2007

Cost Effective Forward Tracing Data Lineage

Mingwu Zhang

Xiangyu Zhang
Purdue University, xyzhang@cs.purdue.edu

Sunil Prabhakar
Purdue University, sunil@cs.purdue.edu

Report Number:
07-004

Cost Effective Forward Tracing Data Lineage

Mingwu Zhang
Xiangyu Zhang
Sunil Prabhakar

CSD TR #07-004
March 2007

Cost Effective Forward Data Lineage Tracing

Mingwu Zhang Xiangyu Zhang and Sunil Prabhakar
Department of Computer Science, Purdue University
West Lafayette, IN 47907-1398, USA
{mzhang2, xyzhang, sunil}@cs.purdue.edu

Abstract

Data lineage plays a critical role in verifying data correctness in scientific databases and data warehousing. In this paper, we clearly define forward data lineage in bag semantics and show its properties. We propose a tracing method that piggybacks normal query evaluation. Our method effectively supports aggregation and variable granularity lineage. More importantly, it features cost effective tracing through sophisticated implementation. The implementation, based on roBDDs, exploits the clustering *and* overlapping *characteristics of data lineage to significantly reduce the overhead. Prior lineage tracing techniques compute lineage backward. More specifically, given a query that produces a view, the lineage is computed by executing the reverse query. While such techniques are believed to be very efficient due to the lazy computation, they are indeed inferior to the proposed forward method in most aspects.*

1 Introduction

Data lineage provides a mapping from the *view* data to the *base* data and thus plays a critical role in many database applications.

In scientific databases, with the advance of high-throughput experimental technology, scientists are tackling large scale experiments which produce enormous amounts of data. Consequently, the need to automatically process and validate data becomes more and more pressing. Data lineage, or data *provenance* [12, 10] greatly simplifies such a procedure. For example in biochemistry computing, the post-experiment data processing is often quite complicated. Data lineage tracing can faithfully record such a complicated procedure. An abnormal data lineage usually implies the result data is not trustworthy, thus expensive wet-bench experiments should not be planned based upon these unreliable data.

In data warehousing, *views* over source data are often computed and materialized in order to answer queries about the source data. More sophisticated analysis such as data mining can be applied to make predictions. In the construction of a data warehouse, the data cleaning procedure often takes more than half of the total time. In this procedure, a key problem is to handle inconsistent or missing values in the data, which is usually resolved by replacing those values with the most likely values or values derived from a model. These *substituted* values are not as reliable as the normal data, but they are nonetheless consumed to produce analytic results. Without data lineage, we lose the capability of tracing back to these unreliable sources. Other applications include view maintenance such as the well known *incremental view update* problem [13, 9].

As data reliability is becoming a prominent issue in databases, the need of developing efficient lineage tracing techniques surfaces. The existing works can be divided into two categories. The first category is *backward*, or *lazy*, approaches [5, 6]. In this category, reverse queries are constructed to compute a subset of the base tuples with respect to a result tuple and the base table themselves. The computed base tuples contribute to the result tuple. Since the lineage is computed on demand and by a reverse method, it is called the *lazy* or *backward* approach. The lineage computed by these techniques are also called *backward* lineage in this paper. The second category is *forward*, or *eager*, approaches [3]. The lineage computed by these approaches is called *forward* lineage. In this category, lineage is computed during the query evaluation such that the lineage for each result tuple is computed, stored and ready to be queried.

Backward techniques have the advantage of computing lineage on demand so that lineage is not explicitly computed and stored, and thus requires less space. However, there are several drawbacks of this approach.

Firstly, backward techniques cannot handle bag semantics which arise naturally in these applications. Secondly, if aggregation operations or nested queries are involved, backward computation often requires recomputing or materializing intermediate views, even for computing the lineage of a single result tuple. Thus, this may entail a reverse query that is as expensive as the original query itself. Thirdly, revisions cannot be handled efficiently. A revision is a correction of a base tuple discovered after the base data has been used to generate derived data. Revisions are common in many applications (e.g. if some equipment is discovered to be faulty, or there is an error in the cleansed data). Data lineage is often motivated by the need to discover the impact of an incorrect data item in order to invalidate these results. In such cases, the backward approach is forced to compute the lineage for each and every tuple that could potentially be related to the invalidated base data. More details on these limitations can be found in Section 2.

Forward lineage tracing has the potential to address each of these limitations and drawbacks. Compared to backward approaches, forward techniques trace lineage for all tuples including intermediate tuples during query evaluation. Therefore, no additional query is needed. Handling revisions is straight-forward since the lineage is pre-computed and available to query. Bag semantics and aggregation can be correctly handled by a forward method. Our technique falls in the forward computation category. The state-of-the-art of forward methods is the work by Bhagwat et al. [3]. They propose an efficient annotation propagation technique which propagates attribute annotations from base tables to the result view. Theoretically, such an annotation management system can be extended to trace data lineage. However, the limitations of their technique, such as inability to handle aggregation and bag semantics, make it unsuitable for our purpose of general lineage tracing. Nonetheless, many of our query rewriting rules are inspired by their work.

This paper presents a cost-effective general forward lineage tracing technique. The contributions are:

- A formal definition of forward lineage that covers both set and bag semantics. The properties of forward lineage. A set of query rewriting rules for computing the forward lineage. These rules are capable of handling general views such as subqueries in the `where` clauses and aggregations.

- As pointed out by [3], the storage scheme is the challenge for forward methods. Based on the characteristics of lineage sets, we propose two schemes: *set* based and *Reduced Ordered Binary Decision Diagram* (RoBDD) based. In our technique, RoBDDs are used as a compact representation of sparse sets.

- Propose and implement variable-granularity lineage that is more meaningful for data with large lineage sets.

- An experimental evaluation (with implementation in PostgreSQL, TPC-H dataset and Walmart dataset) against backward approaches that validates the advantages of our approach.

The rest of the paper is organized as follows. Section 2 motivates our technique. Section 3 presents the formal definition of forward lineage, its properties, and the query rewriting rules. Section 4 discusses the storage scheme based on RoBDD. Section 5 presents the experimental results. Related work is discussed in section 6. Conclusions are given in section 7.

2 Motivation

This section discusses the limitations of the existing lineage tracing techniques. As we mentioned earlier, they can be divided into *forward* and *backward* methods. The methods discussed in [5, 6] belong to the backward category. The technique in [3] belongs to the forward category. The goal of the forward technique in [3] is essentially to propagate attribute annotations, which is not exactly general lineage tracing although it can be extended for that purpose. As a consequence, their solution is more tuned to the attribute level annotation propagation. In this sense, the backward methods are more closely related to our technique since they tackled the same problem as we do. Therefore, in this section and also the rest of our paper, the comparison with backward methods is the focus. Readers that are interested in the comparison with the existing forward methods are referred to the related work section.

The motivation examples are based upon a database that has three relations: `store`, `item`, and `sales` as shown in Table 1 2 3. They list the department stores, the items that are sold by these stores, and the sales records, respectively.

s_id	s_name	city	state
01	Target	Palo Alto	CA
02	Target	Albany	NY
03	Macy's	San Francisco	CA
04	Macy's	New York City	NY

Table 1. Store

i_id	i_name	category
01	binder	stationery
02	pencil	stationery
03	shirt	clothing
04	pants	clothing
05	pot	kitchenware

Table 2. Item

t_id	s_id	i_id	price	num
01	01	01	4	1000
02	01	02	1	3000
03	01	04	30	600
04	02	01	5	800
05	02	02	2	2000
06	02	04	35	800
07	03	03	45	1500
08	03	04	60	600
09	04	03	50	2100
10	04	04	70	1200
11	04	05	30	200

Table 3. Sales

2.1 Bag Semantics

The first example is about bag semantics. Many queries employ bag semantics for the sake of efficiency. Therefore, it is important to be able to trace lineage for bag semantics. Consider the query below, it produces the view in Table 4, in which there are duplicate tuples.

Q_1:

```
create table view
select d.s_name, i.i_name
from  store d, item i, sales s
where d.s_id = s.s_id and i.i_id = s.i_id and
     i.category = 'stationery'
```

s_name	i_name
Target	binder
Target	pencil
Target	binder
Target	pencil

Table 4. The Result Of Q_1

According to [6], the reverse query is constructed as Q_2. The execution of this query results in the lineage table in Table 5. We can see that the reverse query is not able to distinguish *where* the first tuple $< Target, binder >$ in Table 4 comes from. Therefore, it has to assume that the tuple originates from a pool of tuples in the three base tables with s_id as {01,02} and i_id as {01}, which can be seen from the tuples with $s\_name = Target$ and $i\_name = binder$ in Table 5. According to relational semantics, we know the two $< Target, binder >$ tuples must actually result from two unique base tuple subsets, respectively. Knowing exactly where does each tuple come from could be very helpful in data verification because that saves the analyst from inspecting the entire lineage pool if she can localize the problematic tuple, which is often feasible in practice by looking at information such as the order of the result tuples.

Q_2 (reversing Q_1):

```
select v.s_name, v.i_id, d.s_id, i.id
from  store d, item i, sales s, view v
where d.s_id = s.s_id and i.i_id = s.i_id and
     i.category = 'stationery' and
     v.s_name = d.s_name and
     v.i_name = i.i_name
```

s_name	i_name	s_id	i_id
Target	binder	01	01
Target	binder	02	01
Target	pencil	01	02
Target	pencil	02	02
Target	binder	01	01
Target	binder	02	01
Target	pencil	01	02
Target	pencil	02	02

Table 5. The Result Of Q_2

In our system, the query Q_1 is rewritten as Q_3 below. The instrumentation is highlighted.

Q_3 (translated from Q_1):

```
select d.s_name, i.i_name,
        {d.s_id} U {i.i_id} U {s.t_id} As DL
from  store d, item i, sales s
where d.s_id = s.s_id and i.i_id = s.i_id and
     i.category = 'stationery'
```

The result view becomes Table 6. The *DL* column presents the lineage of each tuple. For example, the two $< Target, binder >$ tuples have the lineage of {d.01, i.01, s.01} and {d.02, i.01, s.04}, respectively. Note that we use the key as the lineage for each tuple in the base tables. A more general implementation scheme is to use a global $id(t)$ function to assign unique ids to base tuples.

s_name	i_name	DL
Target	binder	{d.01, i.01, s.01}
Target	pencil	{d.01, i.02, s.02}
Target	binder	{d.02, i.01, s.04}
Target	pencil	{d.02, i.02, s.05}

Table 6. The Result Of Q_3

2.2 Materializing Intermediate Views

Another limitation of backward approaches is that they require recomputing or materializing intermediate views for ASPJ subqueries. For example, the below query Q_4 retrieves the stores and their sums of sales records for the items that are sold more than 5000 units. A reverse query can not be constructed to directly compute the lineage on the base tables without the intermediate table. According to [6], the subquery is first materialized by the query Q_5 to a table `intermediate_tab`. The query Q_6 is executed to reverse the result tuples to the base tuples using the table `intermediate_tab`.

Q_4:

```
create table my_tab as
select temp_tab.s_name, sum(temp_tab.total) from (
    select store.s_name, item.category,
                 sum(item.num) as total
    from  item, store, sales
    where sales.s_id = store.s_id and
                 item.i_id = sales.i_id
    group by  store.s_name, item.category
    having total > 5000 ) as temp_tab
group by  temp_tab.s_name
```

Q_5 (materialization):

```
create table intermediate_tab as
select store.s_name, item.category,
      sum(item.num) as total
from  item, store, sales
where sales.s_id = store.s_id
     and item.i_id = sales.i_id
group by  store.s_name, item.category
having total > 5000
```

Q_6 (reverse query of Q_4):

```
create table reverse as
select item.*, store.*, sales.*
from  my_tab t1, intermediate_tab t2,
     item, store, sales
where t1.s_name = t2.s_name
        and t2.category = item.category
        and item.i_id = sales.i_id
        and store.s_id = sales.s_id
        and t2.s_name = store.s_name
```

Based on our technique, the query Q_4 is translated to Q_7, in which `DLUNION` is a new aggregation operator that aggregates the lineage sets of all tuples that satisfy the grouping condition. Note that in the sub-query, because the tuples are first joined and then aggregated, the corresponding lineage operation is the composition of union and aggregation. One may raise the question that although our forward method avoids recomputation or materialization, it computes the lineage for all tuples including the intermediate ones, which may be expensive as well. As shown in later sections, our sophisticated design nicely addresses this concern.

Q_7 (translated Q_4):

```
select temp_tab.s_name, sum(total),
 DLUNION((temp_tab.DL)
from  (
   select d.s_name, i.category, sum(i.num) as total,
    DLUNION ({d.s_id}U{i.i_id}U{s.t_id}) As DL
   from  item i, store d, sales s
   where s.s_id = d.s_id and i.i_id = s.i_id
   group by  d.s_name, i.category
   having total > 5000
) as temp_tab
group by  temp_tab.s_name
```

In scientific databases, it happens that a base tuple is consumed and propagated to derived tables or views before it is eventually identified as being unreliable or even wrong by experiments. In such a case, a very desirable service would be to recognize the set of tuples in the derived tables/views that are affected by a particular source tuple. Backward approaches, by their nature, are inefficient for such a demand. The only solution is to compute the backward lineage for each result tuple and then check if the problematic base tuple is in the lineage. Our forward technique can naturally meet such a requirement by only assigning a lineage id to the problematic tuple. In other words, all other base tuples have null lineage. After evaluating the query, the result tuples with non-empty lineage are the ones affected by the problematic tuple. If such a query reevaluation is infeasible due to the cost, one can still choose to compute the lineage for all tuples during the original query evaluation with our forward method.

3 Forward Data Lineage

Informally, the lineage we are interested in is the tuple subsets in the base tables which a result tuple is derived from. Assume each tuple in the base tables is identified by a unique number, the lineage of a result tuple is essentially a set of these id numbers. If a base tuple is updated, a new id is assigned to it.

The formal definition of data lineage with regard to relational operators is given as follows. The definition for each operator follows the format of $DL(t, Op(V_1, ..., V_M))$, meaning the data lineage of the result tuple t in the view defined by $Op(V_1, ..., V_M)$.

Here we consider only ASPJ queries, more general queries will be discussed in the last subsection.

Definition 1 (Forward Data Lineage)

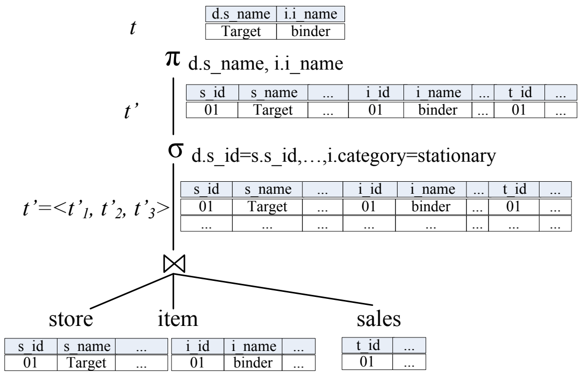
- **Base case**: *$DL(t, R) = \{id(t)\}$, in which $t \in R$ and R is a base table.*
- **Selection**: *$DL(t, \sigma_C(V)) = DL(t, V)$, in which $\sigma_C(V) = \{t \mid t \in V \texttt{ and } t \texttt{ satisfies condition } C\}$.*
- **Projection**: *$DL(t, \pi_A(V)) = DL(t', V)$, in which $\pi_A(V) = \{t'.A \mid t' \in V\}$.*
- **Join**: *$DL(t, \bowtie_\theta (V_1, ..., V_m)) = DL(t_1, V_1) \cup DL(t_2, V_2) \cup ... \cup DL(t_m, V_m)$, in which $\bowtie_\theta (V_1, ..., V_m) = \{< t_1, ..., t_m > \mid t_i \in V_i \texttt{ for } i = 1...m, \texttt{and the } t_i' \texttt{ s satisfy condition } \theta\}$.*
- **Aggregation**: *$DL(t, \alpha_{G,aggr(B)}(V) = \bigcup_{t' \in T} DL(t', V)$, in which $\alpha_{G,aggr(B)}(V) = \{< T.G,\ aggr(T.B) > \mid T \subseteq V,\ \forall tt, tt' \in T,\ \forall tt'' \notin T\ :\ tt'.G = tt.G \wedge tt''.G \neq tt.G\}$, G is a group-by attribute list from V, and $aggr(B)$ abbreviates a list of aggregation functions applied to attributes of V.*

The *base case* states that the lineage of a tuple t in a base table R contains a unique id assigned to this tuple by the global function $id(t)$. One may think that we actually turn bag semantics into set semantics by using the $id()$ function. It is true that using this function distinguishes all the duplicate tuples in the base table, but duplicates may occur in intermediate views, in which we no longer assign unique ids to tuples.

For *select* operations, the lineage of t in a view of $\sigma_C(V)$ is essentially the lineage of t in V because the fact that t satisfying C does not affect the lineage of t. In a later subsection, we will discuss how we handle the case that C is a query such that t's lineage is affected.

Similarly, for *project* operations, the lineage of t in a view of $\pi_A(V)$ is the lineage of t' in V, in which t is produced by projecting the attribute list A of t'.

For *join* operations, $t \in \bowtie_\theta (V_1, ..., V_m)$ has the form of $< t_1, ..., t_m >$ and t_i is the part of t from V_i, $i \in 1, ...m$. The lineage of t in the joined view is essentially the union of the lineages of t_i in V_i.



t
d.s_name | i.i_name
Target | binder
π d.s_name, i.i_name
t'
s_id | s_name | ... | i_id | i_name | ... | t_id | ...
01 | Target | ... | 01 | binder | ... | 01 | ...
σ d.s_id=s.s_id,…,i.category=stationary
t'=<t'1, t'2, t'3>
s_id | s_name | ... | i_id | i_name | ... | t_id | ...
01 | Target | ... | 01 | binder | ... | 01 | ...
... | ... | ... | ... | ... | ... | ... | ...
⋈
store
item
sales
s_id | s_name | ...
01 | Target | ...
i_id | i_name | ...
01 | binder | ...
t_id | ...
01 | ...

Figure 1. The Derivation of The First Tuple in Table 4

For *aggregation* operations, t is a tuple in the result view that is aggregated on an attribute list B with the group-by condition G. The lineage of t in the aggregated view is the union of the lineages of all the tuples that fall in the same group.

Note that the definition is recursive. More precisely, forward lineage is recursively defined based on the view derivation.

For example in Table 1, the first tuple $t =< Target, pencil >$ in Table 4 is derived by the query $v(\texttt{store}, \texttt{item}, \texttt{sales}) = \pi_{d.s\_name,i.i\_name}(\sigma_{d.s_id=...=category}(store\ d \bowtie item\ i \bowtie sales\ s)$. Let us assume t is projected by the π operation from the tuple $t' =< d.s\_id = 01, d.s\_name = Target, ..., i.i\_id = 01, ..., s.s\_id = 01, ..., s.sum = 1000 >$ in the intermediate view $v' = \sigma_{d.s_id=...=category}(store\ d \bowtie item\ i \bowtie sales\ s)$. According to the relational semantics, $t' =< t_1' =< d.s\_id = 01, ... >, t_2' =< i.i\_id = 01, ... >, t_3' =< s.s\_id = 01, ... >>$ is a tuple in $v'' = store\ d \bowtie item\ i \bowtie sales\ s$.

Based on the definition, $DL(t, v) = DL(t', v') = DL(t', v'') = DL(t_1', d) \cup DL(t_2', i) \cup DL(t_3', s) = \{id(d.01),\ id(i.01),\ id(s.01)\}$.

Since Definition 1 is based on the view derivation, which may be altered by semantic preserving transformations. It is critical for us to understand the algebraic properties of forward data lineage, which will be discussed by the following subsections.

3.1 SPJ Queries

We first consider SPJ (*Select-Project-Join*) queries, In order to simplify the formalization, we assume the view definition is a tree. In other words, if a relation is referenced more than once, such as self joins, we consider the two relation instances as different relations. The same simplification was adopted by Cui et al. in their backward tracing paper [6]. The generalization to graph definition is discussed at the end of this subsection.

Let $\widetilde{DL}(t, v(R_1, ..., R_k)) = < R_1^*, ..., R_k^* >$ be $DL(t, v(...))$ represented as subsets of the base tables R_1,..., R_k, in which v is the view and $tt \in R_x^*$ if and only if $id(tt) \in DL(t, v(...))$.

Theorem 1 (Completeness)
$v(\widetilde{DL}(t, v(R_1, ..., R_k)) = \{t\}$

This theorem says that the data lineage of t derives exactly t itself. This matches our intuition about what lineage is – a subset of base tuples that produce the result tuple. It can be proved by induction on the height of the view definition tree assuming the view being the root and the base tables being the leaves.

Let h be the height of the definition tree. **Proof :**

(1) For $h = 1$, meaning the view is produced by applying a relational operator on the base table(s):

(a) $\boldsymbol{v(R) = \sigma_C(R)}$
$DL(t, v) = DL(t, R) = \{id(t)\}$ according to Definition 1
$\Rightarrow$ $\widetilde{DL}(t, v) = \{t\}$ – (i)
$t \in v(R) \Rightarrow t$ satisfies C according to relational semantics – (ii)
$\sigma_C(\widetilde{DL}(t, v)) = \sigma_C(\{t\}) = \{t\}$ according to (i) and (ii).

(b) $\boldsymbol{v(R) = \pi_A(R)}$. The proof is similar to case (a).

(c) $\boldsymbol{v(R_1, ..., R_m) = \bowtie_\theta (R_1, ..., R_m)}$
$DL(t, v) = DL(t_1, R_1) \cup ... \cup DL(t_m, R_m) = \{id(t_1), ..., id(t_m)\}$ according to Definition 1
$\Rightarrow$ $\widetilde{DL}(t, v) = < \{t_1\}, ..., \{t_m\} >$ – (i)
$\Rightarrow$ $t \in v(R_1, ..., R_m) \Rightarrow t_i$'s satisfy θ according to relational semantics – (ii)
$\Rightarrow$ $\bowtie_\theta (\widetilde{DL}(t, v)) = \bowtie_\theta (\{t_1\}, ..., \{t_m\}) = < t_1, ...t_m > = \{t\}$ according to (i) and (ii).

(2) For $h = k$, we denote a view definition $v(...)$ with height x as $v^x(...)$.

(a) $\boldsymbol{v^k(R_1, ..., R_m) = \sigma_C(w^{<k}(R_1, ..., R_m)}$
$DL(t, v^k) = DL(t, w^{<k})$ according to Definition 1
$\Rightarrow$ $\widetilde{DL}(t, v^k) = \widetilde{DL}(t, w^{<k})$ – (i)
$\Rightarrow$ $t \in v^k(R_1, ..., R_m) \Rightarrow t$ satisfies C according to relational semantics – (ii)
$\Rightarrow$ $w^{<k}(\widetilde{DL}(t, w^{<k})) = \{t\}$ by induction assumption – (iii)
$\Rightarrow$ $v^k(\widetilde{DL}(t, v^k)) = \sigma_C(w^{<k}(\widetilde{DL}(t, v^{<k})))$
$=$ $\sigma_C(w^{<k}(\widetilde{DL}(t, w^{<k})))$ according to (i)
$=$ $\sigma_C(\{t\})$ according to (iii)
$=$ {t}

The remaining cases can be proved similarly.

Deriving t does not precisely describe the property of lineage since tables that subsume the lineage of t may derive t as well. The second theorem says that the lineage sets are the smallest sets that produce the result tuple. In other words, all the base tuples in the lineage sets contribute to the result tuple.

Theorem 2 (Minimality)
$\forall X \subset \widetilde{DL}(t, v(R_1, ..., R_k) : v(X) = \phi$

Similarly, it can be proved by induction on the height of the view definition tree. **Proof :**

(1) For $h = 1$,

(a) $\boldsymbol{v(R) = \sigma_C(R)}$
$\widetilde{DL}(t, v) = \{t\}$
$\Rightarrow$ $X = \phi$
$\Rightarrow$ $v(X) = \phi$

(b) $\boldsymbol{v(R) = \pi_A(R)}$. The proof is similar to case (a).

(c) $\boldsymbol{v(R_1, ..., R_m) = \bowtie_\theta (R_1, ..., R_m)}$
$\widetilde{DL}(t, v) = < \{t_1\}, ..., \{t_m\} >$ according to Definition 1
$\Rightarrow$ X has the form of $< \{t_1\}, ..., \phi, ..., \{t_m\} >$
$\Rightarrow$ $v(X) = \bowtie_\theta (X) = \phi$

(2) For $h = k$:

(a),(b) The proofs for these two cases are trivial.

(c) $\boldsymbol{v^k(R_1,...,R_m) = \bowtie_\theta (w_1^{<k}(R_1,...,R_m), ..., w_n^{<k}(R_1,...,R_m))}$

$DL(t, v^k) = DL(t_1, w_1^{<k}) \cup ... \cup DL(t_n, w_n^{<k})$ according to Definition 1

$\Rightarrow$ $\widetilde{DL}(t, v^k) = \widetilde{DL}(t_1, w_1^{<k}) \cup ... \cup \widetilde{DL}(t_n, w_n^{<k}) = < R_1^*, ..., R_n^* >$

$\Rightarrow$ X has the form of $< R_1^*, ..., Y \subset R_i^*, ..., R_n^* >$, let $tt \in Y$ and $tt \notin R_i^*$

$\Rightarrow$ tt must belong to some $\widetilde{DL}(t_j, w_j^{<k})$ – (i) Since the view definition is a tree, meaning no relations or intermediate views are referenced more than once, there is no overlap between the lineage sets of different w_x views. Therefore, $w_j^{<k}(X) = \bowtie_\theta (\widetilde{DL}(t_j, w_j^{<k}) - \{tt\}) = \phi$ by (i) and the induction assumption.

$\Rightarrow$ $v^k(X) = \bowtie_\theta (w_1^{<k}(X), ..., w_n^{<k}(X)) = \bowtie_\theta (..., \phi, ...) = \phi$

During query evaluation, various plans may be chosen for a query by the optimizer which results in different view derivations. Since our definition is based on the structure of the view definition, it is important to understand the effect of query transformations on lineage tracing. The next theorem states that query optimizations do not change the forward lineage for SPJ queries if the definition has a tree structure.

Theorem 3 (Equivalence With Tree Definition)

Given two equivalent SPJ views v_1 and v_2 s.t. $\forall D : v_1(D) = v_2(D)$, let $v_1'(D) = \{<t, DL(t, v_1(D))> | t \in v_1(D)\}$ and $v_2'(D) = \{< t, DL(t, v_2(D)) > | t \in v_2(D)\}$ in

$$v_1'(D) = v_2'(D)$$

In order to prove Theorem 3, we first prove the following lemmas.

Lemma 1 *Given two tuples t_1 and t_2 in the result view v, $DL(t_1, v) \neq DL(t_2, v)$.*

This lemma says different tuples, even though they may have the same value, always have distinct lineage sets. For example, the two duplicate tuples $< Target, pencil >$ have different lineage according to Table 6. The proof is trivially done through the contradiction to Theorem 1.

Lemma 2 *Let $v(R_1, ..., R_m)$ be the view, tuples $t_1, ..., t_n \subseteq v$ have the same value t due to the bag semantics, $\forall \widetilde{DL}_x \subset < R_1, ..., R_m >$: $v(\widetilde{DL}_x) = t \rightarrow \exists i$: $\widetilde{DL}(t_i, v) \subseteq \widetilde{DL}_x$)*

This lemma says any lineage set that produces a tuple with the value t must be the superset of the lineage set of some existing tuple with the same value.

It can be proved by induction on the height of the definition tree. **Proof :**

(1) For $h = 1$,

(a) $\boldsymbol{v(R) = \sigma_C(R)}$.
Let $t_1, ..., t_n$ be the set of result tuples that have value t

$\Rightarrow$ $\widetilde{DL}(t_j, v) = \{t_j\}$, $j \in \{1, ..., n\}$ according to the lineage definition

$\Rightarrow$ Any lineage set that produces value t must contains one of the $\widetilde{DL}(t_j, v)$ according to the relational semantics.

(b) $\boldsymbol{v(R) = \pi_A(R)}$. The proof is similar to case (a).

(c) $\boldsymbol{v(R_1, ..., R_m) = \bowtie_\theta (R_1, ..., R_m)}$.
Let $t_1, ..., t_n$ be the set of result tuples that have value t

$\Rightarrow$ $\widetilde{DL}(t_j, v) =< \{t_{j1}\}, ..., \{t_{jm}\} >, j \in \{1, ..., n\}$

$\Rightarrow$ Any lineage set that produces value t must be the superset of one of the $\widetilde{DL}(t_j, v)$ according to the relational semantics.

(2) For $h = k$,

(a) $\boldsymbol{v^k(R_1, ..., R_m) = \sigma_C(w^{<k}(R_1, ..., R_m)}$
Let $t_1, ..., t_n$ be the set of result tuples that have value t

$\Rightarrow$ $\widetilde{DL}(t_j, v^k) = \widetilde{DL}(t_j, w^{<k})$, $j \in \{1, ..., n\}$ – (i)

$\Rightarrow$ Assume there is a $\widetilde{DL}_x$ that does not subsume any $\widetilde{DL}(t_j, v^k)$ but $v^k(\widetilde{DL}_x) = \{t\}$

$\Rightarrow$ $\widetilde{DL}_x$ does not subsume any $\widetilde{DL}(t_{\{1,...,n\}}, w^{<k})$ and $w^{<k}(\widetilde{DL}_x) = \{t\}$

$\Rightarrow$ Contradiction to the induction assumption.

The remaining cases can be proved similarly.

Proof : [**of Theorem 3**]

Assume $v_1'(D) \neq v_2'(D)$, therefore, $\exists \{t_{11}, ..., t_{1n}\} \subset v_1(D)$ and $\{t_{21}, ..., t_{2n}\} \subset v_2(D)$ s.t. $t_{11}, ..., t_{1n}, t_{21}, ..., t_{2n}$ all have the value of t, recall that we consider bag semantics, but $T_1 = \{< t_{11}, DL(t_{11}, v_1) >, ..., < t_{1n}, DL(t_{1n}, v_1) >\} \neq T_2 = \{< t_{21}, DL(t_{21}, v_2) >, ..., < t_{2n}, DL(t_{2n}, v_2) >\}$

According to Lemma 1, $DL(t_{1x})$'s and $DL(t_{2x})$'s are unique, $x \in \{1, ..., n\}$. Therefore, there must exist a $< t_{1i}, DL(t_{1i}, v_1) >$, which is in T_1 but not in T_2.

According to the definition of view equivalence, $v_2(\widetilde{DL}(t_{1i}, v_1))$ produces a tuple with value t. This is contradictory to Lemma 2.

View Definition As A Graph. Next we generalize the previous discussion to the case that the view definition is a graph. Theorem 1 does not hold in presence of a graph definition such as self joins. For example, query $v(\texttt{store}) = \pi_{R_1.s_id,R_1.s_name,R_2.s_id,R_2.s_name}(\texttt{store}\ as\ R_1 \bowtie \texttt{store}\ as\ R_2)$ produces the table below.

R_1.s_id	R_1.s_name	R_2.s_id	R_2.s_name
001	Target	001	Target
001	Target	002	Target
... ...			

The lineage of the second result tuple is $DL(< 001, Target, 002, Target >, v) = \{1, 2\}$, and thus the corresponding $\widetilde{DL}$ has the first two tuples in the `store` table. We can see that $v(\widetilde{DL}) = \{< 001, Target, 001, Target >, < \mathbf{001}, \mathbf{Target}, \mathbf{002}, \mathbf{Target} >, ...\}$.

Similarly, Theorem 2 does not hold in presence of a graph definition. Theorem 3 remains true.

Theorem 4 (Equivalence With Graph Definition) *Theorem 3 is true for all SPJ queries.*

In the previous discussion, we treat different relation instances as different relations and thus unique ids are assigned to their tuples. In the case of a graph derivation, a relation is referenced multiple times, and thus the same set of id numbers are assigned to the multiple instances of the same relation. From the fact that Theorem 3 is true in the previous instance based id assignment, we can infer that it remains true in the relation based assignment because multiple old ids are mapped to one new id. For example, let us assume relations R_1 and R_2 are the base tables, R_1 is referenced twice such that the derivation is a graph. Let us first treat the two instances of R_1 as two relations, denoted by R_1^1 and R_1^2. Let us further assume the two equivalent lineage sets before and after transformations are $\{R_1^1.3, R_2.7, R_1^2.3\}$ and $\{R_1^2.3, R_2.7, R_1^1.3\}$. $R_1^1.3$, $R_1^2.3$, $R_2.7$ represent unique tuple ids. Now let us consider the original graph structure. It has the effect of mapping previous multiple unique ids to the same id. In our example, it maps both $R_1^1.3$ and $R_1^2.3$ to $R_1.3$. The two lineage sets become $\{R_1.3, R_2.7\}$ and $\{R_1.3, R_2.7\}$, which remains equivalent.

3.2 ASPJ Queries

In this subsection, we discuss the properties of forward lineage for ASPJ queries with *aggregation* operators.

If we assume that the view definition is a tree, theorem 1 holds.

Theorem 5 (ASPJ Completeness) *Assuming a tree definition, $v(\widetilde{DL}(t, v(R_1, ..., R_k)) = \{t\}$ for ASPJ queries.*

The proof is elided for brevity.

Theorems 2 and 3 do not hold even the definition is a tree .For example, queries $v_1(R) = \alpha_{G,avg(x)}(R)$ and $v_2(R) = \alpha_{G.avg(x)}(\sigma_{x\neq 0}(R))$ are equivalent. However, they produce different lineages if R has some tuples whose x fields have the value of 0. Note that although the two queries are equivalent, it is unclear how a query planner can exploit this type of equivalence and perform automatic transformation. In order to prove it, we need to know exactly the set of transformation rules in a DBMS.

Comparison With Backward Lineage In [6], Cui et al. used reverse queries to trace backward lineage. Their definition covers general queries including ASPJ subqueries in both bag and set semantics. Their technique has been shown to be especially effective in tracing data lineage for SPJ queries with set semantics in [6]. Next we compare their definition with ours. Experimental comparison is performed in Section 5.

Backward lineage (set semantic) is defined as following according to [6]. Let $v(R_1, ..., R_m)$ be the view, $\widetilde{DL}_b(t, v)$ represents the backward lineage of tuple t.

Definition 2

- *if $v = R_i$, $\widetilde{DL}_b(t, v) = \{t\}$;*
- *if $v = Op(R_1, ..., R_m)$, in other words, the view is produced by applying a relational operation on the base tables, $\widetilde{DL}_b(t, v) =< R_1^*, ..., R_m^* >$ s.t.*
  (i). $v(\widetilde{DL}_b(t, v)) = Op(R_1^*, ..., R_m^*) = \{t\}$
  (ii). $\forall R_i^* : \forall t' \in R_i^* :$
  $v(\widetilde{DL}_b(t, v)) = Op(R_1^*, ..., \{t'\}, ..., R_m^*) \neq \phi$
- *if $v = Op(v_1, ..., v_k)$, which means the view is produced by applying a relational operation on intermediate views, $\widetilde{DL}_b(t, v) = \widetilde{DL}_b(t, v_1) \cup ... \cup \widetilde{DL}^b(t, v_k)$*

The following theorem states the relationship between forward lineage and backward lineage.

Theorem 6 *Assuming ASPJ queries and set semantics, forward lineage and backward lineage are equivalent.*

Proof Sketch: assuming every tuple can be identified by its global id assigned by function id(t), then if $v = R_i$, $\widetilde{DL}_b(t, v) = \{t\} = \{id(t)\}$. if the view is produced by applying a relational operation on the base tables, equation (i) is theorem 1 and equation (ii) is theorem 2. If $v = Op(v_1, ..., v_k)$, which means the view is produced by applying a relational operation on intermediate views, we can show $\widetilde{DL}_b(t, v) = \widetilde{DL}_b(t, v_1) \cup ... \cup \widetilde{DL}^b(t, v_k)$ for forward lineage by induction. The proof is elided for brevity.

Note that even though the two definitions are equivalent, the different realization techniques cause different efficiency in different scenarios.

3.3 Query Rewriting

In the previous subsections, we discuss the definition and the properties of forward lineage. In this subsection, we discuss how to rewrite SQL queries to trace forward lineage.

Let R be a base table, `T` be its schema, and V_1,...,V_m be base tables or intermediate views with their schemas being $\texttt{T}_1, ..., \texttt{T}_m$. The set of rewriting rules are given as follows.

◇ *Base Tables*:
T$\longrightarrow$ T'=<T, $\boldsymbol{DL}$ >; $\forall \boldsymbol{t} \in \boldsymbol{R}: \ \boldsymbol{t.DL} = \{\boldsymbol{id(t)}\}$.

◇ *Selection & Projection*:
`select ... from` V `where` C $\longrightarrow$ `select` **`...,`** $\boldsymbol{V.DL}$ **`as`** $\boldsymbol{DL}$ `from` V `where` C.

◇ *Join*:
`select ... from` $V_1, ..., V_m$ `where` C $\longrightarrow$ `select` **`...,`** $\boldsymbol{V_1.DL \cup ... \cup V_m.DL}$ **`as`** $\boldsymbol{DL}$ `from` $V_1, ..., V_m$ `where` C.

◇ *Aggregation*:
`select ... from` V `group-by` G `having` C $\longrightarrow$ `select` **`...,`** **DLUNION($\boldsymbol{V.DL}$)** **`as`** $\boldsymbol{DL}$ `from` V `group-by` G `having` C

The instrumentation is highlighted in the above rules. We can see that a new field DL is added to the base tables in order to facilitate lineage tracing. Function $id(t)$ assigns a unique id to a tuple. One can choose to create separate lineage tables if altering the base tables is not desirable. If the query is a simple `select`, the DL field is added to the selection list. In case of join, the query is instrumented to union the DL fields of the tables being joined. Finally, to handle aggregation queries, a new aggregation operator `DLUNION` is introduced to aggregate the DL sets by union. Examples of the rewriting rules have been given in the motivation section.

Query Optimization In general, queries are optimized by estimating the cost of operations and then reordering the operations accordingly based upon a set of equivalence laws. The aforementioned query rewritings do not affect the numbers of tuples in any circumstance. As a consequence, it is very likely the query optimizer will not take a different plan. However, the cost of lineage processing operations such as `DLUNION` highly depends on the cardinality of the lineage set, which may be affected by the order of the relational operations. We would leave this problem to our future work.

3.4 Handling More General Queries

In this section, we discuss how we handle more general views such as subqueries in `where` clauses.

One type of relational operations that we have not covered are the *union, intersection*, and *difference* operations. The union operation is trivial because it does not change the lineage of tuples. In other words, after union, each tuple has the same lineage as it had before the operation.

The intersection and difference are more intriguing. Let us assume two views v_1 and v_2 and we are interested in the lineage of a tuple $t \in (v_1 - v_2)$. It is apparent that $t \in v_1$. Now we have two different definitions of the lineage of t. The first definition says that $DL(t, v_1 - v_2) = DL(t, v_1)$ since the data value of t is copied from v_1. Such a definition is close to the *where* lineage in [5], but not exactly the same, because the *where* lineage does not consider aggregation while we do. Therefore, we call this type of lineage as *data lineage*. The forward lineage we have discussed so far belongs to this category.

In contrast, the second definition says $DL(t, v_1 - v_2) = DL(t, v_1) \cup (\bigcup_{t' \in v_2} DL(t', v_2))$. In other words, the lineage of t is the union of the lineage of t in v_1 and the lineage of all tuples in v_2. The logic behind this is that the appearance of t in $(v_1 - v_2)$ is *controlled* by the fact that each $t' \in v_2$ does not equal to t. We call it *control lineage*. It is similar to the *why* lineage in [5] but more restrictive. If *data* definition is assumed, our technique trivially handles the intersection and difference operations. If *control* definition is assumed, a small extension to the rewriting rules suffices. The extension is elided for brevity. The idea is to aggregate the lineage of v_2 tuples and then union with $DL(t, v_1)$.

We have considered the case that subqueries appear in the `from` clauses. However, in general subqueries can appear in the `where` clauses as conditions. Consider the example of Q_8, which selects the items that sell for more than 1500 units in store 01. If the *data* definition is considered, the rewriting is trivial. If the *control* lineage is requested, Q_8 is rewritten to Q_9, in which the sub-query is hoisted to the `from` clause and its lineage is aggregated. The aggregated lineage is unioned with the lineage of a tuple t in the main query if t satisfies the condition.

Q_8:

```
select i_name from item
where item.i_id IN
     (   select i_id from Sales s
      where s.s_id=1 and s.num > 1500 );
```

Q_9 (translated from Q_8:

```
select i.i_name, i.i_id U temp.DL  from item i,
  ( select DLUNION({s.t_id}) As DL from Sales s
    where s.s_id=1 and s.num >  1500 ) as temp
where item.i_id IN
     (   select i_id from Sales s
      where s.s_id=1 and s.num > 1500 );
```

Such a rewriting rule can be applied to any uncorrelated sub-query that appears in the `where` clause if *control* lineage is desired.

Handling correlated queries is much more challenging. It seems to us that query rewriting is not sufficient in this case. We would leave it to our future research.

4 Tracing Forward Lineage Efficiently

In the previous section, forward lineage has been formally studied. This section discusses how to implement forward lineage tracing in practice.

4.1 Characteristics of Data Lineage

In order to achieve efficient implementation, we first discuss the characteristics of lineage sets.

Table 7 shows the characteristics of the lineage sets for the queries in the *TPC-H* benchmark. The results for all the queries are not presented because some of them are correlated queries or queries with syntax that are not handled by our current implementation. Note that all the listed queries are ASPJ queries. From the table, we observe the following:

- For some queries, such as Q3, Q10 and Q16, the lineage size is small and the cardinality is very large. For the remaining queries, the lineage size is very large and the cardinality is very small. It corresponds to queries that aggregate a large number of base tuples into a few result tuples. There are no queries that have mid-sized lineage.
- As shown by the replication factor, the data lineage of many queries tend to have significant overlap in their binary representation. In other words, the binary representation of a base tuple often contributes to multiple result tuples. This is usually due to the `join` operations and the binary representation itself.
- As shown by the clustering factor, tuple ids in a data lineage set are often clustered. More specifically, if a tuple id x appears in the lineage, the tuple id $x + \delta$ is likely to appear in the same lineage as well. Note that we assign ids in chronological order such as `shipdate` or `orderdate`. If such an order is not available, we assign based on the key.

Our experience with a real *Wal-mart* transaction data set is similar. Note that both the *TPC-H* and the *Wal-mart* data sets belong to data warehousing, other workload may display different characteristics. We will leave this to our future study.

4.2 Representation Sets Using Binary Decision Diagrams (BDDs)

The first implication from the above observations is that a practical implementation needs to handle large lineage sets. A simple set based implementation may entail substantial runtime overhead because each operation may involve handling sets with the cardinality of hundreds or thousands. It is known that *reduced ordered* BDDs can efficiently implement set operations [11]. In this section we show that the characteristics of data lineage can be exploited by roBDDs.

We begin by describing how ordered BDDs are used to represent sets and how they are reduced. Given π, the total order on a set of variables $v_0, ..., v_n$, an *ordered* BDD is a directed acyclic graph that satisfies the following properties:

- There are exactly two nodes without outgoing edges, labeled by the constants 1 and 0 respectively. They are called **sinks**.
- Each non-sink node is labeled by a variable v_i, and has two outgoing edges, called the **1-edge** and the **0-edge**. The *1-edge*s are drawn as solid arrows while *0-edge*s are drawn as dashed arrows.
- The order in which the variables appear on a path is consistent with the variable order π.

Let us assume that we are given a universal set which contains integers 0 through 15. We show how any set drawn from this universal set can be represented using an ordered BDD. Since each element of the set can be uniquely represented using four bits, we represent it using an ordered BDD with four variables, corresponding to the four bits, with v_3, v_2, v_1, v_0 as the variable order. The ordered BDD representing the set $\{0, 1, 2, 3, 4, 9, 10, 11, 12\}$ is shown in Fig. 2(a). To decide if 4 (i.e., 0100) is in the set, we follow the path 0100 to see if the 1-sink is reached.

An ordered BDD can be converted to a more compact *reduced ordered* BDD (roBDD) using two rules:

	Q3	Q5	Q6	Q7	Q9	Q10	Q11	Q15	Q16
cardinality	11620	5	1	4	175	37967	1048	1	18308
avg. lineage size	4.6	4285.8	114160	4397	6738.1	6.3	32077	225955	12.06
replication factor	5	5	6	5	14	11	6141	5	13
clustering factor	0	21.54	33.09	21.83	8.87	0	100	99.99	5.09

The **replication factor** of a query having the value x means that on average the binary representation of a base tuple can appear in x result tuples. The larger the x value, the more overlap the result lineage sets have.

Assume the tuples ids in the result lineage are ordered, the **clustering factor** shows how many percentage of these ids have a < 3 id distance to their neighbors.

Table 7. Characteristics of Lineage

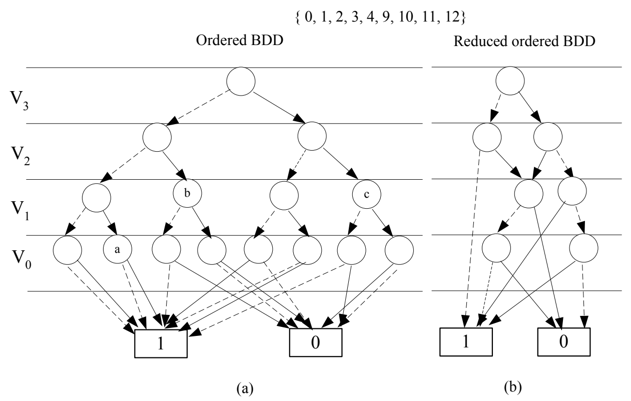


Figure 2. Reduced ordered BDD for a set.

the *Elimination Rule* and the *Merging Rule*. According to the *elimination rule*, if both edges of a node n point to the same successor s (i.e., the value of the variable corresponding to the current level in the BDD does not effect path selection), then the node n is eliminated by redirecting all incoming edges of n to its successor s. According to the *merging rule* if two nodes n_1 and n_2 are isomorphic, we can remove the redundancy by merging the two nodes and redirecting the incoming edges of these nodes to the merged node. The resulting BDD is called a reduced ordered BDD or roBDD. For the example ordered BDD of Fig. 2(a), node a can be removed using the elimination rule and nodes b and c can be merged using the merging rule. The final roBDD is shown in Fig. 2(b). These reduction rules make roBDD a very efficient for representation of large sets. For example, a universal set can be represented by a single node in roBDD while the cardinality can be any large.

Since we are interested in maintaining lineage, we need to construct multiple roBDDs. These roBDDs can also share nodes and thus we obtain a multiple rooted roBDD where each root corresponds to a distinct lineage set. The example in Fig. 3 illustrates how roBDDs of two sets $\{0, 1, 2, 3, 4, 9, 10, 11, 12\}$ and $\{0, 1, 2, 3, 4, 8, 9, 10\}$ are represented. As we can see, the two roBDDs share nodes. If two lineage sets overlap, they share nodes in the graph. This is how roBDDs exploit the *overlapping* characteristic.

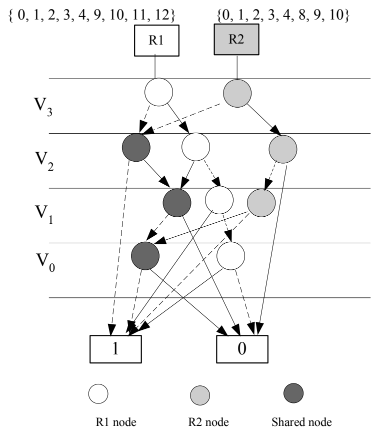


Figure 3. roBDDs for two sets.

More over, the *clustering* characteristic of lineage gives rise to the opportunities of applying the elimination rule and the presence of multiple clusters create isomorphic subgraphs, enabling the application of the merging rule.

Due to the aforementioned reduction, set operations with roBDDs can be performed efficiently because they depend on the number of nodes in the involved roBDDs, which is often much smaller than the cardinalities of the sets represented. For example, binary operations (e.g., union) on two sets whose roBDD representations contain n and m nodes can be performed in time $O(n \times m)$ [11].

4.3 Variable Granularity Lineage

The second implication is that the support for variable granularity lineage is desired. For data verification, manually inspecting thousands or even millions of lineage tuples is impractical.

Based on the query rewriting rules described earlier, variable granularity lineage tracing can be easily supported by introducing a new `lineage-on` clause which has similar syntax to the `group-by` clause. Thereby, the following query traces lineage with respect to the attribute setting of L.

`select ... from` V `lineage-on` L `...`

For example, assume Q_4 in Section 2 is added the new clause "`lineage-on store.s_name`", the result lineage is computed with respect to store names instead of base tuples. We need to slightly change the first rewriting rule in section 3.3 so that a unique id is assigned to a unique value in L through hash functions. It is unclear how the reverse query method can achieve this. With the new clause, the user can also turn off forward lineage tracing by simply having an empty L.

5 Experimental Results

5.1 Implementation

The forward lineage tracing system is built on top of PostgreSQL v7.4.5. We have two implementations. The *set* implementation represents lineage as ordered sets using PostgreSQL's array data type. A union of two ordered set is a simple merge. The *bdd* implementation represents lineage as roBDDs using the BuDDy [1] BDD package. We also implement the reverse query algorithm for comparison. The experiments are conducted on a Sun-Blade-1000 machine with dual 1.2 GHz UltraSPARC-III+ and 2G memory running SunOS 5.8. The benchmark is *TPC-H*.

5.2 TPC-H Dataset

In this experiment we use benchmark *TPC-H*. The data is generated by setting the scale factor to 1 and the size of the data is 1GB. The schema and the cardinality of each tale can be found in [2].

5.3 Performance

The first set of experiments compare the performance of different algorithms and implementations. The results are presented in Figures 4 and 5 for ASPJ and SPJ queries, respectively. The ASPJ queries are the standard ones provided by *TPC-H*. All queries are not evaluated. The missing queries are either correlated or having syntax that is not handled by our current implementation. Since *TPC-H* does not have SPJ queries, we compose a number of SPJ queries based on the same data set. These queries feature different numbers of join operations. The details of these queries can be found in appendix A.

Four approaches are evaluated: i) the *original* query without any lineage tracing; ii) forward tracing using the *set* implementation; iii) forward tracing using *bdd*; and iv) backward tracing using reverse queries [7]. In order to achieve fair comparison, we collect two numbers for the backward approach. *Reverse-one* shows the average time for querying the lineage of a single result tuple. *Reverse-all* shows the time for querying the lineage for all result tuples. Note that reverse-all is not reverse-one times the number of result tuples.

BDD vs. Set Let us first compare the two forward implementations. We can see from Figures 4 and 5, for queries that produce small lineage sets (see Figure 7), such as all the SPJ queries, and some ASPJ queries like Q3, Q10, *set* is comparable to or better than *bdd*. In other words, a simple set based implementation suffices in this case because the union operations of small sets are usually very efficient. For queries that produce large lineage sets like Q5 and Q6, *Bdd* is consistently substantially better than *set*. For ASPJ queries, the overhead for *set* is 7.96 times larger than that of *bdd* on average. The average overhead of *bdd* is 387%, if Q11 is excluded, the average is only 74%. Note that since roBDDs is memory resident, we only use roBDDs to compute lineage, and the final results are translated from roBDDs to sets for storage. The execution time of *bdd* includes the translation overhead. The translation time for Q11 is dominant. It clearly demonstrates the great benefit of using roBDDs, which perfectly exploits the regularity of data lineage.

Forward vs. Backward. Next, let us compare forward tracing with the lazy backward tracing enabled by reverse queries. As Figure 5 shows, for SPJ queries, *set* incurs an almost negligible increase of execution time (the average overhead is 1.7%) because the lineage sets have very small cardinalities. In contrast, the overhead is 325% for *reverse-all* and 82% for *reverse-one*. The backward approach performs poorly because of the need to perform an expensive reverse query for which the execution time increases as the number of joins increases.

For ASPJ queries, as shown in Figure 4, *bdd* is consistently much better than *reverse-all* except Q10, in which these two are comparable. Q10 is a query whose result lineage has very little regularity. For ASPJ queries, the overhead for *reverse-all* is 19.44 times larger than that of *bdd* on average. Note that for Q7, Q9, and Q11, since reverse queries result in huge joined tables, reverse-all fails to run due to the `out-of-memory` errors.

Reverse-one is comparable to *bdd*, but we want to

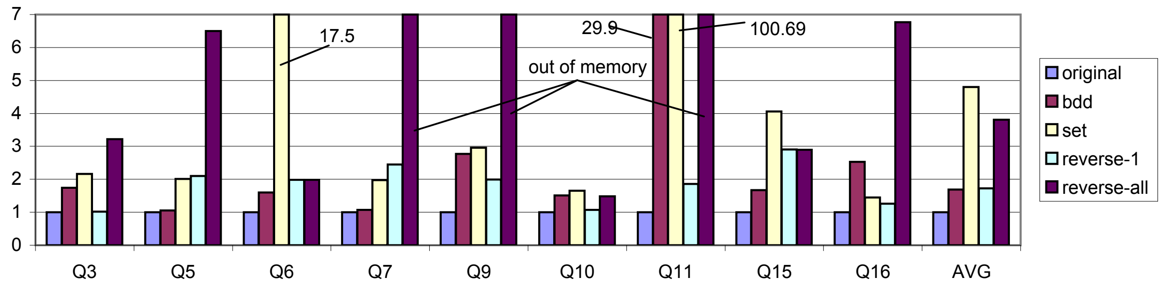


Figure 4. Performance of ASPJ Queries.

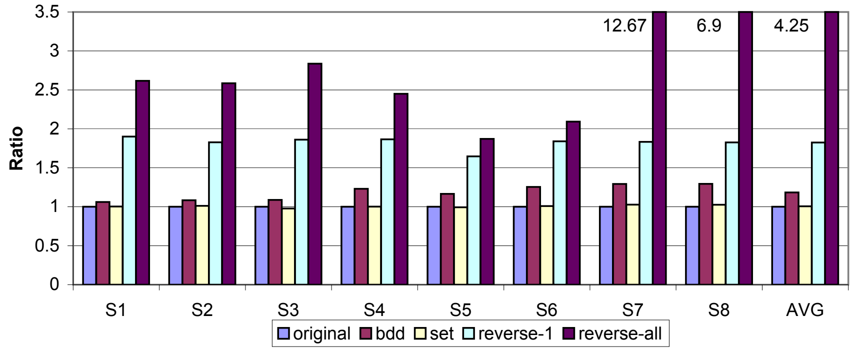


Figure 5. Performance of SPJ Queries.

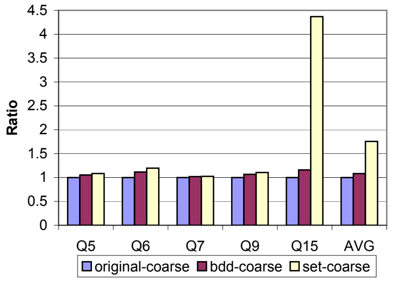


Figure 6. Performance of Variable Granularity Lineage Tracing.

emphasize that *bdd* produces the result lineage for all result tuples at once and *reverse-one* can only generate lineage for one result tuple. In other words, even if the lineage for a single tuple is desired, the user may still choose to use forward lineage tracing and yield much abundant results with comparable performance.

This experiment clearly shows that the proposed forward method is better than the reverse query method in terms of performance.

5.4 Space

We have also carried out experiments to evaluate the space overhead. Since we do not store any roBDDs, the required space is for storing the final lineage sets. We use one 32-bit integer to represent one id. Therefore, the space requirement for ASPJ queries is $4 \times$ `row one in Figure 7` $\times$ `row two in Figure 7` . The results range from 70KB to 134MB. The space requirement is smaller for SPJ queries because the lineage for SPJ queries are often small fixed numbers.

5.5 Variable Granularity Forward Lineage

As mentioned earlier, ASPJ queries often produce result tuples with the lineage of thousands of base

	Q5	Q6	Q7	Q9	Q15
cardinality	5	1	4	175	1
avg. lineage size	394	364	334	402	91
space (byte)	7880	1456	5344	281400	364

Table 8. Space for Variable Granularity Tracing.

tuples. Thereby, the support of variable granularity lineage provided by our method is highly desirable. We rewrite Q5, Q6, Q7, Q9, and Q15 by adding the `lineage on (year of shipdate, week of shipdate, mode)` clause so that lineage is computed with respect to the combination of the three attributes. The performance is shown in Figure 6. The space requirement is shown in Table 8.

From Table 8, the lineage size has been significantly reduced so that it becomes human manageable. The space requirement is also reduced. From Figure 6, *Bdd* still delivers better performance than *set* for these mid-sized lineage. We were not able to construct the reverse queries and hence the comparison with the backward approach is not available here. To summarize,

- Forward methods have much better performance than the backward approach;
- RoBDD implementation is good for mid-sized and large lineage sets; Set implementation is good for small lineage sets;
- Forward methods generate all lineage at once, and the backward approach is able to generate lineage

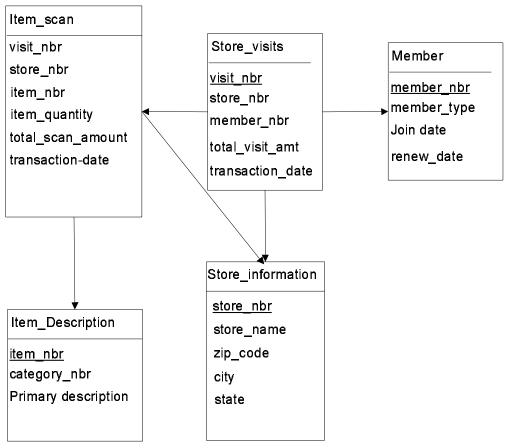


Figure 7. The Schema of Walmart Dataset

on demand. They have different application scenarios. However, the user can choose to discard some of the lineage to save space.

- Variable grained lineage, which is supported by the forward method, relieves users from inspecting huge lineage sets, significantly reduces the space, and thus makes the technique practical.

5.6 Walmart Dataset

The dataset is taken from a real Walmart transaction databases corresponding to sales at a number of stores. A subset of this data corresponding to a single day's transactions was used for our experiments. The size of this data is 300 Megabytes. The schema is shown in Figure 7.

5.6.1 Performance

We evaluate queries with 0, 1, 2, and 3 joins for SPJ queries(denoted as $Q1$, $Q2$, $Q3$, $Q4$, respectively) and 0, 1, 1, and 2 joins for ASPJ queries (denoted as $Q5$, $Q6$, $Q7$, $Q8$, respectively). These queries can be found in appendix B. Note that for SPJ queries, as the number of joins increases, the size of the lineage sets increases linearly. For ASPJ query, when the number of joins increases, the intermediate lineage size before aggregate increase linearly. The size of the lineage of the resulting tuples depends on the `group by` clause.

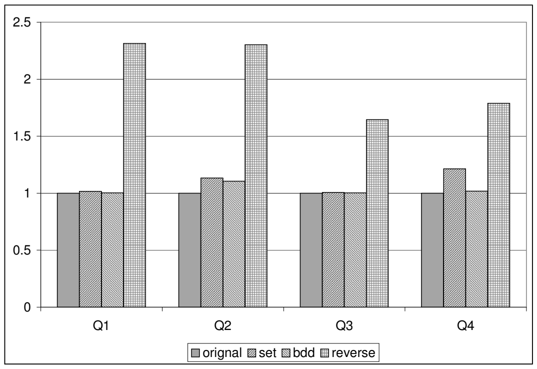
Execution Time Calculating the lineage information at execution incurs a performance overhead. We measure the execution time to compute the fine-grained lineage and compare it with the execution time without the lineage. Four approaches were tested: i) the original query without any lineage tracing (**original**; ii) forward tracing using the set implementation **set**); iii) forward tracing using roBDD **bdd**); and iv) backward tracing using the approach of [6] (**reverse**). Figure 8 shows the performance of the various approaches relative to that of the original query without lineage tracing. Figure 8(a) shows the results for SPJ queries and (b) shows the results for ASPJ queries.

As Figure 8 (a) shows, for SPJ query, our algorithm incurs an almost negligible increase of execution time. For Q3 and Q4, it is less than 0.5% and for Q2 it is 10%. In contrast, the overhead for the reverse query approach is significantly higher (e.g it more than doubles the time for Queries Q1 and Q2). This cost increase is incurred even for the tracing of a single result tuple. The reverse query performs poorly in this experiment because of the need to perform an expensive reverse query for which the execution time increases as the number of joins increases.

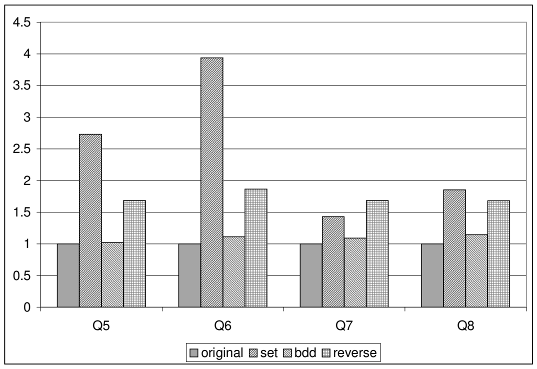
Notice that the implementation using sets also performs better than a reverse query. This is because for SPJ queries, the lineage size of each resulting tuple is the number of tables appearing in the `from` clause of the query. Therefore, the lineage size is very small. Both set and bdd implementations piggy-back the query execution and incur very small overhead, however, bdd performs slightly better than the set implementation.

For ASPJ queries, as Figure 8 (b) shows, our bdd implementation outperforms both set and reverse query implementations. Since bdd computes all lineage at query execution time, while reverse query only calculates the lineage on the fly, to achieve a fair comparison, the reverse query consists of materializing all necessary intermediate tables for lineage calculation and the reverse query for tracing one tuple. As expected, as the number of joins increases, the reverse query time increases accordingly. The execution times of our bdd and set implementations depend heavily on the size of the lineage. For Q5 and Q6, since the lineage size is large, the set operations for the integer set implementation becomes very expensive and is 60% and 110% more than reverse query for Q5 and Q6 respectively. When the lineage size decreases, the set implementation may outperform reverse query as we have seen in Q6. Whereas the execution time of bdd implementation stays low. The bdd implementation only has less than 15% overhead over the original query. This is due to the efficient set operations from bdd. In all cases, bdd outperforms set and reverse query implementations.

We should point out that in this experiment, the reverse query only calculates one tuple's lineage. If we need to calculate the lineage for multiple tuples the reverse query will include an extra join which would



(a)



(b)

Figure 8. Performance of the various approaches for (a) SPJ queries; and (b) ASPJ queries. The y-axis shows the execution time ratio to that of the the same query without any lineage tracing (forward or backward).

	Q1	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**	**Q8**
cardinality	14898	14898	12376	12376	130	121	3293	7608
avg lineage size	1	2	3	4	11248	2034	405	946
total lineage size	14898	29796	37128	49504	1462254	246156	1335824	1343557
space (bytes)	59592	119184	148512	198016	5849016	984624	5343296	5374228

Table 9. Space requirements for the various queries.

introduce a further slowdown. For the forward lineage approaches, there is no extra cost for tracing multiple tuples since the lineage of all tuples is computed all the time. The execution time for the reverse query depends heavily on the cardinality of intermediate table. If the cardinality of reverse table is large, then the execution time will increase. The execution time also depends on the number of joins in the original query. The lineage size has little effect on the execution time of reverse query.

Space In order to investigate the storage overhead of storing fine-grained lineage, we measure the space cost in the number of bytes. The result is shown in Table 9. The first row (**cardinality** gives the number of tuples in the result for each query. The second row **avg lineage size** shows the average size of the lineage for each result tuples. The third row **total lineage size** gives the size of the lineage for the entire result and the fourth row **space** gives the space overhead incurred by storing fine-grained lineage. We use a 32 bit integer to represent one id. The storage of lineage for SPJ queries range from 39Kb to 198Kb and 984Kb to 5.8Mb. The space requirement for SPJ queries is smaller because the lineage size is proportional to the number of joins and often small numbers. On the other hand, for ASPJ queries, the storage could be very large due to the aggregation.

Overall we see that the forward lineage tracing approach is very efficient in terms of execution time. It incurs only a negligible overhead during normal query execution while providing the lineage for all the resulting data. In contrast, a reverse lineage tracing approach has to pay a high overhead for computing the lineage of even a single result tuple. There is however, an overhead for storage space that must be paid for using the forward lineage tracing. However, given the increase in efficiency of computing the lineage of all results tables, we believe this is a very reasonable overhead. The immediate availability of the lineage also have significant benefits for many applications. For example, if a base tuple is found to be incorrect, it is possible to easily identify all result tuples that are affected by the given base tuple if forward lineage is available. However, with the reverse lineage approach it would be necessary to compute the lineage for all result tuples and then test which ones are affected by the given base tuple.

6 Related Work

The need for DBMS support for fine-grained lineage tracing has been well known for new applications such as scientific databases. Although the need is urgent, it remains an unsolved problem. Recently, there has been increasing interest in this area. Cui *et al.* [6, 7, 8] propose fine-grained tracing in the context of data warehousing. The notion of reverse queries that are automatically generated is presented in order to produce all tuples that participated in the computation of a given query. Woodruff and Stonebaker [14] support fine-grained lineage using inverse or weak inverse functions. That is, the dependence of a given result on base data is captured using a mathematical function. It is not clear if such functions can be identified for a given application. The identification task is highly non-trivial and makes the approach impractical. Both works adopt a lazy approach to compute fine-grained lineage backward upon request from the user. Although the reverse query approach usually works well, if the lineage for a large number of tuples needs to be generated, reverse queries may not be efficient.

Forward lineage has been used by Bhagwat *et al.*[3] where they propose three schemes to propagate *where* lineage embedded in annotations attached to attributes in relational data. The notion of *where* lineage is first proposed in [5]. *Where* lineage specifies where the data is copied from. In [3], the *where* lineage is a unique address recorded in the annotation, along with other non-lineage information. The paper propose a SQL extension, called pSQL to trace the where lineage for a fragment of SQL that corresponds to conjunctive queries with Union(also known as Select-Project-Join-Union) in set semantics. Aggregation is briefly discussed in the paper, only min, max could have a valid where lineage definition, therefore, only min and max can be traced. As where lineage only records where the data is copied from, *where* lineage is not sufficient for many database applications (e.g. scientific database).

The use of *where* lineage is limited since it can not handle data processing (it only handles copying of data). Furthermore, it is only applicable for set semantics and does not handle bag semantics. The approach presented in this paper is inspired by the forward lineage approach proposed in [3]. It goes beyond the notion of *where* lineage and is able to handle arbitrary SPJ and ASPJ queries, and also bag semantics.

The intuition behind *where* lineage is rooted in the classical view maintenance problem. Another limitation of these approaches is that the lineage information is stored as unstructured text which makes it very difficult to analyze.

In scientific computation, provenance or lineage has been extensively studied for datasets on the grid. Most of attention has been given to coarse-grained lineage and work-flow management problem. [4] surveys the use of work-flows in scientific computation. In this paper we are concerned with fine-grained lineage (i.e. lineage for each tuple).

In this paper we present a novel technique for forward computation of fine-grained lineage for arbitrary SPJ and ASPJ queries with both set and bag semantics. This is achieved by query rewriting techniques that piggy-back the lineage computation with query processing. The execution overhead introduced by this lineage computation is found to be extremely low (especially in comparison to the reverse query methods). While they do incur a significant space overhead, the advantage of having pre-computed lineage is worth the extra cost in storage for many scientific applications.

7 Conclusion

Data lineage plays a critical role in verifying data correctness in scientific databases and data warehousing. Prior lineage tracing techniques compute lineage *backward*. While such techniques are quite efficient in certain scenarios such as SPJ queries with set semantics, their inherent limitations make them inapplicable or inefficient in other scenarios, for example, queries with bag semantics and nested queries. Following the direction of forward annotation propagation [3], we propose a cost effective *forward* lineage tracing technique. It efficiently handles bag semantics and non-correlated subqueries for general views. Moreover, it enables forward lineage inspection, namely, identifying the set of results that rely on a particular base tuple, which could be highly desirable in scientific databases. For the first time, we define data lineage in bag semantics and prove its properties. Reduced Ordered Binary Decision Diagrams (RoBDDs) are used to exploit the characteristics of data lineage, which results in a cost effective implementation. The experimental validation is based upon an implementation in PostgreSQL and testing with real data warehouse data from Walmart. The results establish the extremely low overhead of our query rewriting techniques that compute lineage during query execution. In particular, the overhead for our technique was found to be lower than 15% for all queries whereas the overhead for tracing the lineage for a single tuple using the reverse query approach can be as high as 220%. While this is not the first work dealing with forward lineage tracing, it is the first to handle arbitrary SPJ and ASPJ queries (beyond simple copying of data). In future work, we plan to address the interesting and challenging problem of tracing lineage for correlated nested queries.

References

[1] Buddy, a binary decision diagram package. Department of Information Technology, Technical University of Denmark.

[2] http://www.tpc.org/tpch/.

[3] D. Bhagwat, L. Chiticariu, W. C. Tan, and G. Vijayvargiya. An annotation management system for relational databases. In *VLDB*, 2004.

[4] R. Bose and J. Frew. Lineage retrieval for scientific data processing: a survey. *ACM Comput. Surv.*, 37(1):1–28, 2005.

[5] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *ICDT*, pages 316–330, 2001.

[6] Y. Cui and J. Widom. Lineage tracing in a data warehousing system. In *ICDE*, pages 683–684, 2000.

[7] Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. *VLDB J.*, 12(1):41–58, 2003.

[8] Y. Cui, J. Widom, and J. L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.*, 25(2):179–227, 2000.

[9] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *SIGMOD*, 1995.

[10] P. Groth, S. Miles, W. Fang, S. C. Wong, K. P. Zauner, and L. Moreau. Recording and using provenance in a protein compressibility experiment. In *HPDC'05*, July 2005.

[11] C. Meinel and T. Theobald. Algorithms and data structures in VLSI design, 1998. Springer.

[12] S. Miles, P. Groth, M. Branco, and L. Moreau. The requirements of recording and using provenance in e-science experiments. *Journal of Grid Computing*, 2006.

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

[14] A. Woodruff and M. Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *ICDE*, pages 91–102, 1997.

APPENDIX

A SPJ Queries for *TPC-H* dataset

S_1:

```
select o_orderdate, o_totalprice, l_quantity, l_shipdate
from lineitem, orders
where o_orderkey = l_orderkey
and o_orderdate >= date '1994-01-01'
and o_orderdate <= date '1994-01-01' + interval '3 day';
```

S_2:

```
select c_name, o_orderdate, o_totalprice,
      l_quantity, l_shipdate
from lineitem, orders, customer
where o_orderkey = l_orderkey
and c_custkey = o_custkey
and o_orderdate >= date '1994-01-01'
and o_orderdate <= date '1994-01-01' + interval '3 day';
```

S_3:

```
select c_name, n_name, o_orderdate, o_totalprice,
      l_quantity, l_shipdate
from lineitem, orders, customer, nation
where o_orderkey = l_orderkey
and c_custkey = o_custkey
and c_nationkey = n_nationkey
and o_orderdate >= date '1994-01-01'
and o_orderdate <= date '1994-01-01' + interval '3 day';
```

S_4:

```
select p_name, p_brand, p_type, s_name,  ps_supplycost,
      l_shipdate, o_orderdate
from part, partsupp, supplier, lineitem, orders
where p_partkey = ps_partkey
and ps_suppkey = s_suppkey
and l_partkey = ps_partkey
and l_suppkey = ps_suppkey
and o_orderkey = l_orderkey
and o_orderdate >=date '1994-01-01'
and o_orderdate <= date '1994-01-01' + interval '3 day';
```

S_5:

```
select c_name,  o_orderdate, l_shipdate, p_name,
       p_brand, p_type, s_name,  ps_supplycost
from customer, part, partsupp, supplier, lineitem, orders
where p_partkey = ps_partkey
and ps_suppkey = s_suppkey
and l_partkey = ps_partkey
and l_suppkey = ps_suppkey
and o_orderkey = l_orderkey
and c_custkey = o_custkey
and o_orderdate >=date '1994-01-01'
and o_orderdate <= date '1994-01-01'
+ interval '3 day;
```

S_6:

```
select c_name,  o_orderdate, l_shipdate, p_name,
      p_brand, p_type, s_name,
      n_name, ps_supplycost
from customer, part, partsupp, supplier,
    lineitem, orders, nation
where p_partkey = ps_partkey
and ps_suppkey = s_suppkey
and l_partkey = ps_partkey
and l_suppkey = ps_suppkey
and o_orderkey = l_orderkey
and c_custkey = o_custkey
and s_nationkey = n_nationkey
and o_orderdate >=date '1994-01-01'
and o_orderdate <= date '1994-01-01'
+ interval '3 day';
```

S_7:

```
select c_name, n2.n_name,  o_orderdate,
      l_shipdate, p_name, p_brand, p_type,
      s_name, n1.n_name, ps_supplycost
from customer, part, partsupp, supplier,
    lineitem, orders, nation n1, nation n2
where p_partkey = ps_partkey
and ps_suppkey = s_suppkey
and l_partkey = ps_partkey
and l_suppkey = ps_suppkey
and o_orderkey = l_orderkey
and c_custkey = o_custkey
and s_nationkey = n1.n_nationkey
and c_nationkey = n2.n_nationkey
and o_orderdate >=date '1994-01-01'
and o_orderdate <= date '1994-01-01'
+ interval '3 day';
```

S_8:

```
select c_name, n2.n_name as cust_nation ,r_name,
      o_orderdate, l_shipdate, p_name,
      p_brand p_type, s_name,
      n1.n_name as supp_nation, ps_supplycost
from customer, part, partsupp, supplier,
    lineitem, orders, nation n1, nation n2, region
where p_partkey = ps_partkey
and ps_suppkey = s_suppkey
and l_partkey = ps_partkey
and l_suppkey = ps_suppkey
and o_orderkey = l_orderkey
and c_custkey = o_custkey
and s_nationkey = n1.n_nationkey
and c_nationkey = n2.n_nationkey
and n1.n_regionkey = r_regionkey
and o_orderdate >= date '1994-01-01'
and o_orderdate <= date '1994-01-01' + interval '3 day';
```

B Queries for Walmart dataset

Q_1:

```
select item.store_nbr, item.item_nbr,
item.total_scan_amount
from item_scan item
where item.store_nbr =126;
```

Q_2:

```
select visit.visit_nbr, item.item_nbr,
item.total_scan_amount
from Item_scan item, store_visits visit
where visit.visit_nbr = item.visit_nbr
and visit.store_nbr =126;
```

Q_3:

```
select visit.visit_nbr, item_desc.primary_desc,
item.total_scan_amount
from item_scan item, store_visits visit, item_description
where visit.visit_nbr = item.visit_nbr
and item.item_nbr = item_desc.item_nbr
and visit.store_nbr =126;
```

Q_4:

```
select visit.visit_nbr, info.store_name,
item_desc.primary_desc, item.total_scan_amount
from item_scan item, store_visits visit,
item_description item_desc, store_information info
where visit.visit_nbr = item.visit_nbr
```

```
and item.item_nbr = item_desc.item_nbr
and info.store_nbr = visit.store_nbr
and visit.store_nbr =126;
```

Q_5:

```
select sum(item_quantity)
from  item_scan
group by store_nbr;
```

Q_6:

```
select sum(item.total_scan_amount)
from item_scan item, store_information info
where item.store_nbr = info.store_nbr
group by info.city;
```

Q_7:

```
select  item.store_nbr,  sum(item.total_scan_amount)
from item_scan item, item_description item_desc
where item.item_nbr = item_desc.item_nbr
group by  item.store_nbr, item_desc.category_nbr
having sum(item.total_scan_amount)>3000;
```

Q_8:

```
select  info.state,  sum(item.total_scan_amount)
from item_scan item, item_description item_desc,
store_information info
where item.item_nbr = item_desc.item_nbr
and item.store_nbr = info.store_nbr
group by  info.state, item_desc.category_nbr
having sum(item.total_scan_amount)>3000;
```