Storing Auxiliary Data for Efficient Maintenance and Lineage Tracing of Complex Views*

Yingwei Cui and Jennifer Widom
Computer Science Department, Stanford University
{cyw, widom}@db.stanford.edu

Abstract

As views in a data warehouse become more complex, the view maintenance process can become very complicated and potentially very inefficient. Storing *auxiliary views* in the warehouse can reduce the complexity and improve the efficiency of view maintenance, and the same auxiliary views can help in efficiently answering *lineage tracing queries* over the warehouse views. In this paper, we study the problem of selecting auxiliary views to materialize in order to minimize the total view maintenance and lineage tracing cost. We consider relational views with arbitrary use of aggregation operators, and we define an initial search space for our optimization problem based on a normal form for such view definitions. We present several auxiliary view selection algorithms, and to study their performance we conduct experiments using the TPC-D benchmark in addition to synthetic view definitions and statistics. The results of our experiments show: (1) the exhaustive algorithm that selects the optimal set of auxiliary views is far too expensive in many cases; (2) two heuristic algorithms that we present select good (often optimal) sets of auxiliary views in a much shorter time; (3) even auxiliary views selected by a very simple algorithm can significantly reduce the overall view maintenance and lineage tracing cost.

1 Introduction

Data warehousing systems collect data from multiple, distributed sources and integrate the information as *materialized views* in local databases [CD97, IK93, LW95, Wid95]. Users can then perform data analysis and mining on the warehouse views. The materialized views in the warehouse need to be kept up-to-date when data at the sources changes. As the view definitions become more complex in order to support sophisticated data analyses, the view maintenance process can become very complicated and potentially very inefficient. Most previous work on view maintenance, e.g., [CW91, GMS93, LW95, LYC+99, Qua96], considers simple views containing at most one level of aggregation. In order to efficiently maintain complex views which may contain multiple levels of aggregation, it is clearly advantageous to store *auxiliary data* in addition to the original view to reduce overall view maintenance cost.

From a different perspective, for in-depth analysis of warehouse data sometimes it is useful to be able to "drill through" from selected interesting (or possibly erroneous) view data to the original source data that derived the view data. We call this process *tracing the lineage* of the view data [CWW97]. To trace the lineage of a view data item efficiently, the warehouse also needs to store auxiliary data—to reduce

*This work was supported by the National Science Foundation under grant IIS-9811947, by Sagent Technology Inc., and by an equipment grant from IBM Corporation.

the computation cost at the warehouse, and to reduce or entirely avoid expensive source accesses for lineage tracing. It turns out that the same auxiliary data that can be used to improve the performance of view maintenance as discussed in the first paragraph also can improve the performance of lineage tracing queries. Therefore, the problems of selecting auxiliary data for the two purposes are closely related, and we study the problems together.

The auxiliary data is stored as materialized views in the warehouse, called *auxiliary views* (as opposed to the original warehouse views, which we call *primary views*). Given a complex relational primary view, there are numerous possible sets of auxiliary views to materialize for view maintenance and lineage tracing, with significant performance tradeoffs. In general, the more auxiliary views we materialize, the more efficiently we can maintain and trace the lineage of data in the primary view. However, the auxiliary views themselves also need to be maintained, so materializing too many auxiliary views can increase overall cost.

Previous work has studied the selection of views to materialize for answering queries, e.g., [HRU96, Gup97], and the selection of auxiliary views for efficient maintenance of given primary views, e.g., [LQA97, RSS96]. (Further discussion of this work appears in Section 2.) In [CW00], we introduced the idea of materializing auxiliary views to minimize overall view maintenance and lineage tracing cost, and we studied the problem in the context of select-project-join (SPJ) primary views. This paper investigates the more difficult and general problem of relational views with arbitrary use of aggregation and SPJ operators. As we will see, it is an expensive combinatorial problem, and our overall approach differs from [CW00].

In this paper, we first define a *normal form* for the primary view definition, which suggests an initial search space of possible auxiliary views. We then propose a variety of algorithms for selecting auxiliary views within this search space. Finally, we compare empirically the running time of our algorithms and the optimality of the auxiliary view sets they select, using the TPC-D benchmark [TPC96] in addition to a suite of synthetic view definitions and statistics. The results of our experiments show:

- The exhaustive algorithm that selects the optimal set of auxiliary views is far too expensive in many cases.
- Two heuristic algorithms that we present select good (often optimal) sets of auxiliary views in a much shorter time.
- Even auxiliary views selected by a very simple algorithm can significantly reduce the overall view maintenance and lineage tracing cost.

1.1 Outline of Paper

The remainder of the paper proceeds as follows. Section 2 covers related work. Section 3 presents preliminary material on materialized views, view maintenance, and lineage tracing, including a running example. Section 4 introduces the auxiliary views we consider for efficient view maintenance and lineage tracing, and defines the search space for selecting auxiliary views to materialize. Section 5 describes the cost model and statistics we use for estimating view maintenance and lineage tracing costs, and for studying the performance of our auxiliary view selection algorithms. Section 6 presents several algorithms for selecting auxiliary views within our search space. Section 7 compares the performance of the algorithms using experiments on the TPC-D benchmark, as well as using a variety of synthetic view definitions and statistics.

2 Related Work

Previous work related to this paper falls into three categories: selecting views to materialize in order to minimize query costs, e.g., [HRU96, Gup97], selecting auxiliary views to materialize in order to minimize the cost of maintaining given primary views, e.g., [LQA97, RSS96], and our own previous work in lineage tracing and view maintenance [CWW97, CW00].

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

[RSS96] proposes an exhaustive algorithm for selecting auxiliary views to optimize view maintenance, and suggests simple search space pruning strategies when the view is too complex for exhaustive search. [LQA97] presents an A* algorithm for selecting auxiliary views and indexes on different join combinations for SPJ view maintenance. Both [RSS96] and [LQA97] consider a single algorithm for selecting auxiliary views (and indexes in the case of [LQA97]), designed specifically for optimizing view maintenance. They consider as potential auxiliary views all nodes in all possible relevant query plans, making the search space doubly exponential in the view definition size.

We introduced lineage tracing for relational data warehouses in [CWW97], presenting a formal framework and basic algorithms. In [CW00], we introduced the problem of selecting auxiliary views to simultaneously reduce view maintenance and lineage tracing costs, and we considered the restricted case of SPJ views. We suggested several alternative auxiliary view schemes and compared their performance. In this paper, we tackle the problem for complex relational views with arbitrary use of aggregation and SPJ operators. Arbitrarily complex primary views make the auxiliary view selection problem more complicated and expensive than for SPJ views, and we take a different approach to solving it than for the restricted case considered in [CW00]. We introduce a normal form for our view definitions that suggests an initial (still exponential) search space for useful auxiliary view sets, and then we consider heuristic algorithms that explore various view sets in this search space.

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

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

- Instead of considering a doubly exponential search space of auxiliary views (as in [HRU96, Gup97, LQA97, RSS96]), or a very simple fixed set (as in [CW00]), we explore a "middle ground" based on our view definition normal form.

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

3 Preliminaries

We now introduce the relational materialized views we consider, as well as the processes of view maintenance and lineage tracing, using a running example. Along the way, we illustrate why materializing auxiliary views is important for view maintenance and lineage tracing, and why it is useful to consider the two problems together.

3.1 Materialized Views

To answer a variety of user queries efficiently, a data warehouse typically computes and stores a number of *materialized views* [LW95]. In this paper, we consider relational views with arbitrary use of aggregation, selection, projection, and join operators, which we call *ASPJ views*. We use an algebraic representation for the operators: α for grouping and aggregation, σ for selection, π for (duplicate-eliminating) projection, and $\bowtie$ for join. A view definition is presented using a rooted operator DAG with source tables at the leaves.

Any ASPJ view definition v can be transformed into an equivalent form v' composed of α-π-σ-$\bowtie$ operator sequences, by commuting and combining some select-project-join operators in the view definition [CWW97]. We call the resulting form v's *ASPJ normal form*, and we call each α-π-σ-$\bowtie$ sequence a *segment*. An example will be given shortly. In ASPJ normal form, a segment may omit the π, σ, or $\bowtie$ operator, but each segment except the topmost must include a non-trivial aggregation operator (or it would be merged with an adjacent segment). Since our view definitions are DAGs, they may contain multiple references to a source table or to a segment at any level.

We say that a view is an n-level ASPJ view if traversing from the root to any leaf in its normalized definition crosses at most n segments. The *fan-out* of a segment is the number of operands of the segment's join operator, or 1 if there is no join.

Example 3.1 (Materialized View and ASPJ Normal Form) Consider a data warehouse for a department store chain based on the following four tables, some or all of which may reside at remote source databases.

- `Store(store-id, city, expenses)` gives the city and monthly operating expenses of each store. We assume that each city contains at most one store, and that the operating expenses do not include employee salaries.

- `Product(product-id, price, cost)` gives the retail price and wholesale cost of each product item.

- `Sales(store-id, product-id, num)` gives the expected monthly number of sales for each product at each store.

- `Employee(emp-id, store-id, salary)` gives the monthly salary of each employee at each store.

Consider a materialized view `HighProfit` that keeps track of those cities whose stores are very profitable, i.e., whose monthly income exceeds expenses by at least $100,000. An SQL definition for `High-Profit` is shown in Figure 1, and its normalized view definition tree is shown in Figure 2. We use $\alpha_{G,aggr(A)}$ to represent grouping and aggregation, where G is a list of grouping attributes, and $aggr(A)$

```
CREATE VIEW HighProfit AS
SELECT city
FROM Store,
  (SELECT store-id, SUM(num*(price—cost)) AS profit
   FROM Sales, Product
   WHERE Sales.product-id = Product.product-id
   GROUP BY store-id) AS P,
  (SELECT store-id, SUM(salary) AS salaries
   FROM Employee
   GROUP BY store-id) AS E
WHERE Store.store-id = E.store-id
  AND E.store-id = P.store-id
  AND P.profit—E.salaries—Store.expenses > 100000
```

Figure 1: SQL definition for `HighProfit`

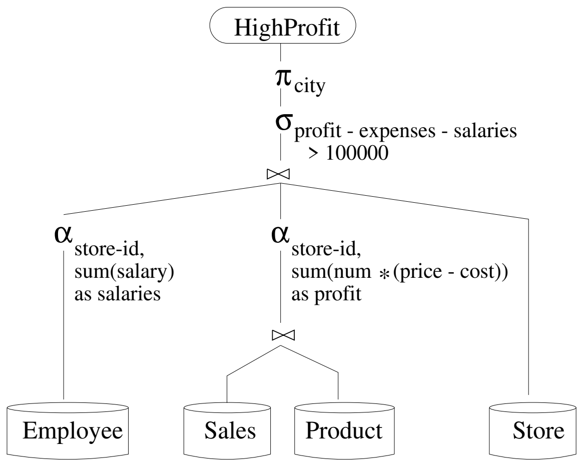

Figure 2: Normal form for `HighProfit`

abbreviates a list of aggregate functions over attributes in set A [CWW97].[1] `HighProfit` is a 2-level ASPJ view containing three segments: the topmost π-σ-$\bowtie$ segment with fan-out 3, the leftmost α segment with fan-out 1, and the middle α-$\bowtie$ segment with fan-out 2. $\square$

3.2 View Maintenance Procedures

Materialized views must be *maintained* to keep their contents up-to-date as the source tables they are defined over change. We assume a standard *incremental view maintenance* approach, as in [GMS93, Qua96]. Insertions and deletions to each source table are monitored and recorded in *delta tables* (Δ and ∇ respectively) in the warehouse. Updates are modeled as deletions followed by insertions. During view maintenance, changes to the view (also expressed as deltas) are computed based on the source delta tables, the view contents, and the source data, using a predefined sequence of queries and updates called the *maintenance procedure*. For 1-level ASPJ views we use the maintenance procedures from [GMS93, Qua96]. The following example shows a 1-level ASPJ view and its maintenance procedure.

Example 3.2 (View Maintenance Procedure) Consider the source tables from Example 3.1 and a 1-level ASPJ view `Profit` corresponding to the middle α-$\bowtie$ segment in Figure 2:

```
CREATE VIEW Profit AS
SELECT store-id, SUM(num*(price—cost)) AS profit
FROM Sales, Product
WHERE Sales.product-id = Product.product-id
GROUP BY store-id
```

Suppose the `Sales` table changes over time, and a set of insertions and deletions to the table are stored in delta tables Δ`Sales` and ∇`Sales`, respectively. The resulting changes to the view `Profit` (Δ`Profit`

[1] This operator is similar to the *generalized projection* of [GHQ95], but we distinguish between projection and aggregation operators because of the way our segments and auxiliary views are defined.

and ∇`Profit`) can be computed by the following maintenance procedure, which uses the *summary-delta* approach from [Qua96]:

```
SELECT store-id, SUM(profit) AS profit INTO SummaryDelta
FROM (SELECT store-id, (num*(price−cost)) as profit
      FROM ΔSales, Product
      WHERE ΔSales.product-id = Product.product-id)
     UNION
     (SELECT store-id, -1*(num*(price−cost)) as profit
      FROM ∇Sales, Product
      WHERE ∇Sales.product-id = Product.product-id)
GROUP BY store-id

SELECT * INTO ∇Profit
FROM Profit
WHERE store-id IN
     (SELECT store-id FROM SummaryDelta)

SELECT store-id, SUM(profit) INTO ΔProfit
FROM ∇Profit UNION SummaryDelta
GROUP BY store-id
```

□

For an n-level ASPJ view where $n > 1$, to compute the changes to the entire view we can compute the changes for one segment at a time using the maintenance procedure for 1-level views, propagating the deltas upward through the view definition DAG. Just as we needed source table `Product` along with Δ`Sales` and ∇`Sales` to compute the deltas for `Profit` in Example 3.2, to compute the deltas for a higher-level segment we may need deltas and/or full contents for each lower segment. For example, suppose we want to compute the deltas for `HighProfit` given Δ`Profit` and ∇`Profit`, where again `Profit` corresponds to the middle α-$\bowtie$ segment as in Example 3.2. We need to join the delta tables for `Profit` with source table `Store`, as well as with the leftmost α segment in Figure 2, then perform the selection and projection in the topmost segment. If we materialize an auxiliary view `Salary` corresponding to the leftmost α segment, we can significantly improve the performance of the maintenance procedure by avoiding recomputation of the aggregate values. In addition to materializing "intermediate" views, if source tables are remote and expensive (or impossible) to access, we may want to replicate some or all of the source tables as auxiliary views at the warehouse.

Note that our approach in this paper applies to all views, including those with non-incrementally maintainable aggregates (e.g., *min*, *max*). In the presence of such aggregates, the maintenance procedure must involve some recomputation, but auxiliary views may still be of benefit.

3.3 Lineage Tracing

Given a materialized view in a data warehouse, in addition to issuing regular queries or performing other kinds of analysis over the view, we may want to *trace the lineage* of selected "interesting" tuples in the view. The lineage of a view tuple is defined as the set of original source tuples that derived the given view tuple. To trace the lineage of a view tuple, we use a predefined sequence of queries called *tracing queries*

(TQs) [CWW97].

Given a 1-level ASPJ view V whose definition is $v = \alpha_{G,aggr(B)}(\pi_A(\sigma_C(T_1 \bowtie \cdots \bowtie T_m)))$, 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:

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

where $\mathbf{T_i}$, denotes the schema of table T_i, $i = 1..m$, and *Split* is an operator that breaks a table into multiple projections: $Split_{\mathbf{A_1},\ldots,\mathbf{A_m}}(T) = \langle \pi_{\mathbf{A_1}}(T), \ldots, \pi_{\mathbf{A_m}}(T) \rangle$.[2] Given a tuple set $T \subseteq V$, we can simultaneously trace all the tuples in T with:

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

Example 3.3 (Lineage Tracing Query) Consider the view `Profit` in Example 3.2. The lineage tracing query for a tuple $t \in$ `Profit` is

$$TQ_{t,\texttt{Profit}} = Split_{\texttt{Sales},\texttt{Product}}(\sigma_{\texttt{store-id}=t.\texttt{store-id}}(\texttt{Sales} \bowtie \texttt{Product}))$$

The SQL presentation of the query is as follows:

```
SELECT Sales.* INTO LN_Sales
FROM Sales, Product
WHERE Sales.product-id = Product.product-id AND Sales.store-id = t.store-id

SELECT Product.* INTO LN_Product
FROM Sales, Product
WHERE Sales.product-id = Product.product-id AND Sales.store-id = t.store-id
```

where `LN_Sales` and `LN_Product` contain the lineage of t according to view `Profit` in the source tables `Sales` and `Product`, respectively. □

To trace the lineage of an n-level ASPJ view where $n > 1$, 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 1-level ASPJ view to compute the lineage for the current traced tuples with respect to the intermediate views or source tables at the next level below. The necessary intermediate results can either be computed at tracing time, or we can materialize certain intermediate results as auxiliary views for the purpose of lineage tracing.

For example, to trace the lineage of a tuple t in view `HighProfit`, we logically define intermediate views `Salary` and `Profit` corresponding to the leftmost α segment and middle α-$\bowtie$ segment, respectively, in Figure 2. We trace the lineage of tuple t in `Salary`, `Profit`, and `Store`, producing $\langle \texttt{LN\_Salary}, \texttt{LN\_Profit}, \texttt{LN\_Store} \rangle$, using the following tracing query:

$$TQ = Split_{\texttt{Salary},\texttt{Profit},\texttt{Store}}(\sigma_C(\texttt{Salary} \bowtie \texttt{Profit} \bowtie \texttt{Store}))$$

where $C = \texttt{profit} - \texttt{expenses} - \texttt{salaries} > 100000 \wedge \texttt{city} = t.\texttt{city}$. Then, we further trace the lineage of the tuples in `LN_Salary` and `LN_Profit` in the source tables to produce `LN_Employee`,

[2] When we execute the tracing query, the selection condition $\sigma_{C \wedge G=t.G}$ is pushed down to individual T_i's whenever possible to improve tracing query performance.

LN_Sales, and LN_Product. As with view maintenance, materializing rather than recomputing intermediate results can significantly improve tracing performance. Also, since lineage tracing queries always return data from source tables by definition, replicating (portions of) the source data as auxiliary views at the warehouse may be advantageous, for the same reasons outlined in Section 3.2.

4 Auxiliary Views for View Maintenance and Lineage Tracing

As motivated in Section 3, it may be advantageous to materialize certain *auxiliary views* in a data warehouse to improve the performance of view maintenance and lineage tracing. View maintenance procedures and lineage tracing queries use the auxiliary views to avoid recomputations and expensive source queries, thereby reducing maintenance and query costs. There are many possible sets of auxiliary views to materialize. In this section, we first specify a number of potentially useful auxiliary views for arbitrary n-level ASPJ primary views (Section 4.1). We then discuss how view maintenance procedures and lineage tracing queries take advantage of the auxiliary views (Section 4.2). Finally, we formally define the auxiliary view selection problem and estimate the size of our search space (Section 4.3).

4.1 The Auxiliary Views We Consider

Let us first define two types of potentially useful auxiliary views, based on a single segment. (Similar auxiliary views were introduced in the context of SPJ primary views in [CW00].) Any segment can be thought of as a view definition $v = \alpha_{G,aggr(B)}(\pi_A(\sigma_C(T_1 \bowtie \cdots \bowtie T_m)))$, where each $T_1, \ldots, T_m$ is either a source table or a lower-level segment (view). Let V denote the materialization of v over $T_1, \ldots, T_m$.

1. *Lineage View (LV)* for v: We can store the intermediate result $LV(v) = \sigma_C(T_1 \bowtie \cdots \bowtie T_m)$ to help trace the lineage of tuples in V. We can rewrite the lineage tracing queries in Section 3.3 using $LV(v)$ as:

$$TQ_{t,v} = Split_{\mathbf{T_1},\ldots,\mathbf{T_m}}(\sigma_{G=t.G}(LV(v)))$$
$$TQ_{T,v} = Split_{\mathbf{T_1},\ldots,\mathbf{T_m}}(LV(v) \ltimes T)$$

   The maintenance procedure for V also can be simplified. If $LV(v)$ is materialized, then we compute $\Delta LV(v)$ and $\nabla LV(v)$, and the query for computing the *summary-delta* table in the maintenance procedure (Section 3.2) can be rewritten as:

$$\texttt{SummaryDelta} = \alpha_{G,aggr(B)}(\alpha_{G,aggr(B)}(\Delta LV) \cup \alpha_{G,-aggr(B)}(\nabla LV))$$

2. *Split Lineage Tables (SLTs)* for v: For a view (segment) v' whose joins is many-to-many, $LV(v')$ can be very large and inefficient to maintain. Thus, another possibility is to "split" the Lineage View and store a set of smaller tables: $SLT_i(v) = \pi_{\mathbf{T_i}}(\sigma_C(T_1 \bowtie \cdots \bowtie T_m)),\ i = 1..m$. The lineage tracing queries can then be rewritten using the SLTs as:

$$TQ_{t,v} = Split_{\mathbf{T_1},\ldots,\mathbf{T_m}}(\sigma_{G=t.G}(\sigma_C((SLT_1(v) \ltimes T) \bowtie \cdots \bowtie (SLT_m(v) \ltimes T)) \ltimes T))$$
$$TQ_{T,v} = Split_{\mathbf{T_1},\ldots,\mathbf{T_m}}(\sigma_C((SLT_1(v) \ltimes T) \bowtie \cdots \bowtie (SLT_m(v) \ltimes T)) \ltimes T)$$

Although these tracing queries look much more complex than with LV, performance can sometimes be much better due to the smaller size of the SLTs. Furthermore, as with LV, the maintenance procedure for V can use the deltas for the SLTs to be much more efficient. See [CW00] for details.

Given a general ASPJ view definition in normal form, in addition to considering Lineage Views and Split Lineage Tables for each segment, we also may consider storing copies of some or all of the source tables, to avoid expensive (remote) source queries during view maintenance and lineage tracing. We refer to these source table copies in the warehouse as *Base Tables* (BTs). Finally, maintaining the results of intermediate aggregations in the view (AGs) also can be very helpful in view maintenance and lineage tracing, as motivated in Section 3.

To summarize, we divide the normalized view definition into three types of components, and for each type of component we have certain choices of possible auxiliary views to materialize:

1. **Topmost Segment**: the segment at the root of the view definition DAG. Note that the α, π, σ, and/or $\bowtie$ operators (but not all of them) may be omitted in this segment. Also note that the topmost node corresponds to the primary view itself, so its contents are always materialized. We may further choose to materialize the Lineage View (LV) or the Split Lineage Tables (SLTs) for this segment, but not both. (If we store one, then storing the other will not further reduce the lineage tracing or overall maintenance cost.)

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

   (a) The contents of the α node (AG)

   (b) The Lineage View (LV) or the Split Lineage Tables (SLTs), but not both

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

Example 4.1 (Auxiliary Views) Recall our example view `HighProfit` from Figure 2. Figure 3 shows the three ASPJ segments in the view definition and all of the possible auxiliary views we consider materializing for `HighProfit`. □

Notice that because of our search space reduction, it is possible that there are useful auxiliary views we are not considering, notably different join combinations in the case of a many-way join. This special case is considered in detail in [LQA97], and we could extend our search space accordingly.

[3]Most existing data warehousing systems automatically store a copy of each source table in the warehouse. However, as we will see in Section 6, sometimes it is not beneficial to store a copy.

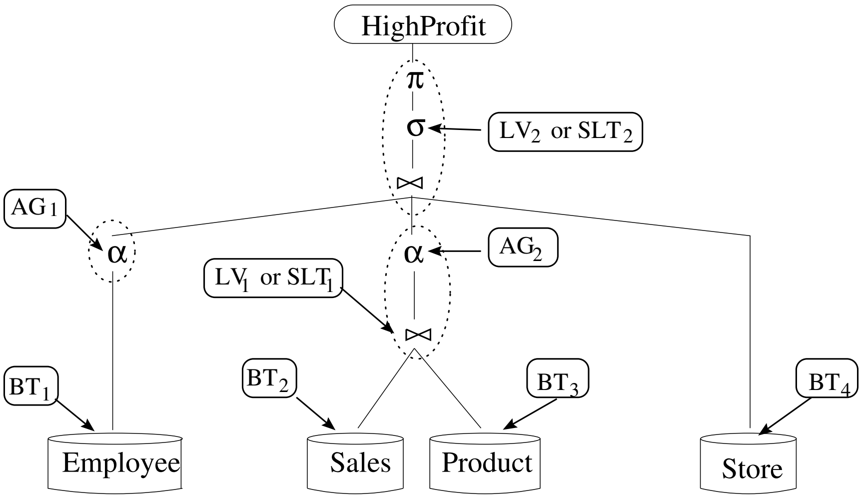

Figure 3: Possible auxiliary views for `HighProfit`

4.2 Using Auxiliary Views for View Maintenance and Lineage Tracing

In Section 4.1 we gave examples of how to rewrite queries for view maintenance and lineage tracing using auxiliary views. In general, when we have a set of auxiliary views available, there may be more than one way to rewrite a query to take advantage of auxiliary views. We assume that the "best" rewriting is selected, and this assumption is reflected in the cost model we present in Section 5.

As an example, the lineage tracing query TQ for a tuple t according to the topmost segment in the definition of `HighProfit` has the following rewritings using the auxiliary views in Figure 3:

$$TQ_1 = Split_{\texttt{Salary,Profit,Store}}(\sigma_{\texttt{profit}-\texttt{expenses}-\texttt{salaries}>\texttt{100000}\wedge\texttt{city}=t.\texttt{city}}(AG_1 \bowtie AG_2 \bowtie BT_4))$$
$$TQ_2 = Split_{\texttt{Salary,Profit,Store}}(\sigma_{\texttt{city}=t.\texttt{city}}(LV_2))$$
$$TQ_3 = Split_{\texttt{Salary,Profit,Store}}(\sigma_{\texttt{city}=t.\texttt{city}}(SLT_2^{\texttt{Salary}} \bowtie SLT_2^{\texttt{Profit}} \bowtie SLT_2^{\texttt{Store}}))$$

Suppose that LV_2, AG_1, and BT_4 are materialized. Then we could use query TQ_2, or (among other options) we could use a tracing query similar to TQ_1 that recomputes the contents of AG_2. In this case it is likely that TQ_2 would be chosen as the best query rewriting based on the available auxiliary views.

4.3 The View Selection Problem and the Search Space Size

We have shown that various auxiliary views can be used in the view maintenance and lineage tracing processes. Our goal is to select among the choices of auxiliary views described in Section 4.1 a set that minimizes overall cost: the cost of lineage tracing plus the cost of maintaining the primary and auxiliary views. Our cost model is described in Section 5. Here let us consider the size of our search space. Suppose we have an n-level ASPJ view in normal form, and consider a balanced view definition tree[4] with a fan-out of m in each segment. There is one topmost segment, and for that segment we have 3 auxiliary view

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

options: LV, SLTs, or nothing (case 1 in Section 4.1). There are $m^1 + m^2 + \cdots + m^{n-1} = O(m^{n-1})$ intermediate segments, each having 2 options for case 2(a) in Section 4.1 (AG or nothing) and 3 options for case 2(b) (LV, SLTs, or nothing). Finally, there are m^n source tables, each having 2 options: BT or nothing. Therefore, the size of the entire search space is

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

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

Example 4.2 (Search Space Size) Consider our example view `HighProfit` (Figure 3). The number of possible auxiliary view sets for `HighProfit` is $2^6 \cdot 3^2 = 384$. □

The number of choices for `HighProfit` is quite manageable. However, real warehouse views tend to have much higher fan-outs, as well as possibly more levels. As we will see in Section 7, even for a view with only 2 levels and average fan-out of 5, we cannot consider all possible auxiliary view sets due to the large search space.

5 Cost Model

In this section we present the model that we use to estimate view maintenance and lineage tracing costs for a given primary view and set of auxiliary views. The statistics our cost model relies on are listed in Table 1. Values for these statistics are set for each experiment, as described in Section 7. We briefly outline our cost estimation procedure as follows.

Let $cost(Q, s)$ denote the estimated cost of evaluating a query Q at the warehouse given a set of statistics s. Q could be a lineage tracing query, or a query or update in a view maintenance procedure. To compute $cost(Q, s)$ we use a fairly conventional cost model for relational queries in a distributed database setting, similar to, e.g., [LQA97, Ull89, ZGMHW95]. Details are omitted, but the cost formulas rely on all of the statistics from Table 1, and assume no indexes.

Suppose we have a primary view v and a set of auxiliary views $\mathcal{A} = \{v_1, ..., v_n\}$. To trace the lineage of tuples in the primary view given the materialized auxiliary views in $\mathcal{A}$, there are various possible rewritings of the lineage tracing queries using the auxiliary views (recall Section 4.2). Our cost model selects the sequence of lineage tracing queries with the lowest estimated cost. Let $q(v, \mathcal{A}, s)$ denote the estimated lineage tracing cost for a given primary view v, set of auxiliary views $\mathcal{A}$, and statistics s:

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

where $Q_1, \ldots, Q_m$ is the set of tracing queries selected for v given auxiliary view set $\mathcal{A}$. Note that the lineage query rate and the average number of tuples traced in a lineage query (part of our usage statistics in Table 1) are included in the input statistics s, and thus are incorporated into the lineage cost estimated by $q(v, \mathcal{A}, s)$.

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

Parameter name	Description
usage statistics (for each primary view)	
query_rate	# of tracing queries per unit time period
query_size	# of tuples traced per query
usage statistics (for each source table)	
update_rate	# of source table updates per unit time period
update_size	# of changed tuples per source table update
data statistics (for each source table)	
tuple_num	# of tuples in a source table
tuple_size	size of tuples in a source table (in bytes)
data statistics (for each view segment)	
fan-out	# of joined tables
join_ratio	# of joining tuples / # of tuples in cross-product
select_ratio	# of selected tuples / # of tuples before selection
proj_ratio	# of bytes projected / tuple size before projection
aggr_ratio	# of tuples in aggregate / # of tuples before aggregation
system statistics (for each source or warehouse)	
block_size	# of bytes in a block
disk_cost	cost to read/write a disk block (in ms/block)
net_cost	network transmission cost (in ms/byte)

Table 1: Statistics for cost estimation

estimated cost. Let $m(v, \mathcal{A}, s)$ denote the estimated maintenance cost for a given primary view v, set of auxiliary views $\mathcal{A}$, and statistics s:

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

where $M_1, \ldots, M_n$ is the set of maintenance queries and updates selected to maintain primary view v and the auxiliary views in $\mathcal{A}$. Note that the source table update rate and average number of source tuples changed in each update (part of our usage statistics in Table 1) are included in the input statistics s, and thus are incorporated into the maintenance cost estimated by $m(v, \mathcal{A}, s)$.

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

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

In our experiments, we measure the *optimality* of given sets of auxiliary views, by which we mean how close the sets of views come to the set that yields the lowest estimated cost. For a given primary view v and statistics s, let $\mathcal{A}_{opt}$ denote the set of auxiliary views within our search space (Section 4) with the lowest total cost $total\_cost(v, \mathcal{A}_{opt}, s)$. For a set of auxiliary views $\mathcal{A}$, we define the optimality of $\mathcal{A}$ as:

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

6 Algorithms for Selecting Auxiliary Views

Having defined the search space for the optimization problem and the cost model that we use, we now introduce four different algorithms for selecting a set of auxiliary views within the search space. The input

Parameter name	**Values**							
	HighProfit	Employee	Sales	Product	Store	segment 1	segment 2	segment 3
query_rate (#/unit time)	100							
query_size (tuples)	1							
update_rate (#/unit time)		10	10	10	0			
update_size (tuples)		1	100	1	0			
tuple_num		10000	1000000	100000	100			
tuple_size (bytes)		1000	500	500	400			
fan-out						1	2	3
join_ratio							0.0002	0.0001
select_ratio								0.1
proj_ratio						0.1	0.1	0.1
aggr_ratio						0.01	0.001	0.2
block_size (bytes)	8K	8K	8K	8K	8K			
disk_cost (ms/block)	1	1	1	1	1			
net_cost (ms/byte)	0	0.00001	0.0001	0.0001	0.0001			

Table 2: Statistics for view HighProfit

to each algorithm is the primary view definition v in ASPJ normal form, and a set of statistics s as specified in Section 5. The output is a set of auxiliary views $\mathcal{A}$.

6.1 Exhaustive Algorithm

The exhaustive algorithm enumerates all choices in the search space, estimates the cost of each choice, and picks the cheapest one. For our example view `HighProfit`, the exhaustive algorithm considers all 384 possible combinations of auxiliary views (recall Figure 3). We set a sample set of statistics s as shown in Table 2, including statistics for view `HighProfit`, source tables `Employee`, `Sales`, `Product`, and `Store`, as well as each ASPJ segment in the view definition (Figure 3). Over this set of statistics, the exhaustive algorithm selects $\mathcal{A} = \{BT_4, AG_1, AG_2, SLT_1, LV_2\}$. The exhaustive algorithm always finds the optimal auxiliary view set according to our cost model. However, the complexity of the algorithm is the same as the search space size: $O(2^k)$ where k is the number of components in the view definition (recall Section 4.3).

6.2 Naive Algorithm

At the other end of the spectrum, we consider a naive algorithm that selects a fixed set of auxiliary views: Lineage Views (LVs) for the topmost and all intermediate segments, aggregation results (AGs) for all intermediate segments, and all Base Tables (BTs). For example view `HighProfit`, the naive algorithm selects $\mathcal{A} = \{BT_1, BT_2, BT_3, BT_4, AG_1, AG_2, LV_1, LV_2\}$. Even though this naive fixed set of auxiliary views may not be optimal—in fact it can be arbitrarily bad compared to the optimal set—our experimental results in Section 7 show that the naive algorithm selects reasonably good view sets in many cases, especially considering its simplicity. The complexity of the naive algorithm is $O(1)$.

6.3 Greedy Algorithm

We also consider a conventional greedy algorithm. This algorithm initializes the auxiliary view set $\mathcal{A}$ to be empty. In each iteration, it adds into $\mathcal{A}$ the auxiliary view (not yet in $\mathcal{A}$) that brings the most benefit, i.e., reduces the total cost the most given the current set of views in $\mathcal{A}$. Iteration continues until there are no more auxiliary views outside $\mathcal{A}$ that can further reduce the total cost. For our example view `HighProfit` and the same sample statistics used in the exhaustive algorithm (Section 6.1), the greedy algorithm selects the optimal set of auxiliary views in the following order: AG_2, SLT_1, AG_1, LV_2, BT_4.

The greedy algorithm has complexity $O(k^2)$ instead of $O(2^k)$ as in the exhaustive algorithm, and it selects the optimal auxiliary view set in most cases (see Section 7). However, the greedy algorithm cannot guarantee an optimal answer, nor even an answer within some percentage of optimal. In Section 7.3, we will see a scenario where the greedy algorithm performs very poorly.

6.4 Three-Step Algorithm

Our last algorithm divides the auxiliary view selection process into three phases. See Figure 4. In the first phase, we use a greedy approach to add auxiliary views of the AG and BT types only. In the second phase, we decide for the topmost and each intermediate segment whether to add LV or SLTs. At this point, it may turn out that some of the AG or BT views selected in the first phase are no longer beneficial given the LV or SLT views selected in the second phase, and they incur maintenance cost. Thus, in third phase we remove AG and BT views that are not beneficial, and we do so in a greedy manner.

For example view `HighProfit` and the same statistics used in Sections 6.1 and 6.3, the three-step algorithm also selects the optimal set of auxiliary views. In phase 1, it selects AG and BT views in the following order: AG_2, AG_1, BT_2, BT_3, BT_4. In phase 2, it selects SLT_1 and LV_2. In phase 3, it removes BT_2 and BT_3 (in that order) because they are no longer beneficial given the views selected in phase 2.

The three-step algorithm has complexity $O(k^2)$, which is the same as the greedy algorithm, but its actual running time is less than the greedy algorithm by a linear factor. The first phase of the three-step algorithm is faster than the greedy algorithm since it only selects from the AG and BT views, instead of from all auxiliary views. The second phase is linear in the number of segments. The third phase only examines the AG and BT views selected in the first phase, which is a small number in most cases.

Like the greedy algorithm, the three-step algorithm usually selects the optimal set of auxiliary views (see Section 7). However, also like the greedy algorithm, the three-step algorithm cannot make any guarantees about the optimality of its answers. In Section 7.3 we will see a scenario where the three-step algorithm performs poorly. Interestingly, in the case we show where the three-step algorithm performs poorly, the greedy algorithm performs well, and vice-versa. Thus, one practical option is to combine the two algorithms: run both algorithms and select whichever answer has lower estimated cost. The running time of the combined algorithm remains $O(k^2)$.

7 Performance Study

In this section, we study the performance of the four algorithms specified in Section 6, comparing their running times and the optimality of the answers they produce. We also compare the cost of the answers produced by these algorithms against the cost of storing no auxiliary views. In Section 7.1, we present

```
input: primary view v, statistics s
output: auxiliary view set 𝒜
begin
    𝒜 ← ∅;

    // phase 1: use greedy algorithm on AG and BT nodes
    𝒜_all ← all possible auxiliary views for v;
    while true do
        for each v_i ∈ 𝒜_all of type AG or BT such that v_i ∉ 𝒜 do
            benefit_i ← total_cost(v, 𝒜, s) − total_cost(v, 𝒜 ∪ {v_i}, s);
        pick v_i with the highest benefit_i;
        if benefit_i ≤ 0 then break else 𝒜 ← 𝒜 ∪ {v_i};
    endwhile;

    // phase 2: decide LV and SLTs
    for the topmost and each intermediate segment do
        // Let LV and SLT be the Lineage View and Split Lineage Tables for the segment
        cost_1 ← cost(v, 𝒜 ∪ {LV}, s);
        cost_2 ← cost(v, 𝒜 ∪ {SLT}, s);
        if cost_1 ≥ cost_2 > cost(v, 𝒜, s) then 𝒜 ← 𝒜 ∪ {LV}
        else if cost_2 > cost_1 > cost(v, 𝒜, s) then 𝒜 ← 𝒜 ∪ {SLT};
    endfor;

    // phase 3: remove useless AGs and BTs
    while true do
        for each v_i ∈ 𝒜 of type AG or BT do
            benefit_i ← total_cost(v, 𝒜 − {v_i}, s) − total_cost(v, 𝒜, s);
        pick v_i with the lowest benefit_i;
        if benefit_i > 0 then break else 𝒜 ← 𝒜 − {v_i};
    endwhile;

    return 𝒜;
end
```

Figure 4: The three-step algorithm

results of experiments using the schema, statistics, and some views from the TPC-D benchmark. In Section 7.2, we present results of experiments using more complex synthetic view definitions. Since the greedy and three-step algorithms perform quite well in all of the experiments in Sections 7.1 and 7.2, in Section 7.3 we show experiments illustrating that greedy and three-step can perform poorly.

7.1 TPC-D Experiments

Our first set of experiments is based on the TPC-D benchmark [TPC96]. We use the schema of tables `Customer`, `Order`, `Lineitem`, `Supplier`, `Nation`, `Region`, `PartSupp`, and `Part` from the

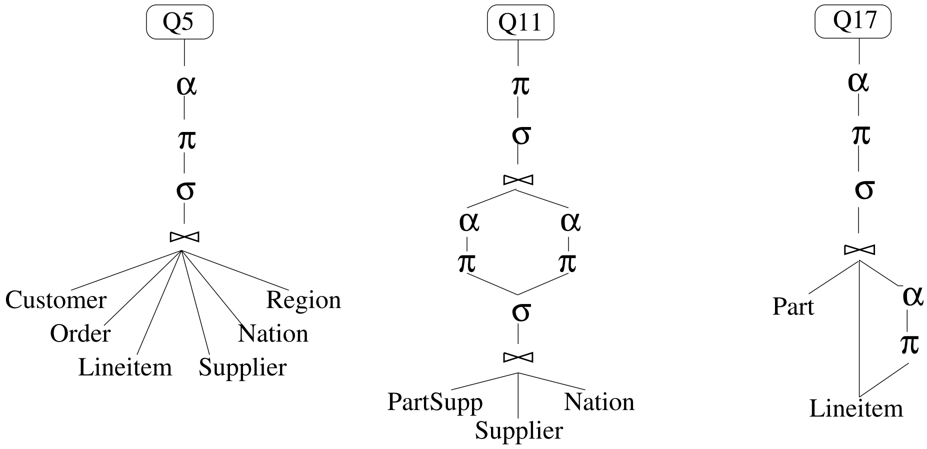

Figure 5: Materialized views for TPC-D experiments

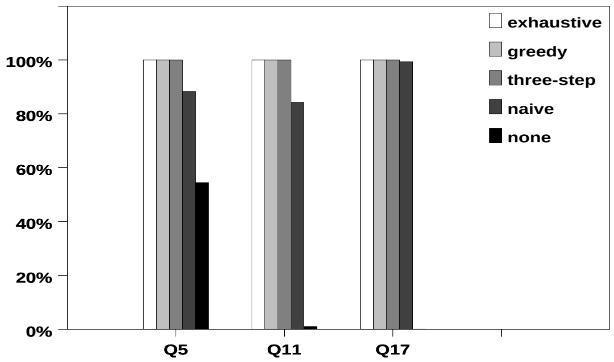

Figure 6: Optimality for TPC-D views

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

Figure 7: Running time in seconds for TPC-D

benchmark for our experiments. The table statistics we use correspond to a scaling factor of 1. The remaining statistics from Table 1 are set according to the benchmark and commonly used database and network system settings. For example, we set the update rate for the `Lineitem` and `Order` tables to be much higher than other tables, since `Lineitem` and `Order` are the *fact tables* according to the benchmark specification. For views, we select queries Q_5, Q_{11}, and Q_{17} from the benchmark, since they are relatively complex and differ somewhat from each other. In each case, we treat the benchmark query as the definition of our primary materialized view to be stored at the warehouse. The general structure of each of the three views is shown in Figure 5. The complete list of statistical settings (recall Table 1) for our three TPC-D experiments is shown in Appendix A, Tables 3–5.

Recall that we are comparing five algorithms—the four algorithms from Section 6, as well as the "algorithm" that selects no auxiliary views (which we call algorithm *none*). Figure 6 plots the optimality of the five algorithms for each of the TPC-D views we consider. Recall from Section 5 that optimality is defined as the cost of the optimal auxiliary view set divided by the cost of the chosen view set. Figure 7 plots the running time of the algorithms. Note that algorithms *naive* and *none* do incur a small running time, which is the time required to compute the cost of their one solution.

We can see from Figure 6 that storing no auxiliary views can be dramatically worse than storing some, even those selected by the naive algorithm. We also see from Figure 6 that the greedy and three-step

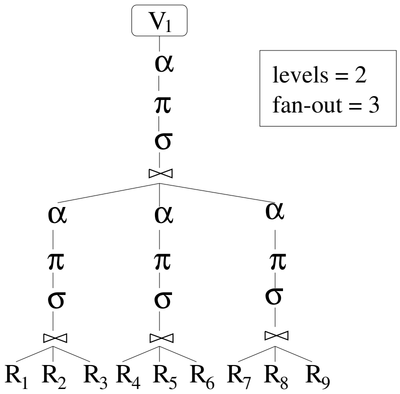

Figure 8: Structure of view v_1

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

Figure 9: Synthetic configurations

algorithms select the optimal auxiliary view set for all three views, and from Figure 7 we see that they do so in a small fraction of the running time required by the exhaustive algorithm. We also note that the three-step algorithm runs considerably faster than the greedy algorithm.

7.2 Synthetic Experiments

Our next set of experiments is conducted using synthetic views and data statistics. The views we consider all have a regular tree definition, as illustrated by view v_1 in Figure 8. We consider seven different views. Figure 9 summarizes the "shape" of each view definition tree (number of levels and fan-out of each segment), along with the query/update ratio, which represents the ratio of the average number of tracing queries per unit time to the average number of source updates (recall Table 1). The complete set of statistical settings for the seven experiments is summarized in Appendix A, Tables 6–8.

Figure 10 plots the optimality of our five algorithms for each of the seven synthetic views we consider. Figure 11 plots the running time of the algorithms. The greedy and three-step algorithms always select the optimal auxiliary view set or, in the one case of the three-step algorithm on v_1, very near to optimal. (Actually, for view v_7 the exhaustive algorithm never finished, so optimality is measured against the auxiliary view set selected by the greedy algorithm.) The greedy and three-step algorithms find their answer in a small fraction of the running time required by the exhaustive algorithm, and three-step is much faster than greedy. Another interesting result is that algorithm *none* performs better when the query/update ratio is lower (experiments v_3–v_5). However, we should not infer that the benefit of auxiliary views is primarily for lineage tracing. In fact, in experiment v_5 the query/update ratio is set to 0 (indicating view maintenance only), and we still see significant benefit to using auxiliary views.

Next, we consider in more detail how the running times of the exhaustive, greedy, and three-step algorithms are affected by view complexity. In Figure 12, we consider views where we fix the number of levels at 2 and increase the fan-out from 1 to 9. The exhaustive, greedy, and three-step algorithms become prohibitive when the fan-out exceeds 3, 7, and 8, respectively. We see similar behavior in Figure 13, where we fix the fan-out at 2 and increase the number of levels from 1 to 8.

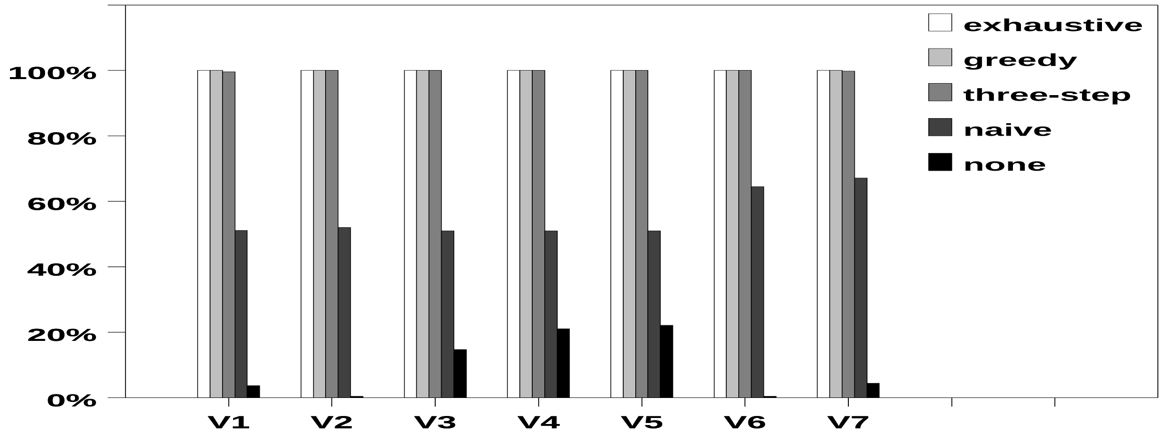

Figure 10: Optimality for synthetic views

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

Figure 11: Running time in seconds for synthetic views

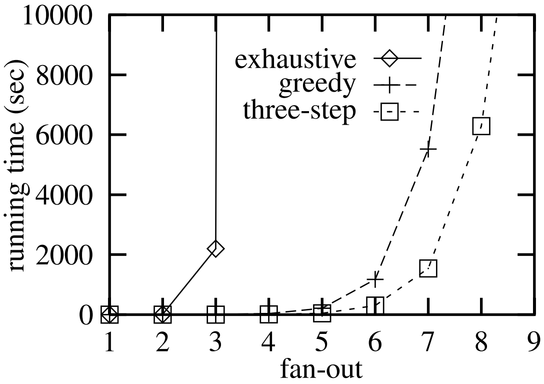

Figure 12: Running time vs. fan-out

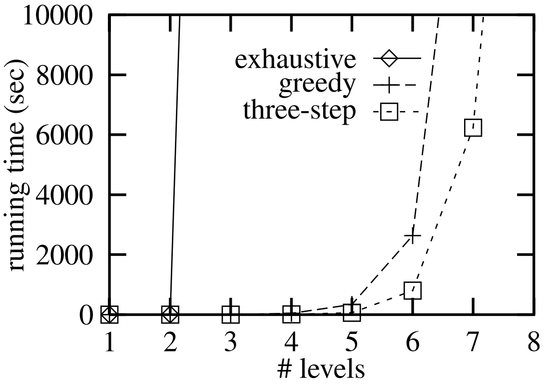

Figure 13: Running time vs. # levels

7.3 When Greedy and Three-Step Fail

The greedy and three-step algorithms select the optimal (or in one case very near to optimal) auxiliary view set in all of the experiments reported in Sections 7.1 and 7.2. However, there are cases in which these algorithms fail to pick an optimal or even near-optimal answer.

Consider a simple view definition v and the auxiliary views that are considered for v in Figure 14. By setting the data statistics (Table 1) to different values, we can vary the costs and benefits of the four different auxiliary views. We have set two different configurations, which we call v_8 and v_9, both based on

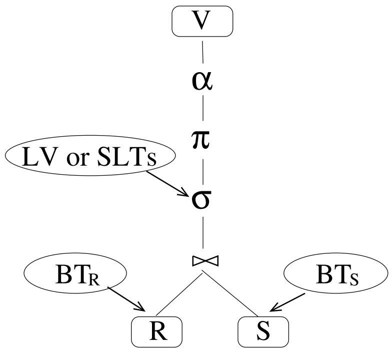

Figure 14: View structure used for v_8 and v_9

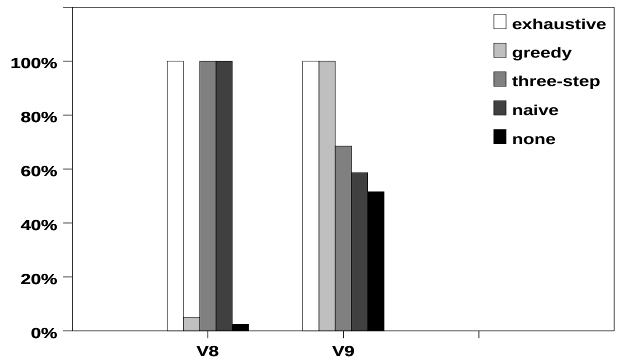

Figure 15: Optimality for v_8 and v_9

the view in Figure 14. The complete set of statistical settings for these two experiments is summarized in Appendix A, Tables 9 and 10. In particular, we set the source table network costs for v_8 to be much higher than for v_9, and we set the join ratio of v_8 higher (less selective) than v_9. The optimal set of auxiliary views for v_8 is $\{BT_R, BT_S\}$, and the optimal set for v_9 is $\{LV\}$.

Figure 15 plots the optimality of all five algorithms on views v_8 and v_9. In particular, the greedy algorithm performs extremely poorly on v_8 (selecting $\{LV\}$ instead of $\{BT_R, BT_S\}$), while the three-step algorithm misses the optimal solution for v_9 (selecting $\{BT_R, BT_S\}$ instead of $\{LV\}$). However, as suggested in Section 6, if we use a combined algorithm that runs both greedy and three-step and then selects the lower-cost solution, we will select the optimal view set for both v_8 and v_9.

8 Conclusion

We have examined the problem of selecting auxiliary views to materialize in a data warehouse in order to reduce the cost of view maintenance and lineage tracing for complex primary views. We specify a normal form for view definitions and use it to define an initial search space of potentially beneficial auxiliary views. We presented four algorithms for exploring the search space and selecting a set of auxiliary views: *exhaustive*, *greedy*, *three-step*, and *naive*. We compared the optimality and running time of the algorithms using experiments based on the TPC-D benchmark, as well as on a variety of synthetic views and statistics.

Our experiments indicate that in terms of running time and optimality, the three-step algorithm appears to be the best, although the running time of the greedy algorithm probably also is fast enough in practice for most complex warehouse views. (The exhaustive algorithm, on the other hand, becomes intractable quite quickly.) Both the greedy and three-step algorithms find the optimal auxiliary view set in most cases, although we have shown (complementary) situations in which either one algorithm or the other performs poorly. Our experiments also illustrate that even a naive selection of auxiliary views reduces overall cost dramatically in most cases, underscoring the importance of materializing auxiliary views for the dual purposes of view maintenance and lineage tracing in a warehousing environment.

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

References

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

[CW91] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proc. of the International Conference on Very Large Databases*, pages 577–589, Barcelona, Spain, September 1991.

[CW00] Y. Cui and J. Widom. Practical lineage tracing in data warehouses. To appear in *Proc. of the Sixteenth International Conference on Data Engineering*, San Diego, California, February 2000.

[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.

[GHQ95] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. In *Proc. of the Twenty-First International Conference on Very Large Data Bases*, pages 358–369, Zurich, Switzerland, September 1995.

[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.

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

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

[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.

[LYC+99] W. Labio, J. Yang, Y. Cui, H. Garcia-Molina, and J. Widom. Performance issues in incremental warehouse maintenance. Technical report, Stanford University Database Group, October 1999. Available at http://www-db.stanford.edu/pub/papers/whips-wm.ps.

[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.

[TPC96] Transaction Processing Performance Council. *TPC-D Benchmark Specification, Version 1.2*, 1996. Available at: `http://www.tpc.org/`.

[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.

[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.

A Statistical Settings for Experiments

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

Table 3: Statistics for Q_5

Parameter name	**Values**						
	Q_{11}	PartSupp	Supplier	Nation	segment 1	segment 2	segment 3
query_rate (#/unit time)	100						
query_size (tuples)	1						
update_rate (#/unit time)		10	1	0			
update_size (tuples)		1	1	0			
tuple_num		800000	10000	25			
tuple_size (bytes)		100	200	100			
fan-out					1	3	3
join_ratio						0.0006	0.0006
select_ratio					0.1	0.04	0.04
proj_ratio					0.6	0.1	0.1
aggr_ratio						0.005	0.005
block_size (bytes)	8K	8K	8K	8K			
disk_cost (ms/block)	1	1	1	1			
net_cost (ms/byte)	0	0.0001	0.0001	0.0001			

Table 4: Statistics for Q_{11}

Parameter name	Values				
	Q_{17}	Part	Lineitem	segment 1	segment 2
query_rate (#/unit time)	100				
query_size (tuples)	1				
update_rate (#/unit time)		10	10		
update_size (tuples)		1	10		
tuple_num		200000	6000000		
tuple_size (bytes)		200	300		
fan-out				3	1
join_ratio				0.0002	
select_ratio				0.002	
proj_ratio				0.1	0.2
aggr_ratio				0.001	0.03
block_size (bytes)	8K	8K	8K		
disk_cost (ms/block)	1	1	1		
net_cost (ms/byte)	0	0.0001	0.00001		

Table 5: Statistics for Q_{17}

Parameter name	Values						
	v_1	v_2	v_3	v_4	v_5	each source table	each segment
query_rate (#/unit time)	1000	100	10	1	0		
query_size (tuples)	1	1	1	1	1		
update_rate (#/unit time)						10	
update_size (tuples)						1	
tuple_num						10000	
tuple_size (bytes)						100	
levels	2	2	2	2	2		
fan-out							3
join_ratio							0.001
select_ratio							0.2
proj_ratio							0.4
aggr_ratio							0.1
block_size (bytes)	8K	8K	8K	8K	8K	8K	
disk_cost (ms/block)	1	1	1	1	1	1	
net_cost (ms/byte)	0	0	0	0	0	0.0001	

Table 6: Statistics for synthetic views v_1–v_5

Parameter name	**Values**		
	v_6	each source table	each segment
query_rate (#/unit time)	100		
query_size (tuples)	1		
update_rate (#/unit time)		10	
update_size (tuples)		1	
tuple_num		1000000	
tuple_size (bytes)		1000	
levels	6		
fan-out			1
join_ratio			
select_ratio			0.5
proj_ratio			0.8
aggr_ratio			0.2
block_size (bytes)	8K	8K	
disk_cost (ms/block)	1	1	
net_cost (ms/byte)	0	0.0001	

Table 7: Statistics for v_6

Parameter name	**Values**		
	v_7	each source table	each segment
query_rate (#/unit time)	100		
query_size (tuples)	1		
update_rate (#/unit time)		10	
update_size (tuples)		1	
tuple_num		10000	
tuple_size (bytes)		100	
levels	2		
fan-out			5
join_ratio			0.001
select_ratio			0.2
proj_ratio			0.4
aggr_ratio			0.1
block_size (bytes)	8K	8K	
disk_cost (ms/block)	1	1	
net_cost (ms/byte)	0	0.0001	

Table 8: Statistics for v_7

Parameter name	**Values**			
	v_8	R_1	R_2	segment 1
query_rate (#/unit time)	100			
query_size (tuples)	1			
update_rate (#/unit time)		0	0	
update_size (tuples)		0	0	
tuple_num		100000	100000	
tuple_size (bytes)		1000	1000	
fan-out				2
join_ratio				0.001
select_ratio				0.2
proj_ratio				0.6
aggr_ratio				0.2
block_size (bytes)	8K	8K	8K	
disk_cost (ms/block)	1	1	1	
net_cost (ms/byte)	0	0.01	0.01	

Table 9: Statistics for v_8

Parameter name	**Values**			
	v_9	R_1	R_2	segment 1
query_rate (#/unit time)	100			
query_size (tuples)	1			
update_rate (#/unit time)		0	10	
update_size (tuples)		0	20	
tuple_num		10000	1000000	
tuple_size (bytes)		1000	1000	
fan-out				2
join_ratio				0.00003
select_ratio				0.2
proj_ratio				0.6
aggr_ratio				0.2
block_size (bytes)	8K	8K	8K	
disk_cost (ms/block)	1	1	1	
net_cost (ms/byte)	0	0.0001	0.001	

Table 10: Statistics for v_9