Performance Issues in Incremental Warehouse Maintenance*

Wilburt Juan Labio, Jun Yang, Yingwei Cui, Hector Garcia-Molina, Jennifer Widom
Computer Science Department, Stanford University
{wilburt,junyang,cyw,hector,widom}@db.stanford.edu
http://www-db.stanford.edu/warehousing/

Abstract

A well-known challenge in data warehousing is the efficient incremental maintenance of warehouse data in the presence of source data updates. In this paper, we identify several critical data representation and algorithmic choices that must be made when developing the machinery of an incrementally maintained data warehouse. For each decision area, we identify various alternatives and evaluate them through extensive experiments. We show that picking the right alternative can lead to dramatic performance gains, and we propose guidelines for making the right decisions under different scenarios. All of the issues addressed in this paper arose in our development of WHIPS, a prototype data warehousing system supporting incremental maintenance.

1 Introduction

Data warehousing systems integrate and store data from remote sources as *materialized views* in the warehouse [LW95, CD97]. When source data changes, warehouse views need to be *maintained* so that they remain consistent with the source data. Commercial data warehousing systems typically recompute all warehouse views periodically to keep them up to date, but this process can be very expensive for large views. In contrast, with *incremental* maintenance, only the portions of the views that have changed are actually modified [GM95]. Because of the potential performance advantage, incremental view maintenance has recently found its way into commercial systems, e.g., [FS96, BDD$^+$98, BS99].

In this paper we experimentally study various options for incremental maintenance when the warehouse data is stored in an off-the-shelf commercial database management system (DBMS). For instance, we investigate how views are best stored in the DBMS, how aggregate views should be maintained, how deletions can be handled by the DBMS, how parameters such as memory and base relation size impact our choices, and several other issues. To illustrate the types of questions we address, we briefly introduce one of the options faced by the warehouse implementor: how to represent views.

Example 1.1 A warehouse often must support bag (duplicate) semantics. For instance, to efficiently maintain complex views involving aggregates, we also need to maintain auxiliary views that may contain duplicates. Furthermore, bag semantics can simplify incremental maintenance, as we will see below. How should views be represented when bag semantics are called for? If view V contains three copies of tuple t, should we explicitly store those three copies? Or should we add a *dupcnt* attribute to V to record the number of copies of t? For each case, how should the DBMS insert or delete new tuples?

To illustrate, consider a concrete but *very simple* example. Consider a source table $R(K, A_1, A_2, ..., A_n)$ with key attribute K, and a simple warehouse view $V(A_1, A_2, ..., A_n)$ defined over R which projects out the key of R. Without bag semantics, maintaining V would be expensive: when a tuple $\langle k, a_1, a_2, ..., a_n\rangle$ is deleted from R, we cannot decide whether to delete $\langle a_1, a_2, ..., a_n\rangle$ from V without querying R, because

*This work was supported by the National Science Foundation under grant IIS-9811947, by NASA Ames under grant NCC2-5278, and by Sagent Technology Inc.

there might exist another tuple $\langle k', a_1, a_2, ..., a_n \rangle$ in R that also derives $\langle a_1, a_2, ..., a_n \rangle$ in V. On the other hand, if duplicates are preserved in V, we should always delete one tuple from V for each tuple deleted from R, and no R queries are required.

Suppose that we implement bag V by keeping explicit copies of tuples. Suppose further that based on the deletions from R we have computed $\triangledown V$, a bag of tuples to be deleted from V. To apply $\triangledown V$ to V, one might be tempted to use the following SQL statement:

DELETE FROM V WHERE (A_1, A_2, ..., A_n) IN (SELECT * FROM $\triangledown V_1$)

Unfortunately, this statement does not work because SQL `DELETE` always removes *all* tuples satisfying the `WHERE` condition. If V has three copies of a tuple t and $\triangledown V$ contains two copies of t, the above `DELETE` statement will delete all three copies from V, instead of correctly leaving one. To properly apply $\triangledown V$, we need to use a cursor on $\triangledown V$ (details will be provided later). However, a cursor-based implementation forces $\triangledown V$ to be processed one tuple at a time.

Given this problem with deletion, we may want to consider the *dupcnt* approach, where we store only one copy for each tuple, together with an extra attribute to record the number of duplicates for that tuple. Under this representation, $\triangledown V$ can be applied in batch with two SQL statements (again, details will be given later). Besides the obvious advantage of being more compact when the number of duplicates is large, how does this count representation compare with the default duplicate representation? In particular, does it speed up overall view maintenance? Are SQL statements really better than a cursor loop for applying $\triangledown V$? $\square$

In this paper we study several issues like the ones illustrated in Example 1.1. For each issue we propose various alternatives, including interesting new variations for aggregate view maintenance that turn out to have important advantages over previous algorithms. In many of the decision areas we discuss, making a wrong decision can severely hamper the efficiency of warehouse maintenance. For example, as we will see in our experiments, the time required to install changes into views can vary by orders of magnitude depending on how maintenance is implemented: as data volumes grow, picking the right strategy can mean the difference between a few minutes and many hours of warehouse maintenance time. Based on the results of our experiments, we provide guidelines for making the right decisions under different scenarios.

For our experiments we use WHIPS (W*are*H*ouse* I*nformation* P*rocessing* S*ystem*), a prototype data warehousing system at Stanford [WGL$^+$96]. (In fact, most of the design issues we consider in this paper arose in the design and implementation of WHIPS.) Because WHIPS is representative of warehousing infrastructures built using commercial DBMS, we believe that our results have applicability well beyond WHIPS—they should prove helpful to any implementation of incremental warehouse maintenance either within or on top of a commercial DBMS.

In closing our introduction, we make two points. First, even though incremental maintenance has enjoyed considerable attention from the research community [GM95], very little research to date covers practical implementation issues backed up by thorough experiments. Thus we believe that our paper makes a unique contribution in this regard. Second, experimental evaluations always raise many questions, especially if the evaluations involve commercial products. Was "enough" memory used? Were the databases studied "big enough?" Should the query optimizer be hand-tuned to maximize performance? Will next year's DBMS invalidate the conclusions because it has a snazzy new feature? As questions like these arise in our paper, keep in mind that our goal is *not* to make absolute performance predictions, but rather to understand the choices and tradeoffs involved. We study incremental maintenance in a realistic, off-the-shelf

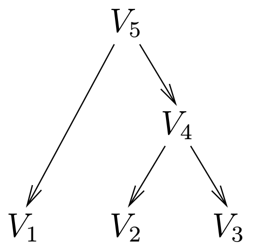

Figure 1: Conceptual organization.

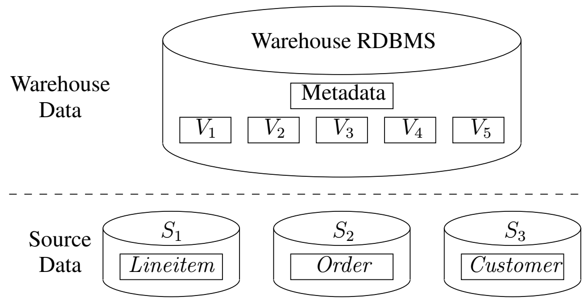

Figure 2: Physical organization.

scenario, and we have varied many of the parameters involved (e.g., memory and database size). Of course there are other interesting scenarios and more questions that can follow our initial study.

The rest of the paper is organized as follows. In Section 2, we give an overview of the WHIPS architecture, which sets the stage for later discussions. In Section 3, we focus on the component of WHIPS responsible for warehouse maintenance and discuss the various choices in building its view maintenance machinery. In Section 4, we conduct experiments to evaluate each alternative and present guidelines for making the best choices. Finally, we discuss related work in Section 5 and conclude the paper in Section 6.

2 WHIPS Architecture

The warehouse views incrementally maintained by WHIPS are derived from one or more independent and usually remote data sources. Data in the warehouse is modeled conceptually using a *view directed acyclic graph* (*VDAG*). Each node in the graph represents a materialized view stored at the warehouse. An edge $V_j \rightarrow V_i$ indicates that view V_j is defined over view V_i. A node with no outgoing edges represents a view that is defined over source data. WHIPS requires that each warehouse view V is defined either only over source data, or only over other warehouse views, because views defined over source data require special algorithms for ensuring their consistency [ZGMHW95]. We call views defined over source data *base views*, and views defined over other warehouse views *derived views*. (In a typical OLAP-oriented data warehouse, fact tables and dimension tables would be modeled as base views, while summary tables would be modeled as derived views.) Figure 1 shows a simple example of a VDAG with three base views V_1, V_2, and V_3, and two derived views V_4 and V_5. The source data from which the base views V_1, V_2, and V_3 are derived is not represented in the VDAG.

The top half of Figure 2 shows that WHIPS physically stores views V_1–V_5 from Figure 1 as tables in a relational DBMS, along with metadata that records each view's definition. The metadata also stores information about the source data from which the base views are derived. In WHIPS, each base view is defined over source data using a single SQL `SELECT-FROM-WHERE` (SFW) statement. This simple base view definition language allows the warehouse designer to filter and combine source data using appropriate selection and join conditions in the `WHERE` clause. Currently, aggregation is not permitted in base view definitions because it is difficult to ensure the consistency of aggregates over remote source relations [ZGMHW95].

Each derived view is defined over other warehouse views using one or more SQL `SELECT-FROM-WHERE-GROUP-BY` (SFWG) statements, where aggregation is permitted. Multiple SFWG statements may be combined using the SQL `UNION ALL` operator.

Example 2.1 As a concrete example, let us suppose that there are three remote information sources S_1, S_2, and S_3 as shown in the bottom half of Figure 2, exporting the tables *Lineitem*, *Order*, and *Customer*,

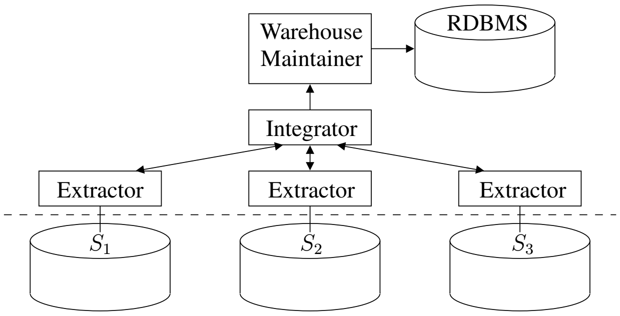

Figure 3: WHIPS components.

respectively. The schema of these tables is loosely based on the TPC-D benchmark [TPC96], but simplified for succinctness of presentation. (All our experiments in Section 4 strictly follow the TPC-D schema.) Base views V_1, V_2, and V_3 at the warehouse could be defined as projections over $S_1.Lineitem$, $S_2.Order$, and $S_3.Customer$ as follows:

```
CREATE VIEW V1 AS SELECT orderID, partID, qty, cost FROM S1.Lineitem
CREATE VIEW V2 AS SELECT orderID, custID, date FROM S2.Order
CREATE VIEW V3 AS SELECT custID, name, address FROM S3.Customer
```

Of course, selection and join operations may be used in base view definitions as well. Derived view V_4 could be defined to count the number of orders each customer has made in 1998:

```
CREATE VIEW V4 AS
SELECT custID, COUNT(*) FROM V2, V3
WHERE V2.custID = V3.custID AND V2.date >= '1998-01-01' AND V2.date < '1999-01-01'
GROUP BY custID
```

□

Three types of components comprise the WHIPS system: the *Extractors*, the *Integrator*, and the *Warehouse Maintainer*. As mentioned previously, WHIPS also relies on a commercial relational DBMS to store and process warehouse data. The WHIPS components, along with the DBMS, are shown in Figure 3. We discuss the components by walking through how warehouse data is maintained when source data changes.

Each *Extractor* component periodically detects *deltas* (insertions, deletions, and/or updates) in source data. One Extractor is used for each information source. For instance, in Figure 3, the Extractor assigned to S_1 detects the changes to the $Lineitem$ table which resides in S_1.

The *Integrator* component receives deltas detected by the Extractors, and computes a consistent set of deltas to the base views stored in the warehouse. The Integrator may need to send queries back to the sources to compute base view deltas. For details, see [WGL$^+$96, ZGMHW95].

The *Warehouse Maintainer* component receives the base view deltas from the Integrator and computes a consistent set of deltas to the derived views. The Warehouse Maintainer then updates all of the warehouse views based on the provided and computed deltas. To compute the derived view deltas and update the materialized views, the Warehouse Maintainer sends a sequence of Data Manipulation Language (DML) commands to the DBMS. These DML commands include SQL queries for computing the deltas, as well as modification statements (e.g., `INSERT`, `DELETE`, cursor updates) for updating the materialized views.

The interaction among the WHIPS components when warehouse data is first loaded is similar to the description above for maintenance. First, the Extractors identify the source data needed to load the warehouse based on the base view definitions. The Integrator then computes a consistent set of initial base views using

orderID	*partID*	*qty*	*cost*
1	*a*	1	20
1	*b*	2	250
1	*a*	1	20

Figure 4: DUP representation (V_1^{DUP}).

orderID	*partID*	*qty*	*cost*	*dupcnt*
1	*a*	1	20	2
1	*b*	2	250	1

Figure 5: CNT representation (V_1^{CNT}).

the source data from the Extractors. The Warehouse Maintainer in turn computes a consistent set of initial derived views, and then sends a sequence of Data Definition Language (DDL) and DML commands to the DBMS. The DDL commands (e.g., `CREATE TABLE`) are used for creating the materialized views. The DML commands are used for initially populating the materialized views.

The remainder of the paper focuses on the Warehouse Maintainer component. We refer the reader to [LGM96] and [ZGMHW95] for extended discussions of the Extractors and the Integrator, respectively.

3 The Warehouse Maintainer

The Warehouse Maintainer is the component responsible for initializing and maintaining the warehouse views. There are many possible ways of representing the warehouse views and performing incremental maintenance on them. In Sections 3.1 and 3.2, we identify specific important decision areas. For each decision area, we propose several alternatives and analyze them qualitatively. Quantitative performance results will be presented in Section 4. We have decided to separate the performance results from the discussion of the decision areas (rather than mixing them), because some of the decision areas are interrelated and it is important to have a complete overview of the issues before we delve into the detailed performance analysis.

3.1 View Representation and Delta Installation

Views in WHIPS are defined using SQL SFWG statements (for derived views) and SFW statements (for base views) with bag semantics. There are two ways to represent a bag of tuples in a view. One representation, which we call the DUP representation, simply keeps the duplicate tuples, as shown in Figure 4 for a small sample of data in view V_1 from Example 2.1. Another representation, which we call the CNT representation, keeps one copy of each unique tuple and stores the number of duplicates in a special *dupcnt* attribute, as in Figure 5. Let us denote a view V's DUP representation as V^{DUP} and its CNT representation as V^{CNT}. Next, we compare the two representations in terms of their storage costs and implications for queries and view maintenance.

3.1.1 Storage Cost and Query Performance

The CNT representation of a view V has lower storage cost if there are many duplicates in V and if the tuples of V are large enough that the storage overhead of having a *dupcnt* attribute is not significant. The reduction in storage achieved by using V^{CNT} instead of V^{DUP} may speed up selection, join, and aggregation queries over V by reducing I/O. However, projections may be slower when the CNT representation is used. Consider the simple operation of listing the *orderID*'s in V_1:

SELECT *orderID* FROM V_1

If we operate directly on the CNT representation V_1^{CNT} (Figure 5), the answer to the above query will not remain in the CNT representation. We need to group the tuples with matching *orderID*'s and sum up their *dupcnt* values:

```
SELECT orderID, SUM(dupcnt) AS dupcnt FROM V_1^CNT GROUP BY orderID
```

In general, whenever projection is used in a query, aggregation may be necessary to produce an answer in the CNT representation.

3.1.2 Deletion Installation

If V has no duplicates, then the deletions from V, denoted $\bigtriangledown V$, can be installed using a single `DELETE` statement. For example, to install $\bigtriangledown V_1$ in our working example, we use:

```
DELETE FROM V_1 WHERE (V_1.orderID, V_1.partID) IN (SELECT orderID, partID FROM ▽V_1)
```

The above statement assumes that $\{orderID, partID\}$ is a key for V_1. The `WHERE` clause can be modified to handle keys with any number of attributes, where in the worst case, all attributes of the view together form a key. Unfortunately, when V_1 has duplicates and hence no key, the above `DELETE` statement is incorrect because it may delete more tuples than intended, as discussed in Example 1.1.

In general, care must be taken when installing $\bigtriangledown V$ in the presence of duplicates. Under the DUP representation, a cursor on $\bigtriangledown V^{\text{DUP}}$ is required. For each tuple t in $\bigtriangledown V^{\text{DUP}}$ examined by the cursor, we delete one and only one tuple in V^{DUP} that matches t. Doing so generally requires another cursor on V^{DUP}. However, if the DBMS provides some mechanism of restricting the number of rows processed by a statement (such as allowing statements to reference row counts or tuple ID's), then we can avoid the additional cursor on V^{DUP}.

Under the CNT representation, each deleted tuple t in $\bigtriangledown V^{\text{CNT}}$ results in either an update to or a deletion from V^{CNT}. If $t.dupcnt$ is less than the $dupcnt$ value of the tuple in V^{CNT} that matches t, we decrement the matching V^{CNT} tuple's $dupcnt$ value by $t.dupcnt$. Otherwise, we delete the matching tuple from V^{CNT}. This procedure can be implemented with a cursor on $\bigtriangledown V^{\text{CNT}}$. Alternatively, the entire $\bigtriangledown V^{\text{CNT}}$ can be processed in batch with one `UPDATE` statement and one `DELETE` statement, but both statements contain potentially expensive correlated subqueries. The `DELETE` statement is illustrated below. The `UPDATE` statement is twice as long, with one correlated subquery in its `WHERE` clause and one in its `SET` clause. We omit the details due to space constraints.

```
DELETE FROM V^CNT
WHERE EXISTS (SELECT * FROM ▽V^CNT
              WHERE ▽V^CNT.orderID = V^CNT.orderID
              AND   ▽V^CNT.partID = V^CNT.partID
              AND   ▽V^CNT.dupcnt >= V^CNT.dupcnt)
```

We could eliminate the `DELETE` statement by using a row-level trigger that automatically deletes any tuple in V with $dupcnt$ less than or equal to 0. Under this approach, we only need one `UPDATE` statement to decrement the $dupcnt$ values of all V tuples with matching $\bigtriangledown V^{\text{CNT}}$ tuples. The trigger is fired for each updated tuple that satisfies the trigger condition.

3.1.3 Insertion Installation

Under the DUP representation, we can install the insertions into V^{DUP}, denoted $\bigtriangleup V^{\text{DUP}}$, using a single straightforward SQL `INSERT` statement. Under the CNT representation, we can install $\bigtriangleup V^{\text{CNT}}$ with a single `INSERT` statement only if we know that V never contains any duplicates. In general, however, each tuple t in $\bigtriangleup V^{\text{CNT}}$ results in either an update or an insertion to V^{CNT}. If there is a tuple in V^{CNT} that matches t, we increment the matching V^{CNT} tuple's $dupcnt$ value by $t.dupcnt$. Otherwise, we insert t into V^{CNT}.

Again, this procedure can be implemented with a cursor on $\triangle V^{\text{CNT}}$, or we can process the entire $\triangle V^{\text{CNT}}$ in batch with one `UPDATE` statement and one `INSERT` statement, but again, both statements contain potentially expensive correlated subqueries.

3.1.4 Discussion

Intuitively, for a warehouse view that never contains any duplicates (i.e., it has a known key), the DUP representation should outperform the CNT representation in all metrics (storage cost, query performance, and delta installation time) because the DUP representation does not have the overhead of one *dupcnt* attribute per tuple. On the other hand, for a view with many duplicates, we would expect the CNT representation to be preferable because it is more compact. For WHIPS, we are interested in knowing, quantitatively, which representation is better as we vary the average number of duplicates in a view. In addition, we would like to quantify the overhead of the CNT representation for views with no duplicates.

Another issue we wish to investigate is the strategy for delta installation. As discussed earlier in this section, delta installation becomes much simpler if we know that the view will not contain duplicates. Let KEYINSTALL denote the method of installing deltas that exploits a lack of duplicates (i.e., a known key), and let GENINSTALL denote the general method that does not. It is easier for the Warehouse Maintainer component to support only GENINSTALL because it works for all views with or without keys. However, warehouse views frequently do have keys. For instance, dimension tables and fact tables, which are modeled as base views, usually have keys. Summary tables often perform `GROUP-BY` operations, and the `GROUP-BY` attributes become the keys of the summary tables. If KEYINSTALL consistently outperforms GENINSTALL for these common cases, the Warehouse Maintainer should support KEYINSTALL as well.

Finally, we also need to evaluate different implementations of GENINSTALL under the CNT representation. As discussed earlier, GENINSTALL under the CNT representation can be implemented with a cursor loop, with two SQL statements, or with one SQL statement and a trigger. (Under the DUP representation, GENINSTALL must be implemented with a cursor loop.) With a cursor loop or a row-level trigger, we have better control over the execution of the installation procedure, so we can optimize it by hand according to our knowledge of the warehouse workload. On the other hand, the SQL statements are optimized by the DBMS, armed with a more sophisticated performance model and statistics. Although traditional DBMS optimizers were not designed originally for data warehousing, modern optimizers have added considerable support for warehouse-type data and queries [CD97]. It is interesting to determine whether the SQL-based delta installation procedure can be optimized adequately by the DBMS we are using. In Section 4.1, we present answers to all of the questions discussed above based on the experiments we have conducted.

3.2 Maintaining Aggregate Views

Given deltas for base views, the Warehouse Maintainer needs to modify the derived views so that they remain consistent with the base views. A simple approach is to recompute all of the derived views from the new contents of the base views, as many existing warehousing systems do. WHIPS, on the other hand, maintains the derived views incrementally for efficiency. The Warehouse Maintainer first computes the deltas for the derived views using a predefined set of queries called *maintenance expressions*, and then it installs these deltas into the derived views. The maintenance expressions of views defined using SQL SFW statements (without subqueries) are well studied, e.g., [GL95], and we do not discuss them here. For views defined using SQL SFWG statements (i.e., views with `GROUP-BY` and aggregation), we will introduce and contrast four different maintenance algorithms through a comprehensive example.

orderID	*partID*	*qty*	*cost*
1	a	1	20
1	b	2	250
2	a	1	20
3	c	1	500

Figure 6: V_1.

partID	*revenue*	*tuplecnt*
a	40	2
b	500	1
c	500	1

Figure 7: *Parts*.

orderID	*partID*	*qty*	*cost*
1	a	2	20
4	c	1	500
4	d	1	30

Figure 8: $\triangle V_1$.

orderID	*partID*	*qty*	*cost*
1	a	1	20
1	b	2	250

Figure 9: $\bigtriangledown V_1$.

In this example, let us suppose that view V_1 contains the tuples shown in Figure 6. A view *Parts* is defined over V_1 to group the V_1 tuples by *partID*. The *revenue* of each part stored in *Parts* is computed from V_1 by summing the products of *qty* and *cost* for each order for that particular part. *Parts* also records in a *tuplecnt* attribute the number of V_1 tuples that are used to derive each *Parts* tuple. The SQL definition of *Parts* is as follows:

```
CREATE VIEW Parts AS
SELECT partID, SUM(qty*price) AS revenue, COUNT(*) AS tuplecnt
FROM V1 GROUP BY partID
```

The tuples in *Parts* are shown in Figure 7. We use the DUP representation for *Parts* since *Parts* has a key (*partID*) and hence contains no duplicates. Note that the *tuplecnt* attribute differs from the *dupcnt* attribute used under the CNT representation since *tuplecnt* does not reflect the number of duplicates in *Parts*. Nevertheless, like *dupcnt*, *tuplecnt* helps incremental view maintenance by recording the number of base view tuples that contribute to each derived view tuple: The *tuplecnt* attribute is used to determine when a *Parts* tuple t should be deleted because all of the V_1 tuples that derive t have been deleted from V_1. In fact, had *tuplecnt* not been included in *Parts*'s definition, WHIPS would automatically modify the view definition to include *tuplecnt* so that *Parts* could be maintained incrementally.

Now suppose that the tuples shown in Figure 8 are to be inserted into V_1, and the ones shown in Figure 9 are to be deleted. (Note that tuples $\langle 1, a, 1, 20\rangle$ in $\bigtriangledown V_1$ and $\langle 1, a, 2, 20\rangle$ in $\triangle V_1$ together represent an update in which the *qty* of a parts purchased in the first order (*orderID* $= 1$) is increased from 1 to 2.) Next we illustrate how we can maintain *Parts* given the deltas $\bigtriangledown V_1$ and $\triangle V_1$, using four different algorithms.

3.2.1 Full Recomputation

Full recomputation (FULLRECOMP) is conceptually simple and easy to implement. First, we install the base view deltas $\bigtriangledown V_1$ and $\triangle V_1$ into V_1. Then, we delete the entire old contents of *Parts* and compute its new contents from V_1.

3.2.2 Summary-Delta With Cursor-Based Installation

The original *summary-delta* algorithm (SDCURSOR for short) for incremental maintenance of aggregate views [MQM97] has a *compute phase* and a cursor-based *install phase*. In the compute phase, the net effect of $\triangle V_1$ and $\bigtriangledown V_1$ on *Parts* is captured in a *summary-delta* table, denoted $Parts_{\mathrm{SD}}$ and computed as follows:

```
SELECT partID, SUM(revenue) AS revenue, SUM(tuplecnt) AS tuplecnt
FROM ((SELECT partID, SUM(qty*price) AS revenue, COUNT(*) AS tuplecnt
        FROM △V1 GROUP BY partID) UNION ALL
       (SELECT partID, -SUM(qty*price) AS revenue, -COUNT(*) AS tuplecnt
        FROM ▽V1 GROUP BY partID))
GROUP BY partID
```

The summary-delta applies the GROUP-BY and aggregation operations specified in the definition of *Parts* to $\triangle V_1$ and $\bigtriangledown V_1$ and combines the results. Note that the aggregate values computed from $\bigtriangledown V_1$ are negated to reflect the effects of deletions on the SUM and COUNT functions. Given the $\triangle V_1$ and $\bigtriangledown V_1$ shown in Figures 8 and 9, the summary-delta $Parts_{\text{SD}}$ is shown in Figure 10.

In the install phase, SDCURSOR instantiates a cursor to loop over the tuples in the summary-delta $Parts_{\text{SD}}$. For each $Parts_{\text{SD}}$ tuple, SDCURSOR applies the appropriate change to *Parts*. For instance, tuple $\langle a, 20, 0\rangle$ affects *Parts* by incrementing the a tuple's *revenue* by 20 and *tuplecnt* by 0. This change reflects the effect of updating the *qty* of a parts in the first order (see $\bigtriangledown V_1$ and $\triangle V_1$). The *tuplecnt* is unchanged because the update does not change the number of V_1 tuples that derive the a tuple in *Parts*. Tuple $\langle b, -500, -1\rangle$ in $Parts_{\text{SD}}$ affects *Parts* by decrementing the b tuple's *revenue* by 500 and *tuplecnt* by 1, which reflects the effect of deleting $\langle 1, b, 2, 250\rangle$ from V_1 (see $\bigtriangledown V_1$). Moreover, the b tuple is then deleted from *Parts*, since its *tuplecnt* becomes zero after it is decremented. Tuple $\langle c, 500, 1\rangle$ increments the c tuple's *revenue* by 500 and *tuplecnt* by 1, reflecting the effect of inserting $\langle 4, c, 1, 500\rangle$ into V_1 (see $\triangle V_1$). Finally, tuple $\langle d, 30, 1\rangle$ results in an insertion, since there is no existing tuple with the same *partID*.

3.2.3 Summary-Delta With Batch Installation

The summary-delta algorithm with batch installation (SDBATCH) is a variation we propose in this paper on the original summary-delta algorithm from [MQM97] described above. The idea is to do more processing in the compute phase in order to speed up the install phase, since views must be locked during installation. In the compute phase of SDBATCH, we first compute the summary-delta $Parts_{\text{SD}}$ as before. From $Parts_{\text{SD}}$, we then compute the deletions $\bigtriangledown Parts$ and insertions $\triangle Parts$ to be applied to *Parts*. $\bigtriangledown Parts$ contains all the *Parts* tuples that are affected by $Parts_{\text{SD}}$, computed as follows:

```
SELECT * FROM Parts WHERE partID IN (SELECT partID FROM Parts_SD)
```

$\triangle Parts$ is the result of applying $Parts_{\text{SD}}$ to $\bigtriangledown Parts$, computed as follows:

```
SELECT partID, SUM(revenue) AS revenue, SUM(tuplecnt) AS tuplecnt
FROM ((SELECT * FROM ▽Parts) UNION ALL (SELECT * FROM Parts_SD))
GROUP BY partID HAVING SUM(tuplecnt) > 0
```

Notice that we filter out those groups with *tuplecnt* less than one because they no longer contain any tuples after $Parts_{\text{SD}}$ is applied. Given the $Parts_{\text{SD}}$ shown in Figure 10, the resulting $\bigtriangledown Parts$ and $\triangle Parts$ are shown in Figures 11 and 12.

In the install phase of SDBATCH, we first apply $\bigtriangledown Parts$, and then $\triangle Parts$, to *Parts*. Because of the way we compute $\bigtriangledown Parts$ in the compute phase, every $\bigtriangledown Parts$ tuple always results in a true deletion from *Parts*, instead of an update that decrements the *revenue* and *tuplecnt* attributes of an existing *Parts* tuple. Since *Parts* is an aggregate view and hence contains no duplicates, the entire $\bigtriangledown Parts$ can be applied in batch using KEYINSTALL with a simple DELETE statement (Section 3.1.2). Once $\bigtriangledown Parts$ has been

partID	*revenue*	*tuplecnt*
a	20	0
b	-500	-1
c	500	1
d	30	1

Figure 10: $Parts_{\mathrm{SD}}$.

partID	*revenue*	*tuplecnt*
a	20	1
b	500	1
c	500	1

Figure 11: $\bigtriangledown Parts$.

partID	*revenue*	*tuplecnt*
a	40	1
c	1000	2
d	30	1

Figure 12: $\triangle Parts$.

applied to $Parts$, every $\triangle Parts$ tuple always results in a true insertion into $Parts$, instead of an update that increments the *revenue* and *tuplecnt* attributes of an existing tuple. Therefore, $\triangle Parts$ can be applied in batch using KEYINSTALL with a simple `INSERT` statement.

3.2.4 Summary-Delta With Overwrite Installation

Both SDCURSOR and SDBATCH update $Parts$ in place, which requires identifying the $Parts$ tuples affected by the $Parts_{\mathrm{SD}}$ tuples. To avoid this potentially expensive operation, we introduce a new summary-delta algorithm with overwrite installation (SDOVERWRITE). SDOVERWRITE completely replaces the old contents of $Parts$ with the new contents, just like FULLRECOMP. However, SDOVERWRITE differs from FULLRECOMP in that SDOVERWRITE does not recompute $Parts$ from scratch; instead, it uses the summary-delta $Parts_{\mathrm{SD}}$ and the old contents of $Parts$ to compute the new $Parts$. The SQL statement used here is similar to the one used to compute $\triangle Parts$ in SDBATCH:

```
SELECT partID, SUM(revenue) AS revenue, SUM(tuplecnt) AS tuplecnt
FROM ((SELECT * FROM Parts) UNION ALL (SELECT * FROM PartsSD))
GROUP BY partID HAVING SUM(tuplecnt) > 0
```

One technicality remains: there is no easy way to replace the contents of $Parts$ with the results of the above `SELECT` statement since the statement itself references $Parts$. To avoid unnecessary copying, we store the results of the above query in another table, and then designate that table as the new $Parts$. Therefore, compared to the other three algorithms, SDOVERWRITE requires additional space roughly the size of $Parts$.

3.2.5 Discussion

Although the example in this section only shows how the various algorithms can be used to maintain a specific aggregate view, it is not hard to extend the ideas to handle views with arbitrary combinations of `SUM`, `COUNT`, and `AVG` aggregate functions. For views with `MAX` or `MIN`, SDCURSOR, SDBATCH, and SDOVERWRITE are not applicable in general, since we cannot perform true incremental maintenance when a base tuple providing the `MAX` or `MIN` value for its group is deleted.

To summarize, we have presented four algorithms for maintaining SFWG views. FULLRECOMP recomputes the entire view from scratch. Intuitively, if the base data is large, FULLRECOMP can be very expensive. SDCURSOR, SDBATCH, and SDOVERWRITE avoid recomputation by applying only the incremental changes captured in a summary-delta table. These three algorithms differ in the ways they apply the summary-delta to the view.

In WHIPS, we are interested in knowing which one of the four algorithms is best in different settings. We also wish to compare how long the different algorithms must lock the view for update. This measure is important because once a view is locked for update, it generally becomes inaccessible for OLAP queries. FULLRECOMP needs to lock while it is regenerating the view. SDCURSOR and SDBATCH only need to lock the view during their install phases. SDOVERWRITE does not lock the view at all, because it computes the new contents of the view in a separate table. In fact, with an additional table, we can also eliminate the locking time of FULLRECOMP, SDCURSOR, SDBATCH, or any maintenance algorithm in general, since we can always work on a copy of the view while leaving the original accessible to queries. In this case, it is useful to know how much locking time we are saving in order to justify the additional storage cost. In Section 4.2, we experimentally compare the performance and locking time of the four algorithms.

So far, we have focused on how WHIPS maintains a single view. In practice, WHIPS maintains a set of views organized in a VDAG (Section 2). When a view V is modified, other views defined over V need to be modified as well. In order to maintain the other views incrementally, we must be able to capture the incremental changes made to V. Among our four algorithms for aggregate view maintenance, only SDBATCH explicitly computes the incremental changes to V as delta tables $\bigtriangledown V$ and $\triangle V$. Although the summary-delta V_{SD} in a way also captures the incremental changes to V, using V_{SD} to maintain a higher-level view is much harder than using $\bigtriangledown V$ and $\triangle V$, especially if the higher-level view is not an aggregate view. For this reason, we might prefer SDBATCH as long as it is not significantly slower than the other algorithms.

4 Experiments

Section 3 introduced several areas in which we must make critical decisions on warehouse view representation and maintenance, and provided some qualitative comparisons of the alternatives. For each decision area, we now quantitatively compare the performance of various alternatives through experiments.

The base views used in the experiments are from TPC-D [TPC96], often including fact tables *Order* and *Lineitem*, which we call O and L for short. The derived views vary from one experiment to the next. Contents of the base views and their deltas are generated by the standard `dbgen` program supplied with the TPC-D benchmark. A TPC-D scale factor of 1.0 means that the entire warehouse is about 1GB in size, with L and O together taking up about 900MB. The commercial DBMS we use is running on a dedicated Windows NT machine with a Pentium II processor.[1] The database buffer size is set at 6.4MB. We have chosen a small database buffer in combination with relatively small TPC-D scale factors in order to limit the duration and the storage requirement of repeated experiments. To study scalability issues we have explored wide ranges of buffer sizes and scale factors for some of our experiments, and found the performance trends to be consistent with the results presented. An example will be shown in Section 4.2.1.

Recall that the WHIPS Warehouse Maintainer sends a sequence of DML statements to the DBMS to query and update views in the data warehouse. In the experiments, we measure the wall-clock time required for the DBMS to run these DML commands, which represents the bulk of the time spent maintaining the warehouse. We have specifically avoided hand-tuning the DBMS optimizer for our experiments because improvements are very sensitive to the workload and to the expertise of the person doing the tuning. We believe it is more instructive to study performance with an "out of the box" DBMS (system parameters at their default values), under the assumption that all of the results we present could probably be improved somewhat in some cases by careful tuning.

4.1 View Representation and Delta Installation

In Section 3.1.4 we identified three decisions that need to be made regarding view representation and delta installation: (1) whether to use DUP or CNT as the view representation; (2) whether to use GENINSTALL or KEYINSTALL to install deltas when the view has a key; (3) whether to use a cursor loop, two SQL

[1] We are not permitted to name the commercial DBMS we are using (nor the second one we have used to corroborate some of our results), but both are state-of-the-art products from major relational DBMS vendors.

statements, or one SQL statement and a trigger to implement GENINSTALL under the CNT representation. In this section, we present performance results that help us make the best decisions for all three areas. The results are presented in reverse order, because we need to choose the best GENINSTALL implementation for CNT (decision area 3) before we can compare CNT with DUP (decision area 1).

4.1.1 GENINSTALL Under CNT: Cursor vs. SQL

In the first experiment, we compare the time to install deltas for base view L^{CNT} (view L using the CNT representation) using a cursor-based implementation versus a SQL-based implementation for GENINSTALL. It would be interesting to compare the performance of a trigger-based implementation as well, but unfortunately the DBMS we are using does not allow a row-level trigger to modify the table whose update caused the trigger to fire.

For this experiment, a TPC-D scale factor of 0.1 is used. Normally, L has a key $\{orderkey, linenumber\}$ and therefore contains no duplicates. We artificially introduce duplicates into L so that on average each L tuple has two other duplicates (i.e., L has a *multiplicity* of 3). We also vary the update ratio of L from 1% to 10%. An update ratio of $k\%$ implies $|\bigtriangledown L| = |\triangle L| = (k/100)\cdot|L|$, i.e., $(k/100)\cdot|L|$ tuples are deleted from L and $(k/100)\cdot|L|$ tuples are inserted. Finally, we assume L^{CNT} has an index on $\{orderkey, linenumber\}$, since $\{orderkey, linenumber\}$ functionally determine all other 13 attributes of L. In general, if there are no nontrivial functional dependencies to exploit, we would have to equate all 15 attributes of L instead of only $orderkey$ and $linenumber$ when matching L tuples with $\bigtriangledown L$ and $\triangle L$ tuples. We performed experiments where no functional dependencies were exploited, and our results (not shown here) reveal that in this case all implementations of GENINSTALL become slower by almost an order of magnitude. Nevertheless, the overall performance trends remain identical to the results presented in this section.

Figure 13 plots the time it takes for the two GENINSTALL implementations to process $\bigtriangledown L$ and $\triangle L$. Recall from Section 3.1 that cursor-based GENINSTALL processes a delta table one tuple at a time, similar to a nested-loop join between the delta and the view with the delta being the outer table. As a result, the plots of cursor-based GENINSTALL for $\bigtriangledown L$ and $\triangle L$ are linear in the size of $\bigtriangledown L$ and $\triangle L$ respectively.

SQL-based GENINSTALL is optimized by the DBMS. To ensure that the optimizer has access to the most up-to-date statistics, we explicitly ask the DBMS to gather statistics after L, $\bigtriangledown L$, and $\triangle L$ are populated (and the time spent in gathering statistics is not counted in the delta installation time). If the optimizer were perfect, SQL-based GENINSTALL would never perform any worse than cursor-based GENINSTALL, because the cursor-based plan is but one of the many viable ways to execute SQL-based GENINSTALL. Unfortunately, we see in Figure 13 that SQL-based GENINSTALL is consistently slower than cursor-based GENINSTALL for deletion installation. This phenomenon illustrates the difficulty of optimizing `DELETE` and `UPDATE` statements with correlated subqueries, such as the second `DELETE` statement shown in Section 3.1.2. The way these statements are structured in SQL leads naturally to an execution plan that scans the entire view looking for tuples to delete or update. However, in an incrementally maintained warehouse, deltas are generally much smaller than the view itself, so a better plan is to scan the deltas and delete or update matching tuples in the view, assuming the view is indexed. Evidently, the state-of-the-art commercial DBMS used by WHIPS missed this plan, which explains why the plots for SQL-based GENINSTALL go nowhere near the origin. We expect this behavior may be typical of many commercial DBMS's today, and we have confirmed this expectation by replicating some of our experiments on another major relational DBMS.

On the other hand, the DBMS is more adept at optimizing SQL-based GENINSTALL for insertions, presumably because `INSERT` statements can be optimized similarly to regular `SELECT` queries and more

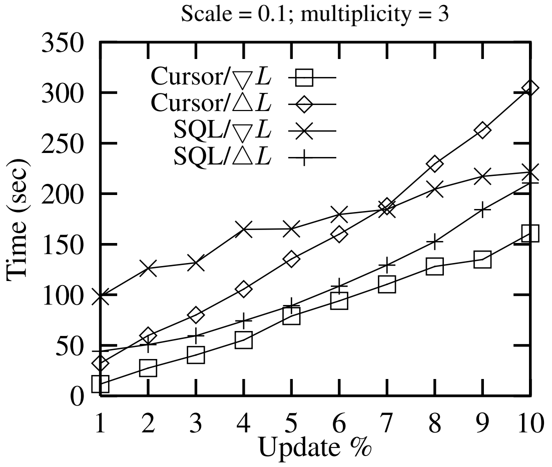

Figure 13: GENINSTALL under CNT.

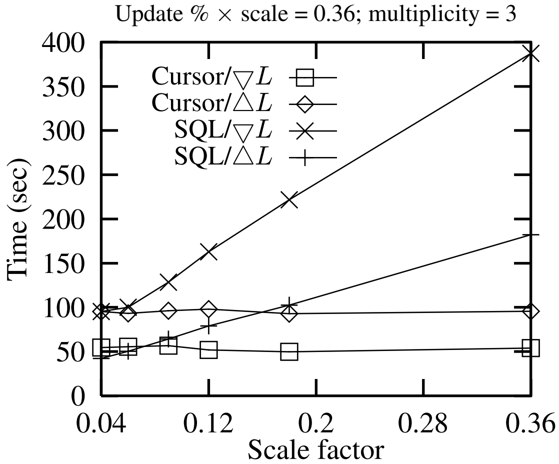

Figure 14: GENINSTALL under CNT.

easily than `DELETE` statements. As shown in Figure 13, SQL-based GENINSTALL is faster than cursor-based GENINSTALL for insertion installation when the update ratio is higher than 1%. To summarize, at TPC-D scale factor 0.1 and update ratio between 1% and 10% on the DBMS we are using, cursor-based GENINSTALL is preferred for deletion installation and SQL-based GENINSTALL is preferred for insertion installation under the CNT representation.

In the next experiment, we investigate how the size of the views might affect this decision. We fix the size of $\bigtriangledown L$ and $\triangle L$ at about 2.6MB each while varying the TPC-D scale factor from 0.04 to 0.36. The update ratio thus varies from 9% to 1%. The results in Figure 14 indicate that the running time of cursor-based GENINSTALL is insensitive to the change in $|L|$, while the running time of SQL-based GENINSTALL grows linearly with $|L|$. Thus, we should use cursor-based GENINSTALL to process both deletions and insertions for large views with low update ratios.

4.1.2 Delta Installation: KEYINSTALL vs. GENINSTALL

In the following experiment, we seek to quantify the performance benefits of using KEYINSTALL instead of GENINSTALL when the view has a key and hence no duplicates. In this case, we are only interested in the DUP representation since DUP should always be used instead of CNT when the view contains no duplicates (Section 3.1.4). Furthermore, under the DUP representation, insertion installation requires one simple `INSERT` statement, which is the same for both KEYINSTALL and GENINSTALL. Therefore, our task reduces to comparing KEYINSTALL and GENINSTALL for deletion installation under the DUP representation.

In Figure 15, we plot the time it takes for KEYINSTALL and GENINSTALL to install $\bigtriangledown L$ in base view L, which contains no duplicates and has an index on its key attributes. We fix the TPC-D scale factor at 0.1 and vary the update ratio from 1% to 10%. At first glance, the results may seem counterintuitive: KEYINSTALL is slower than GENINSTALL when the update ratio is below 5%. However, recall from Section 3.1.2 that KEYINSTALL uses a simple `DELETE` statement to install $\bigtriangledown L$, while GENINSTALL requires a cursor loop. Clearly, the DBMS has failed again to take advantage of the small $\bigtriangledown L$ and the index on L when optimizing the `DELETE` statement. Given this limitation of the DBMS optimizer, we cannot justify implementing KEYINSTALL in addition to GENINSTALL in the Warehouse Maintainer from a performance perspective.

4.1.3 View Representation: DUP vs. CNT

We now evaluate the performance of DUP and CNT representations in the case where the view may contain duplicates. In Figure 16, we compare the the time it takes to install $\bigtriangledown L$ and $\triangle L$ under DUP and CNT

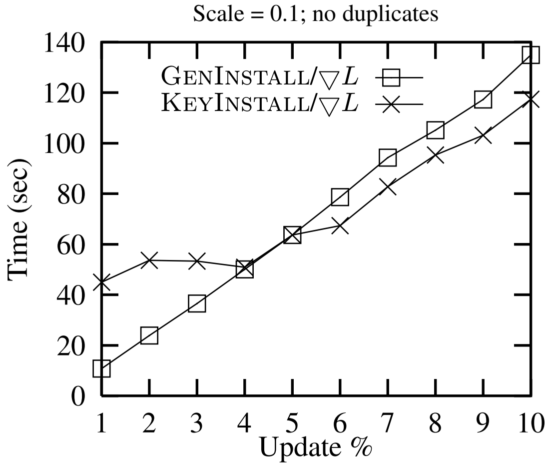

Figure 15: KeyInstall vs. GenInstall under DUP.

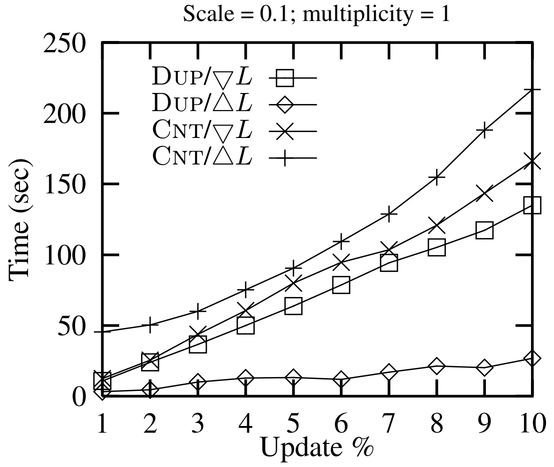

Figure 16: Dup vs. CNT for delta installation.

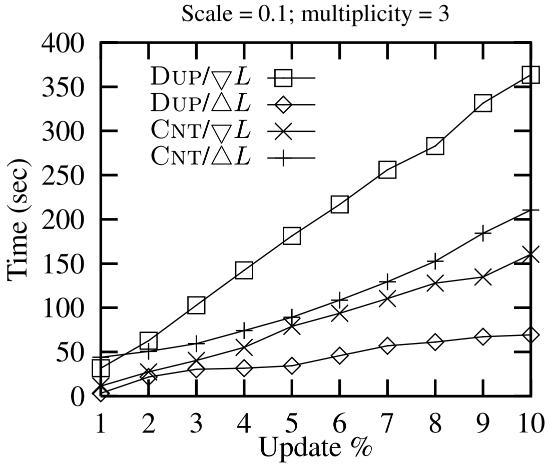

Figure 17: Dup vs. CNT for delta installation.

representations. Again, the TPC-D scale factor is fixed at 0.1 and the update ratio varies from 1% to 10%. The multiplicity of L in this first experiment is close to 1, i.e., L contains almost no duplicates. We create indexes for both L^{DUP} and L^{CNT} on $\{orderkey, linenumber\}$, even though $\{orderkey, linenumber\}$ is not a key for L^{DUP} because of potential duplicates. For the CNT representation, we use cursor-based GenInstall to install $\bigtriangledown L^{\text{CNT}}$ and SQL-based GenInstall to install $\bigtriangleup L^{\text{CNT}}$, as decided in Section 4.1.1 for scale factor 0.1. The results plotted in Figure 16 indicate that delta installation (insertions and deletions combined) under the CNT representation is about twice as expensive as delta installation under the DUP representation.

In the next experiment, we increase the multiplicity of L from 1 to 3, thereby tripling the size of L^{DUP}, $\bigtriangledown L^{\text{DUP}}$, and $\bigtriangleup L^{\text{DUP}}$. On the other hand, the increase in multiplicity has no effect on the size of L^{CNT}, $\bigtriangledown L^{\text{CNT}}$, and $\bigtriangleup L^{\text{CNT}}$. As Figure 17 shows, installing $\bigtriangledown L^{\text{DUP}}$ becomes more than twice as slow as installing $\bigtriangledown L^{\text{CNT}}$, but installing $\bigtriangleup L^{\text{DUP}}$ is still three times faster than installing $\bigtriangleup L^{\text{CNT}}$. Overall, delta installation under the DUP representation is slightly slower than under the CNT representation. By comparing Figures 16 and 17, we also see that the delta installation time under the DUP representation increases proportionately with multiplicity, while the time under CNT remains the same.

To study the effect of view size on delta installation time, we conduct another experiment in which we fix the size of the deltas and vary the TPC-D scale factor from 0.04 to 0.36. Multiplicity of L is set at 3. For

	DUP representation	CNT representation
$\bigtriangledown$ (deletion)	$1.94\,ms$ / tuple	$\frac{2.49\,ms}{\text{multiplicity}}$ / tuple
$\bigtriangleup$ (insertion)	$0.57\,ms$ / tuple	$\frac{4.48\,ms}{\text{multiplicity}}$ / tuple

Table 1: Per-tuple delta installation costs.

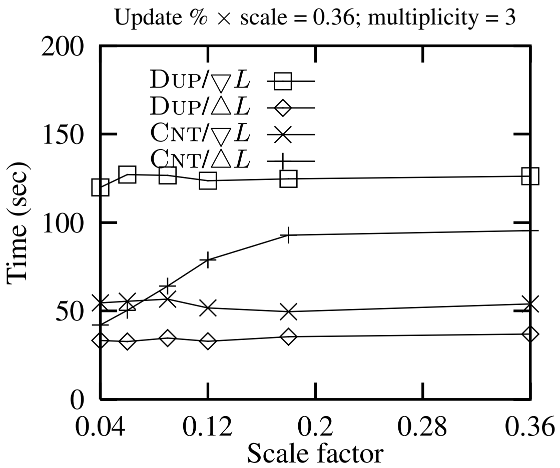

Figure 18: DUP vs. CNT for delta installation.

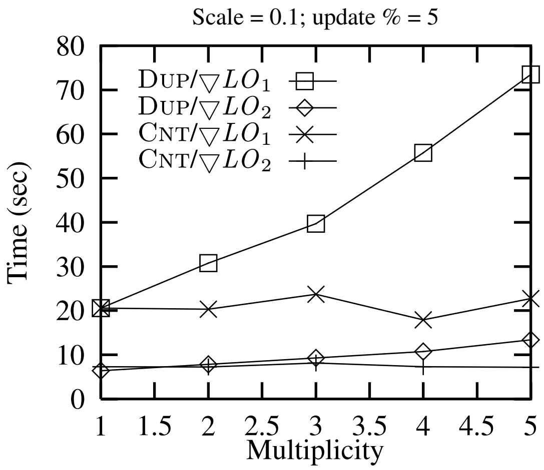

Figure 19: DUP vs. CNT for delta computation.

the CNT representation, we use the best GENINSTALL implementation (either cursor-based or SQL-based, depending on the scale factor; recall Section 4.1.1). The results are shown in Figure 18. Notice that all four plots become nearly flat at large scale factors: when the view is sufficiently large, the cost of installing a delta tuple into the view approaches a constant for each delta type and each view representation. Thus, measured once, these per-tuple costs can be used to estimate the delta installation time in a large data warehouse. Table 1 shows the per-tuple installation costs measured under our experimental settings.

All experiments so far have focused on delta installation. In the next experiment, we compare the time to compute deltas for derived views under DUP and CNT representations. We define one derived view LO_1 as an equijoin between L and O, with a total of 24 attributes. Another derived view LO_2 is defined as the same equijoin followed by a projection which leaves only two attributes. In Figure 19, we plot the the time it takes to compute $\bigtriangledown LO_1$ and $\bigtriangledown LO_2$ given $\bigtriangledown L$ under both DUP and CNT representations. We choose an update ratio of 5% and increase the multiplicity of L from 1 to 5. When the multiplicity of L is close to 1 (i.e., L contains almost no duplicates), DUP and CNT offer comparable performance for computing derived view deltas. The overhead of the extra *dupcnt* attribute and the extra aggregation required for doing projection under the CNT representation (discussed in Section 3.1.1) turns out to be insignificant in this case. As we increase the multiplicity, computing derived view deltas under DUP becomes progressively slower than CNT because the CNT representation is about m times more compact than DUP, where m is the multiplicity.

To summarize, the CNT representation scales well with increasing multiplicity and carries little overhead for computing derived view deltas. However, in the common case where the average number of duplicates is low (e.g., multiplicity is less than 3), the DUP representation is preferable because it is faster in installing deltas, especially insertions, which tend to be common in data warehousing applications.

4.2 Maintaining Aggregate Views

In Section 3.2 we presented aggregate view maintenance algorithms FULLRECOMP and SDCURSOR, as well as two new variations SDBATCH and SDOVERWRITE. This section compares the performance of the

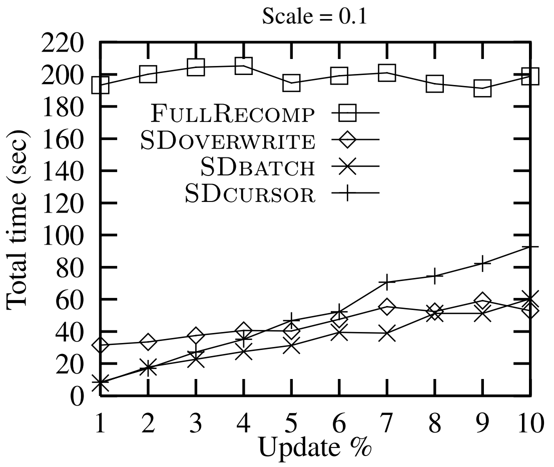

Figure 20: Maintaining V_{large}.

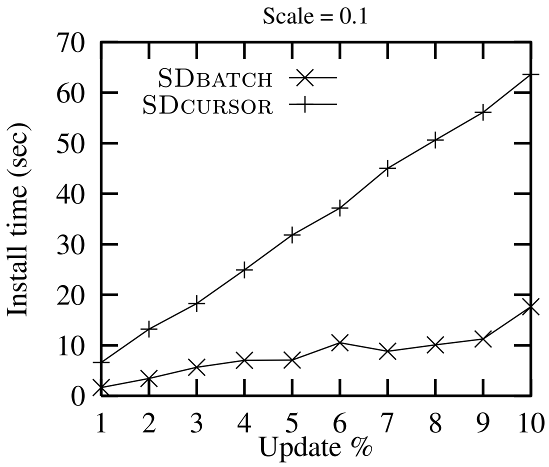

Figure 21: Installing deltas for V_{large}.

four algorithms in terms of the total time required for maintaining aggregate views. We also compare the install phase lengths of SDCURSOR and SDBATCH. Recall from Section 3.2.5 that we want to minimize the length of the install phase in a data warehouse because during this phase the view is locked for update.

We consider two types of aggregate views in our experiments. A *fixed-ratio* aggregate over a base view V groups V into $\alpha \cdot |V|$ groups, where α is a fixed *aggregation ratio*. The size of a fixed-ratio aggregate increases with the size of the base view. On the other hand, a *fixed-size* aggregate over a base view V groups V into a small and fixed number of groups. We will see that the four aggregate maintenance algorithms behave differently for these two types of aggregates.

4.2.1 Maintaining Fixed-Ratio Aggregates

In the first experiment, we use a fixed-ratio aggregate V_{large} which groups the base view L by *orderkey* into approximately $0.25 \cdot |L|$ groups. First, we study the impact of the update ratio on the maintenance time of V_{large} by fixing the TPC-D scale factor at 0.1 and varying the update ratio from 1% to 10%. Figure 20 shows that the update ratio has no effect on FULLRECOMP at all. However, the total maintenance time increases (gradually) with the update ratio for the three summary-delta-based algorithms. Among these algorithms, SDCURSOR is most sensitive to the change in update ratio, and SDOVERWRITE is least sensitive. Figure 20 also shows that FULLRECOMP is far more expensive that the other three algorithms. SDBATCH is the fastest when the update ratio is small, while SDOVERWRITE becomes the fastest when the update ratio is higher than 10%. In Figure 21, we further compare SDCURSOR and SDBATCH in terms of the time of the install phase. By doing more work in its compute phase, SDBATCH has a much shorter install phase than SDCURSOR. This shorter install phase gives our new algorithm SDBATCH an edge in applications where data availability is important.

Figures 22 and 23 show the results of repeating the same experiment with the size of the database buffer set to 240MB instead of 6.4MB (the default in our experiments). Despite the large disparity in buffer size, the performance trends revealed by Figures 22 and 23 are remarkably similar to those revealed by Figures 20 and 21. We have repeated this experiment with a number of different buffer sizes, and found that even though performance is improved by extra memory (though not linearly), the relative performance of the schemes remains the same.

To study the impact of base view size on aggregate maintenance, we fix the size of $\bigtriangledown L$ and $\triangle L$ and vary the scale factor of L from 0.04 to 0.36. Figure 24 shows that the performance of FULLRECOMP

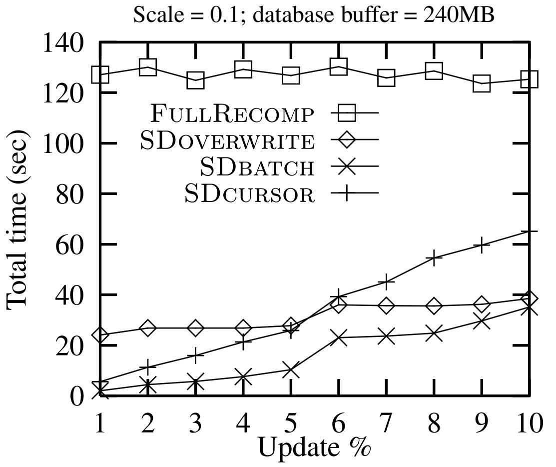

Figure 22: Maintaining V_{large}.

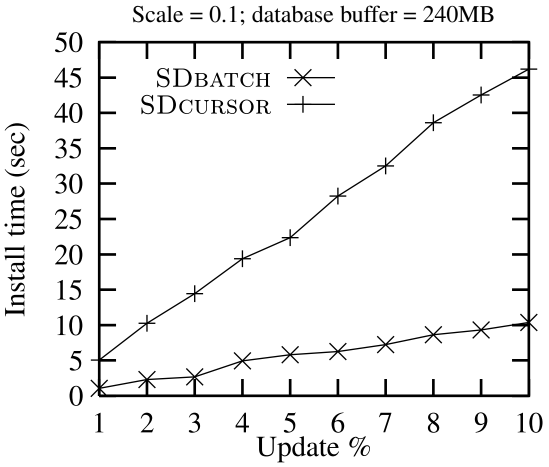

Figure 23: Installing deltas for V_{large}.

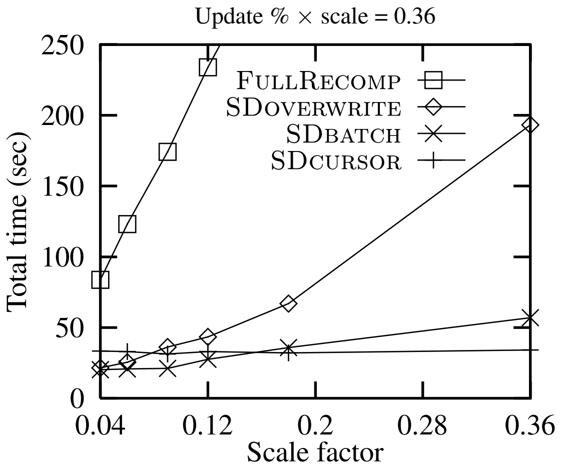

Figure 24: Maintaining V_{large}.

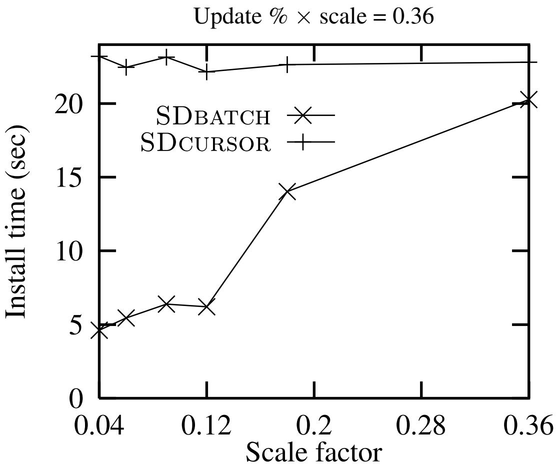

Figure 25: Installing deltas for V_{large}.

deteriorates dramatically as $|L|$ increases, because FullRecomp is recomputing the aggregate over a larger L. Although SDoverwrite and SDbatch are not affected by $|L|$ as much as FullRecomp, their running time still increases because $|V_{large}|$ grows with $|L|$. On the other hand, SDcursor is almost unaffected by the increase in $|L|$. The reason is that SDcursor uses an index on V_{large} to find the matching V_{large} tuples for each tuple in the summary-delta table, and the lookup time of the index remains constant for the range of $|V_{large}|$ in this experiment. Figure 24 also shows that FullRecomp is much slower than the other algorithms. SDbatch is the fastest algorithm for smaller base views, while SDcursor is the fastest for larger base views.

Figure 25 further compares the time of the install phase for SDcursor and SDbatch. The install phase of SDcursor is not affected by the size of the base view, for the same reason discussed above. The install phase of SDbatch takes much less time than that of SDcursor for relatively small base views, but it increases as the base view becomes larger. Once again, the explanation lies with the limitation of the DBMS optimizer. When executing the SQL `DELETE` statement to install $\bigtriangledown V_{large}$, the DBMS fails to make use of the available index on V_{large}; instead, it scans V_{large} looking for tuples to delete, which requires time proportional to $|V_{large}|$, or $0.25 \cdot |L|$.

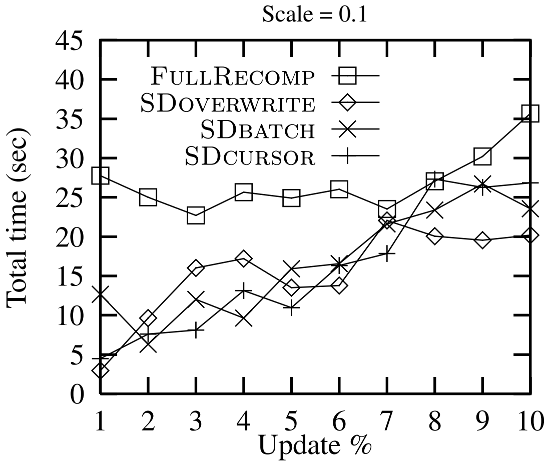

Figure 26: Maintaining V_{small}.

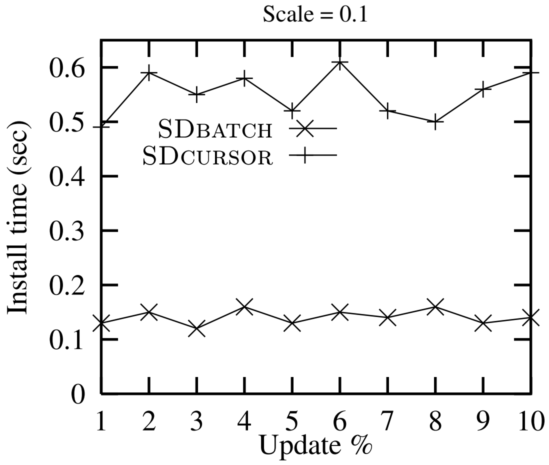

Figure 27: Installing deltas for V_{small}.

4.2.2 Maintaining Fixed-Size Aggregates

We now compare the performance of the four algorithms for a fixed-size aggregate view V_{small}, which groups the base view L by *linenumber* into exactly seven groups, regardless of the size of L. Figure 26 shows the impact of the update ratio on the maintenance of V_{small}. Again, the performance of FULLRECOMP does not change much as the update ratio increases. Plots of the other three algorithms, although somewhat noisy due to inconsistent DBMS behavior on such a small table, show that the total maintenance time increases at about the same rate for all three algorithms. The reason is that the time it takes to compute the summary-delta table (which is used by all three algorithms) increases with the update ratio. On the other hand, the size of the summary-delta is usually fixed for a small fixed-size aggregate such as V_{small}, because any base view delta of reasonable size will likely affect all groups in the aggregate. The type of work after computing the summary-delta varies from one algorithm to another, but for each algorithm, the amount of work depends only on the size of the aggregate and the size of the summary-delta, which are both fixed in this case. Figure 27 further indicates that SDBATCH has a shorter install phase than SDCURSOR, and neither is affected by the update ratio because V_{small} is a fixed-size aggregate. In this case the install phase does not take a significant portion of the total maintenance time, but as we have discussed, the length of the install phase is still an important measure of warehouse availability.

Finally, Figures 28 and 29 plot the total maintenance time and installation time of V_{small} respectively as we fix the size of the base view deltas and vary the size of the base view. We see in Figure 28 that FULLRECOMP becomes dramatically slower as the base view becomes larger. However, the other three algorithms are not affected, again because the size of the aggregate and the size of the summary-delta remain constant despite the increasing base view size. Figure 29 also shows that the install phase of SDCURSOR and SDBATCH is not affected by the increasing base view size. Furthermore, SDBATCH has a faster install phase than SDCURSOR.

4.2.3 Summary

Based on our experiments, we learn that the aggregate maintenance algorithms behave differently on different types of aggregate views. In addition, two other factors—the size of the base view and the size of the base view deltas—also affect the performance of aggregate view maintenance, and different maintenance algorithms react differently to these factors. Table 2 summarizes the sensitivity of each algorithm to various factors using a scale of 0 (insensitive) to 3 (very sensitive). In conclusion, the following guidelines may be

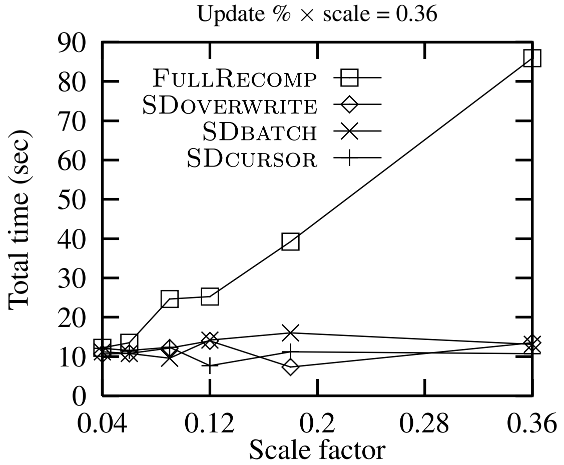

Figure 28: Maintaining V_{small}.

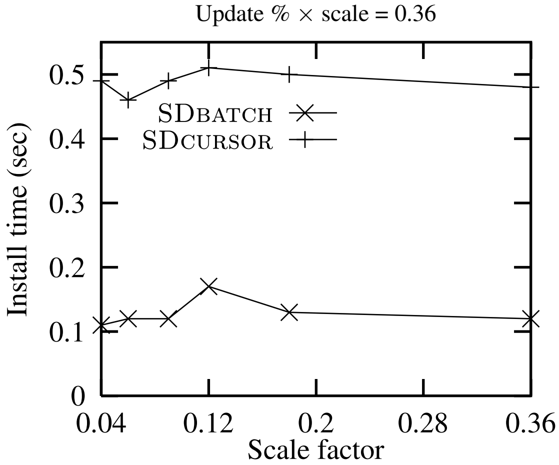

Figure 29: Installing deltas for V_{small}.

	fixed-ratio aggregate		fixed-size aggregate	
	base view delta size	base view size	base view delta size	base view size
FULLRECOMP	0	3	0	3
SDOVERWRITE	1	2	1	0
SDBATCH	2	1	1	0
SDCURSOR	3	0	1	0

Table 2: Sensitivity of aggregate view maintenance algorithms.

used to choose an aggregate view maintenance algorithm for a given scenario:

- Algorithms based on summary-delta tables (SDBATCH, SDCURSOR, and SDOVERWRITE) are much faster than FULLRECOMP, especially for relatively large base views.
- For fixed-ratio aggregates, SDCURSOR is preferred for large base views with small deltas, while SDOVERWRITE is preferred for small base views with large deltas. SDBATCH is a compromise between the two.
- For fixed-size aggregates, all three algorithms based on summary-delta tables perform equally well.
- SDBATCH generally has a shorter install phase than SDCURSOR. However, in the case of fixed-ratio aggregates this advantage of SDBATCH could diminish as the size of the base view increases if the DBMS is unable to optimize `DELETE` statements effectively.

5 Related Work

There has been a significant amount of research devoted to view maintenance; see [GM95] for a survey. However, there has been little coverage of the important details of view maintenance that are the focus of this paper. For instance, [LYGM99] assumes that there is an *Inst* operation for installing deltas into a view, but does not cover the details on how to implement *Inst*. [MQM97] proposes the summary-delta algorithm for aggregate maintenance, but does not consider the various alternatives for applying the summary delta to the aggregate view. In this paper, we have presented three possible implementations of the view maintenance procedure, and evaluated their performance through experiments.

[GMS93] assumes that materialized views use the CNT representation. On the other hand, [GL95] assumes that the DUP representation is used. To our knowledge, our paper is the first to investigate in detail the pros and cons of the two representations for view maintenance, and to present supporting experiments.

We also briefly described the WHIPS data warehousing system. Although significant research has been devoted to both view maintenance and data warehousing, few warehouse prototypes focus on incremental view maintenance the way WHIPS does. [HZ96] presents the *Squirrel* integration mediator, which acts as a data warehouse. [HZ96] focuses on support for virtual attributes in warehouse views, but does not discuss the details of view maintenance in Squirrel. [GHJ96] describes the *Heraclitus* project, in which deltas are treated as first-class citizens in a database programming language. [GHJ96] does not permit duplicates, and it proposes hash-based and sort-merge-based delta installation procedures. In this paper, we also consider the general case where the views may contain duplicates, and we propose cursor-based delta installation procedures to exploit the indexes that typically exist on warehouse views. [CKL$^+$97] presents the *Sword* warehouse prototype, which supports multiple view maintenance policies specifying when to maintain the warehouse views (e.g., immediately versus on-demand). [CKL$^+$97] studies the performance tradeoffs between the different view maintenance policies. The results presented in our paper can be used to optimize the actual implementation of each maintenance policy; thus, our results are complementary to [CKL$^+$97]. [Rou91] describes the *ADMS* prototype, which investigates the advantages of materializing a view using *view-caches* (join indexes) as opposed to view tuples. Commercial DBMS's currently do not allow materialized views to be represented using join indexes. Therefore, a data warehousing system such as WHIPS that is implemented using a commercial DBMS must use actual view tuples. [Vis98] presents *RHODES*, a query optimizer designed for view maintenance, built using the Volcano optimizer generator [GM93]. [Vis98] focuses on optimizing delta computation for SFW views and not on optimizing delta installation. Our results show that delta installation takes a substantial fraction of warehouse maintenance time and therefore warrants the careful study presented in this paper.

As mentioned in Section 1, incremental warehouse maintenance has also found its way into commercial systems, such as the Red Brick database loader [FS96], Oracle materialized views [BDD$^+$98], and DB2 automatic summary tables [BS99]. The results in this paper should be especially useful to the vendors for improving the performance of these systems, since we have conducted all of our experiments on a commercial DBMS as well.

6 Conclusion

This paper has addressed performance issues in incrementally maintained data warehouses, with our own WHIPS prototype serving as a framework for experimenting with a variety of techniques for efficient view maintenance. We identified several critical data representation and algorithmic decisions, and proposed guidelines for making the right decisions in different scenarios, supported by our experimental results. From the results of our experiments, we can see that making the right decision requires considerable analysis and tuning, because the optimal strategy often depends on many factors such as the size of views, the size of deltas, and the average number of duplicates. Ideally, some of the decisions can and should be made by the DBMS optimizer, since it has access to most relevant statistics. However, because of current limitations of DBMS optimizers, many decisions still need to be made by an external agent, such as the Warehouse Maintainer in WHIPS or a human warehouse administrator. As DBMS vendors continue to introduce more data warehousing features, we hope that view maintenance, especially delta installation, will receive an increasing level of support.

Acknowledgements

We are grateful to past WHIPS project members Reza Behforooz, Himanshu Gupta, Joachim Hammer, Janet Wiener, and Yue Zhuge, and to all of our colleagues in the Stanford Database Group who have contributed to WHIPS.

References

[BDD+98] R. G. Bello, K. Dias, A. Downing, J. Feenan, W. D. Norcott, H. Sun, A. Witkowski, and M. Ziauddin. Materialized views in Oracle. In *Proc. of the 1998 Intl. Conf. on Very Large Data Bases*, pages 659–664, New York, New York, August 1998.

[BS99] S. Brobst and D. Sagar. The new, fully loaded, optimizer. *DB2 Magazine*, 4(3):23–29, September 1999.

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

[CKL+97] L. S. Colby, A. Kawaguchi, D. F. Lieuwen, I. S. Mumick, and K. A. Ross. Supporting multiple view maintenance policies. In *Proc. of the 1997 ACM SIGMOD Intl. Conf. on Management of Data*, pages 405–416, Tucson, Arizona, May 1997.

[FS96] P. M. Fernandez and D. A. Schneider. The ins and outs (and everthing in between) of data warehousing. In *Proc. of the 1996 ACM SIGMOD Intl. Conf. on Management of Data*, page 541, Montreal, Canada, June 1996.

[GHJ96] S. Ghandeharizadeh, R. Hull, and D. Jacobs. Heraclitus: Elevating deltas to be first-class citizens in a database programming language. *ACM Trans. on Database Systems*, 21(3):370–426, September 1996.

[GL95] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *Proc. of the 1995 ACM SIGMOD Intl. Conf. on Management of Data*, pages 328–339, San Jose, California, May 1995.

[GM93] G. Graefe and W. J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *Proc. of the 1993 Intl. Conf. on Data Engineering*, pages 209–218, Vienna, Austria, April 1993.

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

[GMS93] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proc. of the 1993 ACM SIGMOD Intl. Conf. on Management of Data*, pages 157–166, Washingtion, D.C., May 1993.

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

[LGM96] W. J. Labio and H. Garcia-Molina. Efficient snapshot differential algorithms for data warehousing. In *Proc. of the 1996 Intl. Conf. on Very Large Data Bases*, pages 63–74, Bombay, India, September 1996.

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

[LYGM99] W. J. Labio, R. Yerneni, and H. Garcia-Molina. Shrinking the warehouse update window. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, pages 383–394, Philadephia, Pennsylvania, June 1999.

[MQM97] I. S. Mumick, D. Quass, and B. S. Mumick. Maintenance of data cubes and summary tables in a warehouse. In *Proc. of the 1997 ACM SIGMOD Intl. Conf. on Management of Data*, pages 100–111, Tucson, Arizona, May 1997.

[Rou91] Nick Roussopoulos. The incremental access method of view cache: Concept, algorithms, and cost analysis. *ACM Trans. on Database Systems*, 16(3):535–563, September 1991.

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

[Vis98] D. Vista. Incremental view maintenance as an optimization problem. In *Proc. of the 1998 Intl. Conf. on Extending Database Technology*, pages 374–388, Valencia, Spain, March 1998.

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

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