. in view maintenance)
+ msg_cost * # of network messages in lineage tracing (resp. in view maintenance)

Parameter name	*Description*	*Base value*	*Variation range*
workload			
query_ratio	# of tracing queries / total # of operations	0.8	$0 \sim 1$
query_size	# of tuples traced per query	10	–
view parameters			
rel_num	# of tables in the view	3	$1 \sim 8$
join_ratio	# of joining tuples / # of tuples in cross-product	0.000125	$0.0001 \sim 0.0005$
select_ratio	# of selected tuples / # of tuples before selection	0.1	–
proj_ratio	# of bytes projected into view / tuple size in bytes	0.2	–
source parameters			
tuple_num	# of tuples in source tables	$10,000$	$1000 \sim 50,000$
tuple_size	tuple size of source tables (in bytes)	1000	–
update_size	# of changed tuples per source table update	10	–
trans_cost	network transmission cost (in ms/byte)	0.2	–
msg_cost	network setup cost (in ms/message)	100	–
database configuration			
block_size	# of bytes in a block at warehouse and source	8000	–
disk_cost	cost to read/write a disk block (in ms/block)	10	–

Table 2: System model parameters

We also consider total tracing query and view maintenance time based on the query/update ratio. Finally, we consider the size of the user view and auxiliary views, which compares the schemes' warehouse storage requirements.

5.3 Experiments and Results

We present a sample of five experiments addressing the questions raised at the beginning of this section. For each experiment we simulated a total of 1000 operations. Each operation is either a lineage tracing query (tracing a set of view tuples) or view maintenance, which computes and applies changes to the user view and auxiliary views based on a set of source changes. We first look at the overall performance of our schemes using the base settings of Table 2, investigating how their cost distributes among the relevant cost components. We then study the impact on the various schemes of source table size, number of source tables, view selectivity, and query/update ratio. Because we measure all ten schemes in our experiments, some graphs are admittedly difficult to decipher, so we highlight the most important aspects of our results in text.

5.3.1 Cost Distribution

Our first experiment compares the performance of our ten proposed schemes under the base settings. Figure 16 shows the cost distributions divided into two parts: the tracing query cost on the right and the view maintenance cost on the left. Figure 17 shows the storage requirement of each scheme, including the user and auxiliary views. (The user view is fairly selective under our base settings, resulting in the high variance in storage requirement.) From Figure 16 we see that under our base settings, LV-S and SLT-S achieve low lineage tracing cost as well as fairly low total cost. $\emptyset$ (NOTH) has the highest tracing cost but low maintenance cost as expected, while conversely LV and SLT have low tracing cost but high maintenance cost. BT and PBT are reasonable compromises between the two extremes. The self-maintainable extensions (LV-S, SLT-S, and PBT-S) significantly reduce

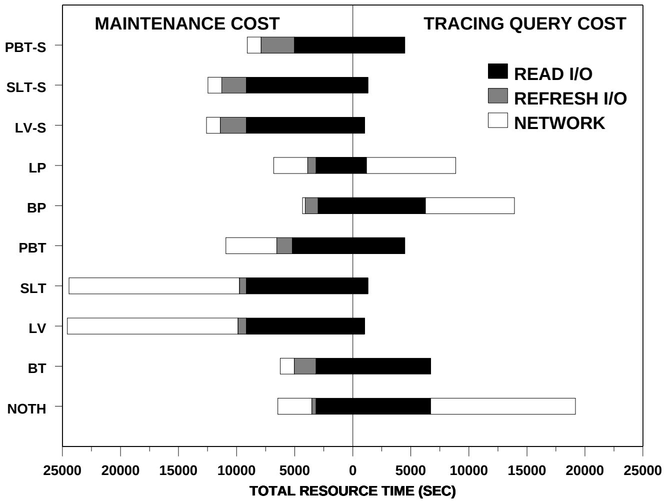

Figure 16: Cost distribution under base settings

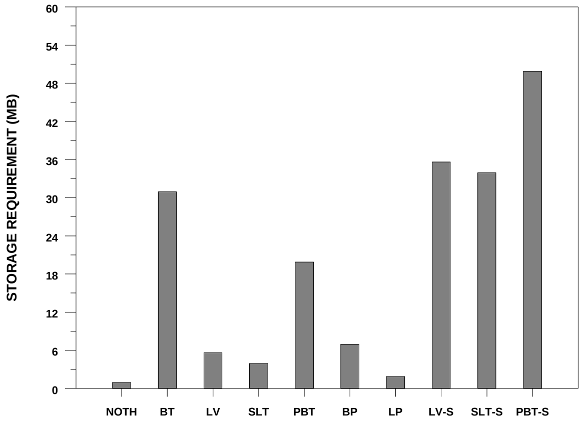

Figure 17: Storage cost

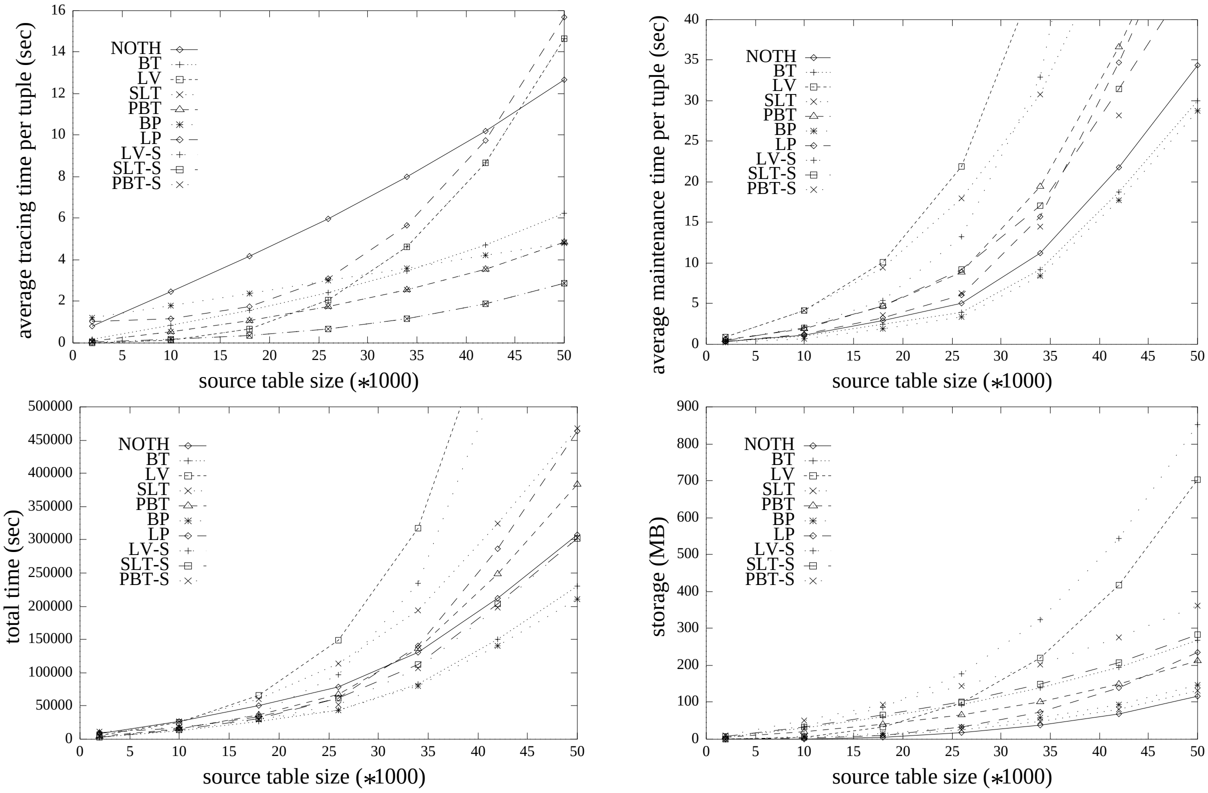

Figure 18: Impact of source table size

the network cost for maintenance, at the expense of higher refresh I/O cost and higher storage requirements (Figure 17). The projection-based schemes (BP and LP) achieve low tracing and maintenance I/O cost, but since they require queries to the sources the network cost remains high.

5.3.2 Impact of Source Table Size

Next, we look at how our schemes are affected by source table size scale-up. In Figure 18, we vary the size of each source table from 1000 to 50,000 tuples, and study the impact on tracing query

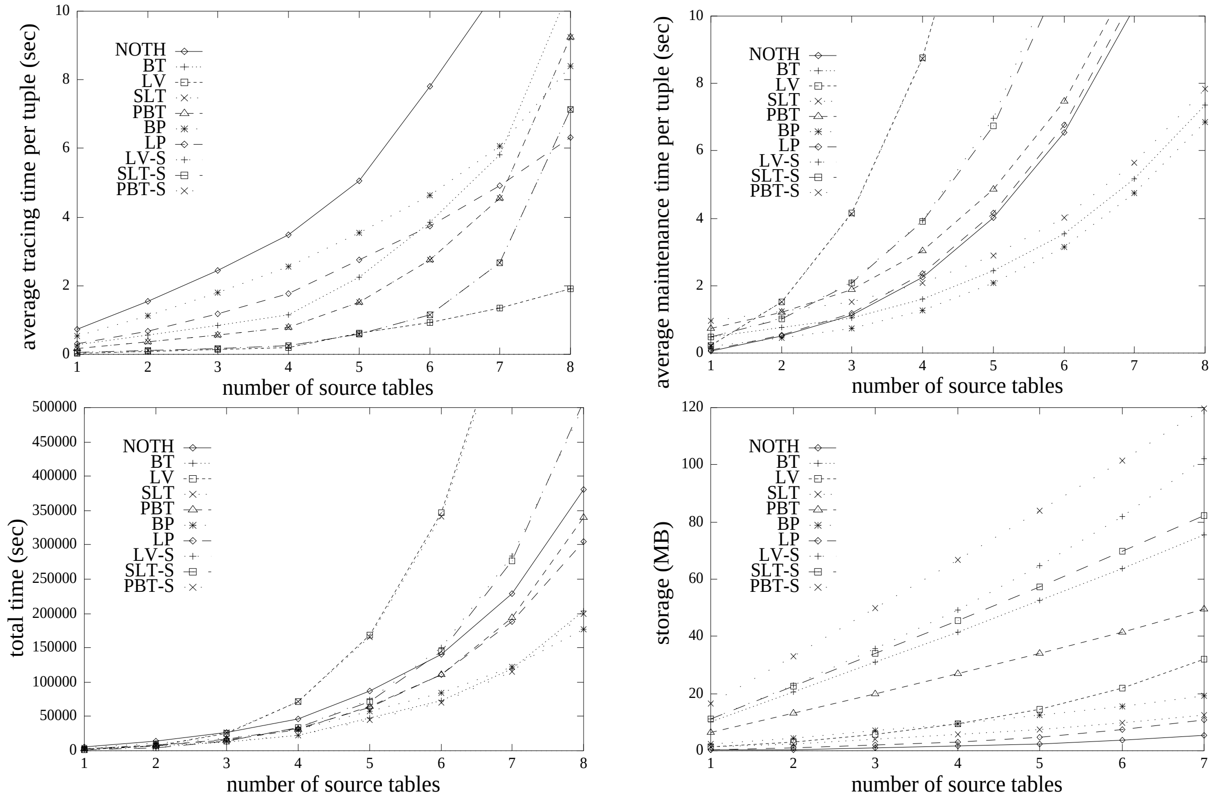

Figure 19: Impact of number of source tables

and maintenance costs as well as on the storage requirement. The x-axis represents the number of tuples in each source table and the y-axis represents the relevant costs. The results tell us that as the source table size increases, SLT and SLT-S provide the lowest tracing cost (the cost is identical for the two schemes), while BP incurs the lowest maintenance and total time costs. SLT and BP both have relatively low storage requirements.

5.3.3 Impact of Source Table Number

We now consider scale-up in the number of source tables. From the results in Figure 19, we observe that LV-S, which performed poorly in lineage tracing when scaling up source table size, presents the best tracing performance in source number scale-up. PBT-S, BT, and BP incur the lowest total cost as the number of source tables increases; ∅, LP, and SLT have the lowest storage requirement.

5.3.4 Impact of View Join Selectivity

This experiment studies how the join selectivity of the user view affects the schemes. Figure 20 shows the results. The x-axis represents the view join ratio, whose value varies from 0.0001 to 0.0005. We can see that the tracing performance of LV, LV-S, and LP degrades substantially as the view join ratio increases, while other schemes are much less sensitive. BT, BP, SLT-S, and PBT-S incur significantly lower total cost than other schemes in this experiment.

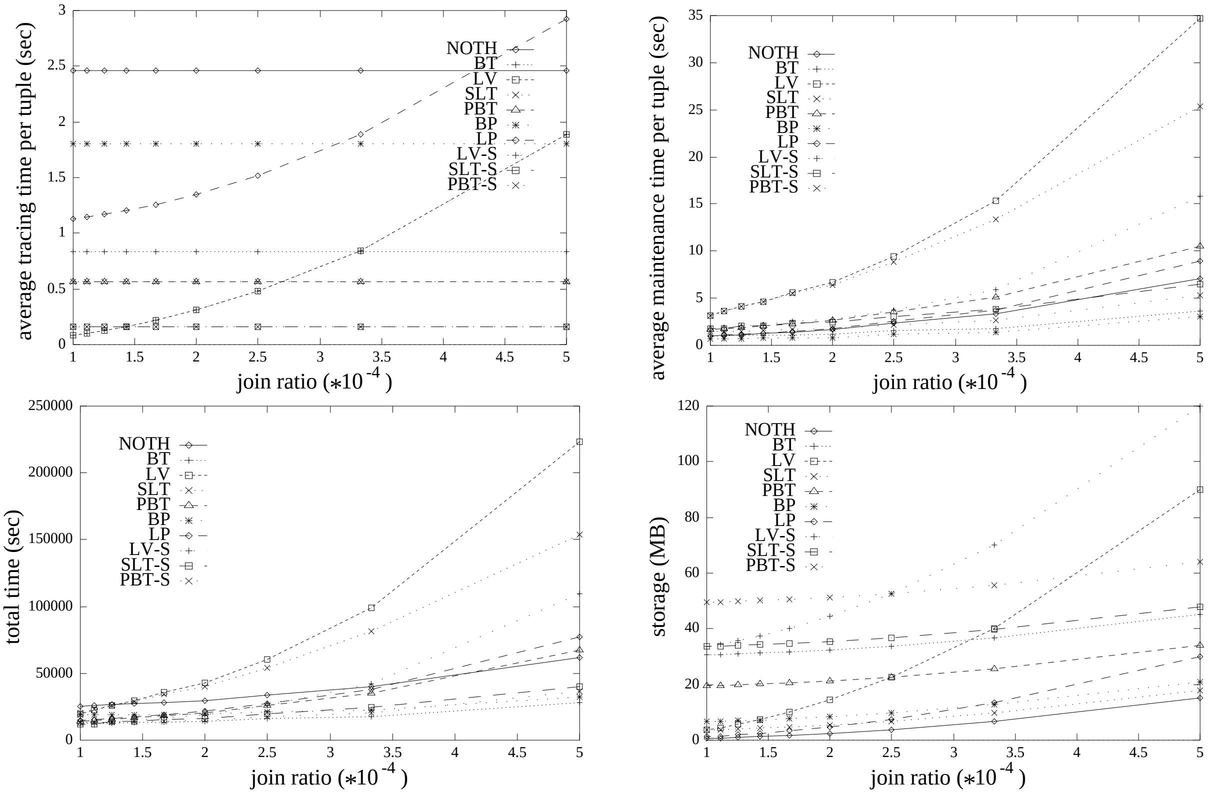

Figure 20: Impact of view join selectivity

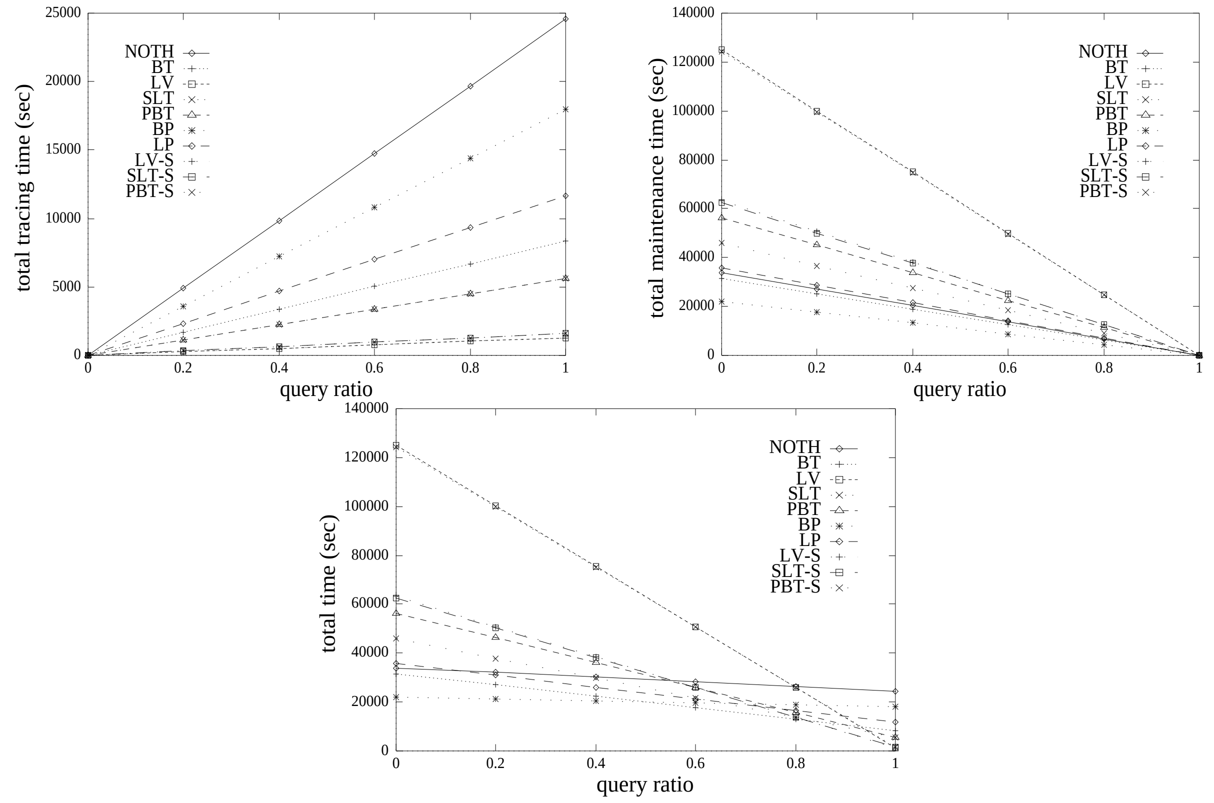

Figure 21: Impact of workload pattern

5.3.5 Impact of Workload Pattern

Our final experiment studies the impact of the workload ratio of tracing queries to view maintenance. In Figure 21, the x-axis varies the ratio from one extreme where the query ratio = 0 (no tracing queries) to the other extreme where the query ratio = 1 (no view maintenance). According to the total cost, LV-S and SLT-S are preferred for high query ratios, BT is preferred for medium query ratios, and BP is preferred for low query ratios.

6 Conclusion and Future Work

We defined the view data lineage problem and presented a lineage tracing algorithm for relational aggregate-select-project-join (ASPJ) views. In brief, given an ASPJ view, to trace its lineage we first transform the view definition into a canonical form, then generate a tracing procedure (Figure 6). For simple (one-level) ASPJ views, the procedure is one tracing query. For more complex views, the algorithm applies tracing queries recursively through the view definition tree. All tracing queries can be optimized using known techniques such as pushing down selections, and we can further optimize the tracing procedure for certain types of views. (Additional motivation and details of our tracing definitions and algorithms can be found in [CWW97], which also extends the results to set operators—including difference—and duplicate semantics.) For the restricted case of SPJ views, we designed several schemes for storing auxiliary views that enable and improve the performance of lineage tracing and view maintenance in a warehousing environment. We compared the schemes through simulations, identifying which schemes perform best in different settings.

Our results can be used as the basis for an efficient data warehouse analysis and debugging tool, by which analysts can browse their views, then "drill through" to the underlying source data that produced view data items of interest. We have implemented such a tool based on our lineage tracing algorithms in the WHIPS prototype data warehousing system at Stanford.

Future work includes the following.

1. Extend our auxiliary view schemes and performance study to cover arbitrary ASPJ views.
2. Extend our techniques beyond relational views. In particular, we are studying the types of data transformations used in commercial data warehouses, and we plan to extend our lineage tracing framework to acommodate them.
3. Lineage tracing is particularly compelling when the lineage of a view tuple may involve not only current source data, but also perhaps historical source data, or source data from previous database versions. We plan to explore extensions of our work along these lines, considering the problem of lineage tracing through versions and/or history.
4. View data lineage as defined in this paper explains how certain base relation tuples cause certain view tuples to exist. Sometimes, an erroneous view tuple may exist not because of erroneous base tuples in its lineage, but because some base tuples are missing; we would like to provide support for identifying such cases.
5. We believe that lineage tracing can be used to help solve related problems such as view schema evolution and view update. Initial ideas are presented in [CWW97].

Acknowledgements

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

References

[BB99] P. Bernstein and T. Bergstraesser. Meta-data support for data transformations using Microsoft Repository. *IEEE Data Engineering Bulletin, Special Issue on Data Transformations*, 22(1):9–14, March 1999.

[CD97] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Record*, 26(1):65–74, March 1997.

[CWW97] Y. Cui, J. Widom, and J.L. Wiener. Tracing the lineage of view data in a warehousing environment. Technical report, Stanford University Database Group, November 1997. 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, Germany, September 1978.

[DB2] IBM Corporation: DB2 OLAP Server. http://www.software.ibm.com/data/db2/db2olap/.

[FJS97] C. Faloutsos, H.V. Jagadish, and N.D. Sidiropoulos. Recovering information from summary data. In *Proc. of the Twenty-Third International Conference on Very Large Data Bases*, pages 36–45, Athens, Greece, August 1997.

[GBLP96] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In *Proc. of the Twelfth International Conference on Data Engineering*, pages 152–159, New Orleans, Louisiana, February 1996.

[GMR95] A. Gupta, I.S. Mumick, and K.A. Ross. Adapting materialized views after redefinitions. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 211–222, San Jose, California, May 1995.

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

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

[HGMW+95] J. Hammer, H. Garcia-Molina, J. Widom, W. Labio, and Y. Zhuge. The Stanford Data Warehousing Project. *IEEE Data Engineering Bulletin, Special Issue on Materialized Views and Data Warehousing*, 18(2):41–48, June 1995.

[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.

[IK93] W.H. Inmon and C. Kelley. *Rdb/VMS: Developing the Data Warehouse.* QED Publishing Group, Boston, Massachussetts, 1993.

[Imp] Cognos: Impromptu Interactive Database Query and Reporting Tool. http://www.cognos.com/impromptu/.

[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.

[LBM98] T. Lee, S. Bressan, and S. Madnick. Source attibution for querying against semi-structured documents. In *Workshop on Web Information and Data Management*, pages 33–39, Washington, DC, November 1998.

[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.

[LW95] D. Lomet and J. Widom, editors. *Special Issue on Materialized Views and Data Warehousing,* IEEE Data Engineering Bulletin 18(2), June 1995.

[Pow] Cognos: PowerPlay OLAP Analysis Tool. http://www.cognos.com/powerplay/.

[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.

[RSS96] K.A. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 447–458, Montreal, Canada, June 1996.

[Sto75] M. Stonebraker. Implementation of integrity constraints and views by query modification. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 65–78, San Jose, California, May 1975.

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

[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.

[ZGMHW95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 316–327, San Jose, California, May 1995.