

00A000076

# Immediate Propagate Deferred Apply for Incremental Maintenance of Materialized Views

**Kevin S. Beyer**\*†  
University of Wisconsin – Madison  
beyer@cs.wisc.edu

**Roberta Cochrane**  
IBM Almaden Research Center  
bobbiec@almaden.ibm.com

**Hamid Pirahesh**  
IBM Almaden Research Center  
pirahesh@almaden.ibm.com

**Richard Sidle**  
IBM Almaden Research Center  
rsidle@almaden.ibm.com

**Jayavel Shanmugasundaram**\*  
University of Wisconsin – Madison  
jai@cs.wisc.edu

**C. Mohan**  
IBM Almaden Research Center  
mohan@almaden.ibm.com

**Kenneth Salem**\*  
University of Waterloo  
kmsalem@db.uwaterloo.ca

## Abstract

Materialized views dramatically improve the processing time of decision support queries. Incremental maintenance of materialized views reduces the time required to keep the views up to date. In this paper, we provide a brief overview of several incremental maintenance policies and describe how multiple views with differing policies can be combined to provide a consistent answer to a query.

We focus on one particular maintenance policy called Immediate Propagate Deferred Apply (IPDA) that reduces the locking contention for views that are refreshed immediately while keeping the storage requirements low. The propagation process of IPDA is relatively straight forward, but the apply process presents several challenges. Is the order of view updates important, even in the presence of integrity constraints? Are any updates to the view lost? Are the effects of transactions atomically applied to a view? Are transaction dependencies preserved? Are the views consistent with each other? We investigate these questions in the context of our prototype implementation of IPDA in DB2.

---

\*While visiting IBM Almaden Research Center

†Contact author: Computer Sciences Dept., 1210 W. Dayton, UW-Madison, Madison, WI 53706-1685, Phone: 608-262-6629



# 1 Introduction

The battle over materialized views in commercial database systems is heating up. Researchers have fueled this battle with many recent contributions [12, 2, 7, 14, 6, 5, 9, 3, 8, 10]. The algorithms in many of these papers are described at a high level and in abstract terms. They do not, however, address many of the challenges faced while fully implementing the algorithms.

Support for materialized views is starting to appear in DBMSs. For example, IBM DB2 and Oracle [1] rewrite queries to make use of materialized views (called *routing*), and they can incrementally maintain certain types of views upon changes to base data. IBM is currently exploring additional support: additional types of queries are incrementally maintainable, new types of queries can be matched during routing, and alternative deferred maintenance policies are being explored.

## 1.1 View Maintenance Policies

This paper describes a new deferred maintenance policy called Immediate Propagate Deferred Apply (*IPDA*). We describe IPDA in relation to three other maintenance policies: *snapshot*, *immediate*, and *DPDA*. Snapshot views are recomputed from scratch at the user's request. Snapshot views frequently do not reflect the current state of the database (i.e., they are out-of-date), and can be expensive to recompute because the entire view is reevaluated.

Immediate views are at the other end of the spectrum; the view is *incrementally* maintained by every transaction that affects it. So immediate views always reflect the current state of the database. The block diagram of immediate maintenance is shown in Figure 1a. As a transaction performs a DML statement, the changes to a base tables are also accumulated in a corresponding delta table. The *propagate* step uses the maintenance expressions defined in [3] to compute the changes to the view, called the *view delta*. The view delta is pipelined into the *apply* step, which immediately applies the changes to the materialized view.

IPDA and DPDA are intermediate policies between the snapshot and immediate refresh policies. Like immediate refresh, IPDA immediately propagates the changes from the base tables to compute the view delta, but instead of applying the changes directly to the view, the changes are stored in a *view delta table* to be applied later.

Compare the IPDA process depicted in Figure 1b with the immediate refresh process; the hollow arrow denotes a break in time, and the diamonds show where additional work is performed. The propagate process for IPDA is identical to immediate refresh, except that IPDA has to correctly store the result in the delta table. When the apply process is performed, the changes from all the transactions that committed changes by the refresh time<sup>1</sup> are applied to the view at the same time, so the IPDA apply process performs some additional work to find these records<sup>2</sup> before the immediate refresh apply step is run. The immediate propagate and apply steps have been aptly described in previous work, so we do not describe the details

<sup>1</sup>The refresh time can be before the time that the apply process is performed.

<sup>2</sup>The records may need some additional transformations before they are ready to be applied.

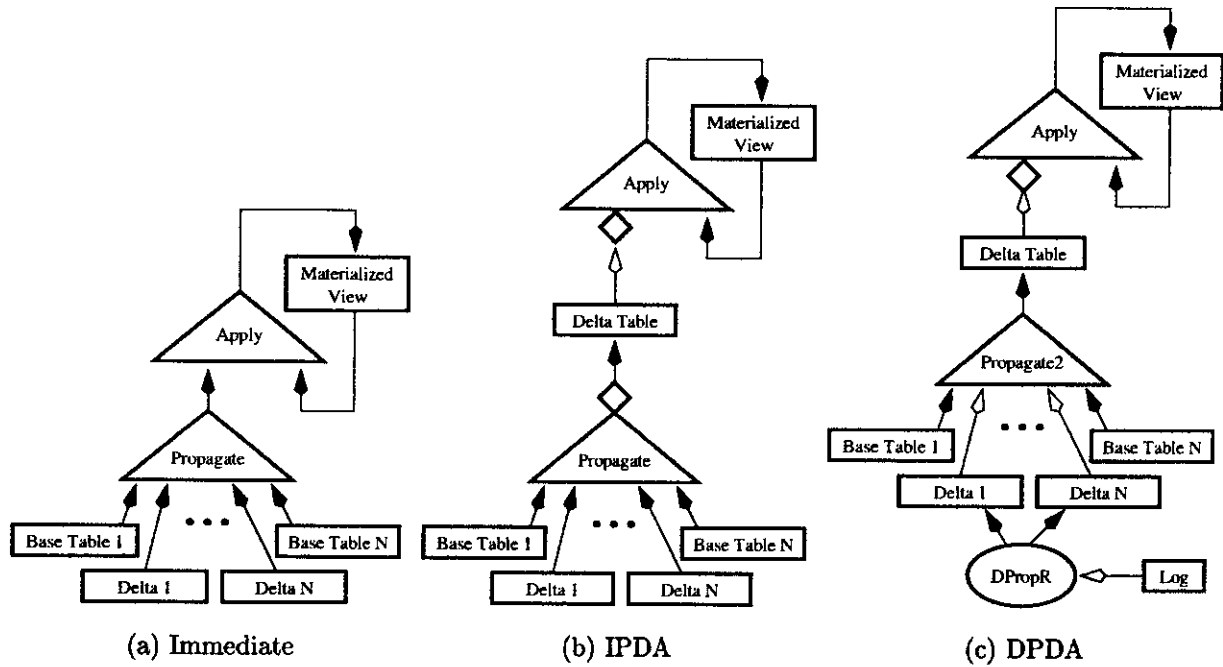


Figure 1: Incremental Refresh Policies

here. However, the majority of this paper describes the tricky details of storing and accessing the view delta in a table.

Figure 1c shows the Deferred Propagate Deferred Apply (*DPDA*) refresh policy. With *DPDA*, changes to base tables are captured by the system and placed in the log. At a later point, the log is examined and the changes to the base tables are placed in their corresponding delta tables (in the case of DB2, by a product called *DPropR*). Alternatively, the base deltas can be created by triggers on the base tables. After the base deltas are created, the propagate process computes the view delta. Unlike immediate refresh and *IPDA* however, the *DPDA* propagate process is complicated by the fact that multiple transactions are propagated at the same time, the propagation occurs asynchronously with changes to the base tables, and the process can be split into multiple transactions to reduce locking contention with update transactions. The *rolling propagation* algorithm addresses these issues; we refer the reader to our companion paper for the full details [11]. Once the *DPDA* propagate process computes the delta table, the apply process is the same as *IPDA*.

*IPDA* and *DPDA* are not new maintenance policies in the research literature. They were described in [3] as “Deferred maintenance with differential tables for views”, and “Deferred maintenance with differential tables and base logs” respectively. However, the paper does not describe the policies in detail, and many challenging problems lie beneath the overview presented.

## 1.2 Motivation for IPDA

The primary goal of IPDA (and DPDA) is to reduce maintenance conflicts caused by immediate refresh, and still perform incremental maintenance. Consider two transactions,  $x_1$  and  $x_2$ , and the following materialized view:

```
StateCount =
  SELECT  c.state, count(*) as count
  FROM    Sales s, Customer c
  WHERE   s.custId = c.custId
  GROUP BY c.state
```

If  $x_1$  adds a record to `Sales` for a customer in Wisconsin but does not immediately commit (perhaps because it has additional work to do), and subsequently  $x_2$  adds a record for some other customer in Wisconsin,  $x_2$  is forced to wait for  $x_1$  to commit because they conflict on the Wisconsin record in `StateCount`.

Since most materialized views contain aggregates, the chance of a conflict using immediate refresh is quite high. In particular, if any of the materialized views perform a CUBE operation, then all transactions that update any table that contributes to the CUBE view are guaranteed to conflict on the grand total record.

IPDA solves this problem by separating the update of the views from the update transactions. The update transactions simply append their view changes to the view delta table, and a (single) separate transaction applies the changes from all the update transactions to the view.

Another important goal of IPDA is support for point-in-time refresh. IPDA allows a view to be rolled forward from its current time to a specified time, called the *refresh time*, that is before the current time. For example, if the `StateCount` view is current as of Friday, Dec. 6, then on Sunday, Dec. 15 we can roll the view up to Friday, Dec. 13. Separating the refresh time from the time at which the apply is performed enables two key features. First, the apply process can be performed during off-peak hours. Second, multiple views can be refreshed to the same point in time, which means they can be combined together to provide a consistent answer to a single query.

A third goal for IPDA is to enable multiple uses of the same delta table. The delta table can be used in several ways. One way is to allow multiple views to have different update schedules. For example, we could have `DailyStateCount` and `MonthlyStateCount` that are current as of the end of the previous day and month, respectively. Another way is to allow both a system maintained view and externally maintained view, which enables applications that are layered on top of the DBMS, like SBase and SAS, to incrementally maintain their own materialized views. The final way that we can make use of the delta table is when materialized views are defined in terms of other materialized views. For example, if some view  $V_2$  is defined in terms of some other view  $V_1$ , then the delta table for  $V_1$  can be used by DPDA to incrementally maintain  $V_2$ .

## 1.3 Contributions

This paper makes the following contributions:

- How to store and access a view delta created by IPDA to reduce the locking conflicts that arise when a view is maintained immediately, and to ensure the IPDA view maintains transactional consistency. The implementation should require as few changes to the system as possible, and make use of as much of the system infrastructure as possible. Therefore, in our prototype implementation, the view delta is stored in a normal database table, not a special data structure, and the propagate and apply processes are implemented by semantic rewrites in the query compiler using SQL statements.
- How to roll a view forward to a particular point in time, which is useful for applications that need to refresh a view to particular time, but want to perform the maintenance at a later time (e.g., during off-peak hours). Another important feature of point-in-time refresh is it allows multiple views to reflect the same state of the database, even when the views are refreshed independently.
- How to compute the net-effect of a view delta to avoid replaying history, and still prevent integrity constraint violations on the materialized view. The net-effect can also be used to condense the information stored in the view delta.
- How the *valid-time* of views can be used to compute a consistent, but not necessarily current, answer to a query when multiple materialized views are used.

## 1.4 Paper Outline

The remainder of the paper is organized as follows: Section 2 presents the definitions and notation used in the paper. The properties that an IPDA implementation should possess are discussed in Section 3. We describe how to find the records to apply in Section 4, and Section 5 discusses how these records should be applied. Section 6 describes how multiple views and base tables can be combined to answer a query consistently. We summarize in Section 7.

## 2 Definitions and Notation

A *base table* is a table that is not defined as a view. A *materialized view*  $V$  is a table that is defined by a query  $Q_V$ , and stores the result of  $Q_V$  in the database. A database *state*  $s$  is a mapping from each (base or materialize view) table  $R$  in the database schema to the multiset of records stored for  $R$ , denoted  $R_s$  ( $s : R \rightarrow R_s$ ). Let  $B^1 \dots B^n$  be the base tables from which  $V$  is ultimately derived. For a database state  $s$ ,  $Q_V(s)$  is the result of evaluating the expanded version of  $Q_V$  (that only involves base tables) over  $B_s^1 \dots B_s^n$ , i.e., the value of  $V$  when no views are materialized. A materialized view  $V$  at database state  $s'$  is *consistent*

with database state  $s$  if  $V_{s'} = Q_V(s)$ . A set of materialized views are *mutually consistent* with database state  $s$  if all of the views are consistent with  $s$ .

A *transaction*  $x$  is a function that maps one database state to another ( $x : s_1 \rightarrow s_2$ ). In a DBMS, many transactions can run concurrently, but the DBMS ensures that transactions appear to run according to some serial schedule. Except where we explicitly state otherwise, we assume that locking is used to ensure a serial schedule, and that locks are held until the end of transaction. With locking, the serial schedule is determined by the order in which the transactions commit.

If two transactions conflict (i.e., one transaction is dependent on the results of the other), or if one transaction completes before another begins, then the commit order is always well defined. However, if two transactions are run concurrently and do not conflict, then the order is not well defined (e.g., in a MPP environment). Fortunately, either order produces a valid serial schedule, and the same state is reached once both transactions have been applied. So a set of transactions  $X$ , which includes the partial order on the transactions, is a function that maps from one database state to another:

$$X : s_1 \rightarrow s_2 \equiv x_n \circ x_{n-1} \circ \dots \circ x_1$$

where the  $x_i$  are all the transactions of  $X$  in any valid serial order.

Using the partial order on transactions defined by transaction conflicts and serial execution, we can define a partial order on the valid database states. Let  $X_s$  be the partially ordered set of transactions that are reflected in state  $s$ . A *valid database state*,  $s$ , is a database state such that  $X_s$  is consistent with the partial order of transactions, each transaction is applied at most once, and all transactions that some transaction in  $X_s$  depends upon is in  $X_s$  ( $\forall x \in X_s, \forall x' < x, x' \in X_s$ ). Given two valid database states  $s$  and  $s'$ ,  $s < s'$  iff  $X_s \subset X_{s'}$ ;  $s = s'$  iff  $X_s = X_{s'}$ .

A *delta table* is a table that describes changes to another (base or materialized view) table  $R$ , denoted  $\Delta R$ .  $\Delta R$  has the same attributes as  $R$  plus (at least conceptually) three additional attributes: **xid**, **timestamp**, and **count**. The **xid** is the identifier of the transaction that caused the change to the view. The **timestamp** is the time at which the change to the view (actually or should have) occurred. The expression  $\sigma_{t_1 < \text{timestamp} \leq t_2}(\Delta V)$ , denoted  $V_{1,2}$ , is the set of changes to the view from  $t_1$  to  $t_2$ .

For aggregate views, the **count** is the change to the count attribute in the view for the associated group,<sup>3</sup> and the other aggregates in the view delta are also relative values (e.g., for a **sum**, the value in the view delta is the increase or decrease to the **sum** in the view).<sup>4</sup> When the **count** for some group in the view becomes zero, the group is deleted. This type of view delta is called a *summary delta table* [8].

For non-aggregate views (e.g., a select-project-join view), a **count** of  $+n$  represents the insertion  $n$  copies of its associated tuple into the view, and a **count** of  $-n$  is the deletion of  $n$  copies. An update to a record is represented as a deletion of the old value followed by an insertion of the new value. We can think of a

<sup>3</sup>The **count** is required to be included in an aggregate view.

<sup>4</sup>We only consider distributive aggregate functions (e.g., SUM). Support for non-distributive aggregates (e.g., MIN, MAX) requires can be handled with extensions that are beyond the scope of this paper.

non-aggregate view as aggregate view with a group by on all of its attributes, and a count attribute that represents the number of duplicates.

### 3 Properties of IPDA Implementations

The goal of the IPDA apply process is to get the same result as a full refresh performed at the specified refresh time. To accomplish this, the system must exactly identify all records in the delta table that *committed* between the previous refresh of the view and the specified time. Once the system finds and applies the records, it must be able to run the next apply process, so the system must maintain sufficient information to determine which records were previously applied.

The methods for finding the committed records are described in Section 4. In this section, we consider the properties that IPDA implementations must possess to ensure the consistency of the view. In particular, we evaluate the correctness our IPDA implementations based on the following properties:

1. *No lost updates.* Any acceptable method for IPDA must ensure that all updates to the view will eventually be applied. Lost updates come in two flavors: lost updates within a transaction, and lost transactions. If part of a transaction is lost, then transactions are not applied atomically. If entire transactions are lost, then transaction dependencies are not preserved.
2. *No redundant updates.* An IPDA method must not apply any update more than once.
3. *Transactions applied atomically to a single view.* An IPDA method must apply all of the effects of a transaction to a single view at the same time.
4. *Transactions applied atomically across views.* When two views are refreshed to the same time, both views should reflect the effects of the same transactions. If one view has an earlier refresh time, it should reflect the effects of a subset of the transactions applied to the later view.
5. *No transactions reordered.* Given two transactions  $x_1$  and  $x_2$ , if  $x_1 < x_2$  then the effects of  $x_1$  should be applied to a view before or at the same time as the effects of  $x_2$ .
6. *Real time preserved.* The changes applied up to a specific time should be exactly the changes that were committed at that time. This property implies all of the previous properties, and also states that the method must consider real time.

If an implementation has these properties, then it produces consistent answers. In the remainder of this section, we argue this point more formally.

Let  $V_s$  be a materialized view that is consistent with some valid database state  $s$ . We assume that every view is consistent when it is first created. Let  $U$  be the set of updates to be applied to  $V_s$ . Every update was caused by some transaction, so let  $U_x$  be the set of updates to  $V$  for a transaction  $x$ . Let  $X_U$  be the set of transactions with some update to  $V$  in  $U$ .

**Lemma 1** *If all the updates in  $U$  belong to a single transaction  $x$ , all the updates of  $x$  are in  $U$  ( $X_U = \{x\}$  and  $U = U_x$ ), and no update is applied more than once, then the result,  $V_{s'}$ , is consistent with the valid database state  $s' = x(s)$ .*

**Proof** Assume  $V_{s'}$  is not consistent with  $s'$ . Then either  $U$  must contain some update that was caused by some other transaction  $x' \neq x$  (so  $X_U \neq \{x\}$ ),  $U$  must be missing some update caused by  $x$  ( $U \neq U_x$ ), or some update was applied more than once. No matter what, we reach a contradiction. ■

**Lemma 2** *Let  $X'$  be the set of transactions with some update to  $V$  plus the transactions on which these depend that are not reflected in state  $s$  ( $X' = X_U \cup \{x' \mid \exists x \in X_U, x' < x \wedge x' \notin X_s\}$ ). If (1) all the updates to  $V$  from any transaction are applied atomically ( $\forall x \in X_U, \forall u \in U_x, u \in U$ ), (2) transaction dependencies are preserved ( $\forall x \in X_U, \forall x' < x, x' \in X' \vee x' \in X_s$ ), and (3) no update is applied more than once, then the result,  $V_{s'}$ , is consistent with the valid database state  $s' = X'(s)$ .*

**Proof** Let  $x_1 \dots x_n$  be a valid serial order of all transactions in  $X'$ . Let the database states  $s_0 = s$  and  $s_i = x_i(s_{i-1}), 1 \leq i \leq n$ . So,  $s_n = s'$ . By Conditions 2 and 3, all the  $s_i$  are valid database states. Let  $V_{s_i}$  be the result of applying the updates of transaction  $x_i$  to  $V_{s_{i-1}}$ . By Conditions 1 and 3 and Lemma 1,  $V_{s_1}$  is consistent with  $s_1$ , and inductively,  $V_{s_i}$  must be consistent with  $s_i$ . Therefore,  $V_{s'} = V_{s_n}$  must be consistent with the valid database state  $s_n = s' = X'(s)$ . ■

## 4 Finding Committed Records

The major challenge in implementing IPDA is finding the records in the delta table to apply. We present a method to solve this problem that explicitly stores the transaction commit order (or an approximation to it) in the delta table. We first describe the basic algorithm, and then refine it to solve consistency problems.

In Section 2, the `timestamp` attribute of a delta table was the time that the record was committed. Since we do not know the commit time of a record when we write it,<sup>5</sup> if we instead store the time that the record was written, can we use this information to find the records to apply? The answer is no for a several reasons, but let's see how close we can get.

Algorithm 1 presents the initial apply process. An example of a delta table is shown in Figure 2. Step 1 finds the transactions that made changes to the view  $V$  between  $t_1$  and  $t_2$ . In this case, transactions are 2, 3, and 4 made changes in the time window, but transaction 4 continued past  $t_2$ , so it is eliminated in Step 2. Step 3 then applies the two changes from transaction 2 and the two changes from transaction 3.

In the following subsections, we discuss the problems with Algorithm 1, and we give solutions to these problems. In Section 4.8, we combine all the solutions together into an improved version of the algorithm, and we prove that the algorithm correctly maintains an IPDA view.

<sup>5</sup>We need the commit time of the transaction that is writing the records

---

**Algorithm 1** Initial Apply Process

---

Let  $t_1$  be the current refresh time of a materialized view  $V$ , and  $t_2$  be the time that we want to roll  $V$  up to.  $t_2$  must be before the current time.

**Step 1:** Read the records of  $\Delta V$  such that  $t_1 < \text{timestamp} \leq t_2$ , and make a list of all the  $\text{xids}$  that appear.  $\Delta V$  is read using read locks (cursor stability is sufficient), so it will only see committed records, which also implies that apply will stall while waiting for any long running transactions to commit.

**Step 2:** We now know all the transactions that made changes to  $V$  during the time window. However, some of these transactions might have continued past  $t_2$ , so read the records of  $\Delta V$  with a time greater than  $t_2$ , and remove the  $\text{xids}$  associated with these records from the list generated in Step 1. This step can read  $\Delta V$  without locking (dirty read), which helps reduce lock contention between apply and the update transactions. We know that any  $\text{xid}$  that remains in the list after this step will not be added to  $\Delta V$  after we finish Step 2 because all the  $\text{xids}$  seen in Step 1 were committed.

**Step 3:** Using the set of  $\text{xids}$  that remain after Step 2, we can retrieve all the records of  $\Delta V$  that have a  $\text{xid}$  in this set and apply them to  $V$ . Again, this step can proceed without locking because we are only reading committed records (guaranteed by Step 1) that are never updated (update transactions only insert records into the delta table). Also, records with a time less than  $t_1$  will be read for transactions that started before  $t_1$ .

---

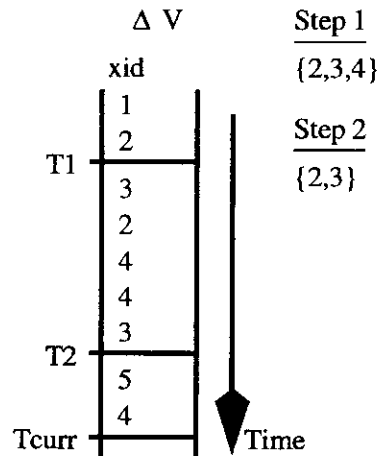


Figure 2: Example Delta Table

## 4.1 Problem: Lost Updates

The most serious flaw with Algorithm 1 is that it is possible for the system to miss records; in other words, some records in  $\Delta V$  might never be applied. Consider the following situation. A transaction,  $x_1$ , is about to add one and only one record to  $\Delta V$ . The propagate process reads the clock at 2:59, and then for some reason, the process is suspended. Then at 3:01, the apply process comes along and reads all the records from 2:00 to 3:00, missing the record, so  $x_1$  is not in the set of  $xids$  generated in Step 1. Now  $x_1$  resumes processing and adds its record to  $\Delta V$  with a time of 2:59 and commits. When the next apply is performed, it will read all the records between 3:00 and 4:00, and again miss the  $x_1$  record.

## 4.2 Solution: Atomic Clock and Insert

The problem is that reading the clock and inserting the record is not an atomic operation. In a transaction log, the time for each log record, the log sequence number (LSN), is assigned at the same time that the record is inserted into the log. Combining the read of the clock with the insertion allows us to assert that if we read the log from  $LSN_1$  to  $LSN_2$ , we have seen all the log records between  $LSN_1$  and  $LSN_2$ .

So the solution to lost updates is to ensure that reading the clock and inserting a record into the delta table cannot be separated by a read of the table. This can be achieved at various levels of granularity that trade-off concurrency and overhead. We describe two methods below.

A coarse-grained method is to obtain a lock (this is not a traditional lock) on each delta table that a statement might insert into (or even one global lock for all the delta tables) at the beginning of the statement, and to release the locks at the end of the statement. The lock can allow multiple writers (update transactions) to proceed, but it blocks readers (the apply process). Concurrency can be improved by adding a predicate to the lock. When a lock is obtained, the current time is associated with the lock. When a reader attempts to get past the lock, it provides the maximum timestamp that it is interested in reading, and it can proceed as soon as the minimum timestamp of all the writer's locks is greater than the reader's maximum timestamp. This locking method works well with IPDA because the apply process is intended to remain far enough away from the tail of the log to avoid conflicting with the update transactions.

A fine-grained method is to obtain a lock before each insertion to a delta table. After the lock is obtained, the clock is read and the record is inserted, then the lock is released. The same predicate locking as described above will work in the fine-grained method as well.

In an MPP environment, the locks can be distributed. Each writer can grab its lock locally, while the reader will grab a lock at every site that it plans to read. Distributing the locks in this manner is beneficial to the system because we expect many more writers (update transactions) than readers (the apply process), and writers will often hit only a single node, whereas the readers will often hit all the nodes of a delta table.

Using locks creates one tricky issue: if an error occurs while inserting the records, after the lock is obtained, special handling is required to release the lock properly. So simply implementing the lock and

unlock with SQL functions that are evaluated before and after the delta table inserts is insufficient. In other words, while processing the compound SQL statement:

```
SELECT ipdaLock() FROM VALUES(1);
INSERT INTO DeltaTable [propagated records];
SELECT ipdaUnlock() FROM VALUES(1);
```

if an error is generated during the INSERT, the system would skip the ipdaUnlock() call.

This problem can be solved in several ways. First, we can add a mechanism to force a statement to be evaluated, even if an error occurred in a preceding statement. Second, at transaction end, the system could release any locks that the transaction is still holding. The IPDA lock could even be placed in the system lock table to be released along with all the traditional locks. Third, when the reader attempts to acquire the lock, it could first release any locks held by nonexistent transactions.

### 4.3 Problem: Transactions Not Atomic Across Views

The list of  $xids$  generated after Step 2 of Algorithm 1 is not an accurate list of the transactions that committed between  $t_1$  and  $t_2$ . A transaction could continue to make additional changes that affect other tables or views and no trace would be found in  $\Delta V$ . Consider the following views:

```
Withdrawals = SELECT id, sum(amount)
               FROM Accounts
               WHERE amount < 0
               GROUP BY id
```

```
Deposits =    SELECT id, sum(amount)
               FROM Accounts
               WHERE amount > 0
               GROUP BY id
```

Say some transaction transfers money from account  $a_1$  to account  $a_2$  at 2:58, and the propagation process adds a record to  $\Delta Withdrawals$  at 2:59 and to  $\Delta Deposits$  at 3:01. If we now refresh both views to 3:00, the effect of the transfer will appear in  $Withdrawals$  but not in  $Deposits$ . So, the sum of withdrawals could be higher than the sum of deposits (a bankers nightmare!), i.e., the transactional consistency across views is violated.

### 4.4 Problem: Transactions Reordered

Since the committed transaction list is inaccurate, it is possible to reorder transactions. For example, consider two transactions,  $x_1$  and  $x_2$ . Say  $x_1$  withdraws \$100 from account  $a_1$  and deposits it from account  $a_2$ , and

$x_2$  deposits \$100 into  $a_3$  and withdraws it from account  $a_2$ . Further assume that  $x_1$  committed before  $x_2$ , that the starting balance of  $a_1$  is \$100, and that both  $a_2$  and  $a_3$  are empty.

The deposit of  $x_2$  could be recorded at 2:59, the withdrawal of  $x_1$  at 3:01 ( $x_2$  is now waiting for  $x_1$  to commit), the deposit of  $x_1$  at 3:02, and the withdrawal of  $x_2$  at 3:03. If we refresh Deposits to 3:00, account  $a_3$  has \$100 in deposits, but account  $a_2$  has no deposits. Since we know that  $a_3$  received its money from  $a_1$  via  $a_2$ , whenever the  $a_3$  deposit shows up, the  $a_2$  deposit should as well, but it did not. The reason for this anomaly is that  $x_2$  was applied, but  $x_1$  was not, which violated the transactional dependency from  $x_1$  to  $x_2$ .

## 4.5 Solution: Commit Records

One way to achieve transactional atomicity across views and to avoid reordering transactions is to add a sentinel record to each delta table that a transaction affected when the transaction commits. All of the sentinel records for the same transaction are assigned the same timestamp, which will be greater than or equal to all the timestamps for the normal delta table records for that transaction.

Actually, the timestamps on the normal records are no longer needed because we only need the maximum timestamp for each transaction. In Algorithm 1, the maximum timestamp is used as an approximation to the commit timestamp, but now the commit timestamp is recorded. So we can record an invalid timestamp for normal records (e.g., NULL or some timestamp that will never be a commit time, like Jan 1, 1900). If we do this, then Step 1 will only consider commit records. Step 2 can be eliminated because Step 1 no longer generates transactions that committed after the apply window.

To add the commit records, the system must track all of the delta tables that were touched by a transaction. We can eliminate this requirement, and reduce the commit records, by placing the commit records in a separate table, which we call the *commit table*. If a transaction added a record to any delta table, it places a single record in the commit table. The record consists of the transaction identifier and the commit time. The delta tables no longer need the timestamp attribute, but still have the xid attribute. Step 1 can now generate its transaction list by reading the commit table with read locks instead of the delta table.

To implement this solution, the system must be able to perform additional work after all of the user work and before the transaction commits. This is essentially a commit trigger, except that we do not need the guarantee that the transaction will definitely commit. Algorithm 1 only examines committed commit records, if a transaction aborts, any uncommitted commit or delta records can be safely rolled back.

The major drawback of using the commit table is it becomes more difficult to remove commit records that are no longer needed. The reason is that the state of all the materialized views in the system must be considered before deleting a commit record. The system can purge any commit records that are older than the minimum refresh time of all the materialized views, but if the system does not look into all of the delta tables to determine the exact set of transactions that it must keep in the commit table, then a single materialized view that is very stale could cause the commit table to become unnecessarily large.

A second drawback is the commit table is less fault-tolerant than distributing the commit records in the

delta tables. When the commit records are in the delta tables, only the nodes that store the relevant delta table records and the materialized view are required to be accessible for the view or its base tables to be updated. When the commit table is used, if a node that contains (the relevant portion of) the commit table is down, then no materialized views can be maintained, and no update transactions can complete.

## 4.6 Problem: Multiple Clocks

Up to this point, we implicitly assumed that one universal clock was used for all timestamps, including commit times. Unfortunately, in a parallel (MPP) system, each node has its own clock that is not perfectly synchronized with all the others. If we blindly use the different clocks, two problems ensue.

First, we could lose updates because the apply process can read all the records in the delta table up to time  $t_1$  according to the clock at one node, and later some update transaction starts and inserts a single record into the delta table with time  $t_2$  according to the clock at a different node. However, because of discrepancies in the clocks,  $t_2$  could be before  $t_1$ . Therefore, the next time around, the apply process will believe it processed the record at  $t_2$ , and the update is lost.

Second, we could reorder transactions. If the commit records for two transactions  $x_1$  and  $x_2$ ,  $x_1 < x_2$ , received commit timestamps of  $t_1$  and  $t_2$  from two different clocks, then  $t_2$  could be earlier than  $t_1$ . Since the timestamps determine the commit order, it appears to the apply process that  $x_2 < x_1$ , so it can apply the changes of  $x_2$  to the view without applying  $x_1$ .

## 4.7 Solution: Use One Clock

The solution to the multiple clock problem is easy: only use one clock. If the same clock is used by all the IPDA processes, then this problem is easily skirted. The way to implement the single clock in a MPP environment is to issue a remote procedure call (RPC) to a designated clock node. One sticky issue is RPCs can be expensive if issued frequently. Fortunately, the solution in Section 4.5 reduces the number RPCs needed from once per delta table record to once per IPDA transaction.

A second problem is this one designated clock node makes the system susceptible to a single-node failure. If we need to shutdown the clock node (cleanly) for some reason, we can easily make some other node the clock node by sending the current timestamp to the new node. But if the clock node crashes, then any transaction that performs IPDA propagation cannot commit until the clock node is back up, and IPDA views can be refreshed.

We have a two work-arounds for this problem. First, whenever a timestamp is requested, the clock node can synchronously give the timestamp to one or more buddy nodes before returning the timestamp to the caller. If the clock node crashes, the system can make one of the buddies the clock node. As a second work-around, if we can assert (with reasonable certainty) that the last timestamp given out by the clock node is less than  $k$  seconds past the maximum time of all the nodes that are still up, then we can delay any IPDA commits for  $k$  seconds and make the node with the maximum time the clock node.

## 4.8 Correct Algorithm

---

**Algorithm 2** Corrected Apply Process

---

Let  $t_1$  be the current refresh time of a materialized view  $V$ , and  $t_2$  be the time that we want to roll  $V$  up to.  $t_2$  must be before the current time. All timestamps come from a single designated clock node.

**Step 1:** Acquire an IPDA read lock on  $\Delta V$  for the time range up to  $t_2$  as described above. The lock can be immediately released once acquired.

**Step 2:** Find the  $xids$  in the commit records (from  $\Delta V$  or the commit table) with  $t_1 < \text{timestamp} \leq t_2$  using read locks (cursor stability is sufficient).

**Step 3:** Using the set of  $xids$  generated in Step 2, we can retrieve all the records of  $\Delta V$  that have a  $xid$  in this set and apply them to  $V$ . This step can proceed without locking because we are only reading committed records (guaranteed by Step 2).

---

Algorithm 2 presents the corrected version of the apply process. In this section, we show that Algorithm 2 is indeed correct and that it satisfies the properties listed in Section 3. In proofs that follow, we assume that:

1. Timestamps are acquired properly from the clock node. Lemma 3 proves that the commit time that we associate with a transaction — some time after the last lock is obtained and before the true commit time of the transaction — produces a valid serial ordering.
2. Unless otherwise specified, full read/write locking is performed, and all locks are held to end of transaction.
3. The propagate process correctly writes its delta and commit records.
4. Any process that prunes the delta tables or commit tables never deletes any record that is still required by some view.

**Lemma 3** *Let every transaction  $x$  that actually commits be assigned a commit time of  $t_x$ . If the assigned commit time  $t_x$  is after the time that  $x$  acquired its last lock ( $t_x^a$ ), and before the time that  $x$  actually committed ( $t_x^c$ ), then ordering the transactions by their assigned commit times is a valid serial order.*

**Proof** Assume that ordering by the assigned commit times is not a valid serial order. Then there exists a pair of transactions,  $x$  and  $y$ , such that  $y$  depends on  $x$  ( $x < y$ ), but  $t_x \geq t_y$ . Since  $x < y$ , then  $x$  committed before  $y$  started, or  $x$  must modify some record  $r$  that  $y$  depends on. If  $x$  committed before  $y$ , then  $t_x < t_x^c < t_y^a < t_y$ , which is a contradiction. If  $x$  modifies a record  $r$ , then it must acquire a write lock on  $r$ , and the lock must be held until  $x$  commits. Furthermore,  $y$  must acquire a lock on  $r$ , but it cannot until after  $x$  commits. So,  $t_x < t_x^c < t_y^a < t_y$  which is a contradiction. Therefore, the assigned commit times must produce a valid serial order. ■

**Lemma 4** *Algorithm 2 applies transactions atomically to a single view.*

**Proof** Assume transaction are not applied atomically to view  $V$ . Then there exists a transaction,  $x$ , such that some effect  $e_1$  of  $x$  has been applied to  $V$ , but some other effect  $e_2$  has not. Since Step 2 only sees committed records, all the effects of  $x$  must have been in  $\Delta V$  before any  $e_1$  was applied. For  $e_1$  to be applied to  $V$ ,  $x$  must have been in the list of transactions generated in Step 2. Since all the effects of  $x$  are in  $\Delta V$ , Step 3 must have found effect  $e_2$  and applied it. This is a contradiction. ■

**Lemma 5** *Algorithm 2 applies all the changes that (effectively) committed between  $t_1$  and  $t_2$ , and no others.*

**Proof** Assume transaction  $x$  committed at (assigned commit) time  $t_x$ ,  $t_1 < t_x \leq t_2$ , but  $x$  was not applied by Algorithm 2. From Lemma 4, we know that all the updates for transaction  $x$  must have been missed. Since the commit time is acquired after all the delta records are written, all the delta records must have been written at or before  $t_x$ . Let  $t_a$  ( $t_a > t_2$ ) be the time that apply acquires its IPDA read lock. For Algorithm 2 to miss the commit record, the commit record must have been written at some  $t_3 > t_a$ , but then transaction  $x$  should have acquired an IPDA write lock at or before  $t_x < t_2$ , so the IPDA read lock in Step 1 would have blocked until the commit record was written and the lock was released ( $t_a > t_3$ ) — a contradiction.

Now assume transaction  $x$  committed at (assigned commit) time  $t_x$ ,  $t_x \leq t_1$  or  $t_x > t_2$ , but  $x$  was applied by Algorithm 2. Only transactions with  $t_1 < \text{timestamp} \leq t_2$  are generated in Step 2, so transaction  $x$  could not have been applied. ■

**Lemma 6** *Algorithm 2 never applies any update more than once.*

**Proof** Assume Algorithm 2 applies some update of some transaction  $x$  more than once. From Lemma 4, we know that all the updates of  $x$  are applied atomically, so the entire transaction must have been applied more than once. Each time Algorithm 2 is run, a new, non-overlapping time window is examined in Step 1, so the commit record for transaction  $x$  must appear in more than one time window, which is impossible since each transaction writes only one commit record. ■

**Lemma 7** *Algorithm 2 applies transactions atomically across views.*

**Proof** Assume transactions are not applied atomically across views. Then there exists a transaction,  $x$ , such that some effects of  $x$  are in some view  $V_1$ , but not in  $V_2$ . From Lemma 5 we know that  $x$  must have committed between  $t_1$  and  $t_2$  to be applied to  $V_1$ , and that all transactions that committed between  $t_1$  and  $t_2$  must be applied to  $V_2$ . Since each transaction writes only one commit record to the commit table (or the same commit time to all the delta tables),  $x$  must have been applied to  $V_2$  as well — a contradiction. ■

**Lemma 8** *Algorithm 2 does not reorder transactions.*

**Proof** Assume Algorithm 2 does reorder transactions. Then there exists a pair of transactions,  $x$  and  $y$ , such that  $y$  depends on  $x$  ( $x < y$ ), but  $y$  is applied, and  $x$  is not. Let  $t_x$  and  $t_y$  be the (assigned) commit times of  $x$  and  $y$ . From Lemma 3, we know  $t_x < t_y$ . If  $y$  is applied,  $t_1 < t_y \leq t_2$ . If  $t_1 < t_x < t_y \leq t_2$ , then

from Lemma 5,  $x$  must also be applied. Otherwise,  $t_x \leq t_1 < t_y$ . But then there must be some time window  $t_a$  to  $t_b$ ,  $t_a < t_x \leq t_b \leq t_1$ , over which Algorithm 2 was previously run, and from Lemma 5,  $x$  was applied in that run. So Algorithm 2 did not reorder the transactions. ■

**Theorem 1** *If Algorithm 2 is used to independently refresh a set of views  $V_i$  to time  $t$ , then the views are mutually consistent with a valid database state at  $t$ .*

**Proof** From Lemma 5 we know that each  $V_i$  has all the updates that were committed by  $t$ . From Lemma 7 we know that all the  $V_i$  are at the same database state. From Lemma 4, Lemma 8, and Lemma 2 we know that each  $V_i$  is consistent with a valid database state. Therefore, the  $V_i$  are consistent with a valid database state at  $t$ . ■

## 5 Net Effect

Once we have the records to apply from Algorithm 2, how these records are applied can be defined in two ways. The first is to insist that every change to a view caused by a transaction must be applied in exactly the order they occurred in the transaction, but simply deferred to a later time. This implies that every record added to the delta table must be completely ordered.

The second way is to consider all the changes at the same time, and to allow the *net effect* of the changes to be applied in any order. The net effect of a delta is the obtained by grouping the delta by the grouping attributes (all attributes except count for a non-summary delta table), and combining the aggregate attributes (including the special count column) using the appropriate function (e.g., sum for sum and count aggregates). If total count in a non-summary delta table is zero, the record is dropped. We cannot drop summary delta records with a total count of zero because other aggregates might be non-zero, which means that the total number of records remained the same, but the aggregates changed. The `xid` and `timestamp` attributes are eliminated by the net effect.

The net effect is best illustrated by an example. Say a transaction causes a record to be inserted to a view, and a subsequent transaction causes that record to be deleted. If both the transactions are applied together, then the net effect allows us cancel the insertion with the deletion, so the record is never applied to the view. For another example, consider the `StateCount` view from Section 1.2, and a single transaction that issues one statement to insert five sales records for Wisconsin, and then a second statement to delete seven records for Wisconsin. Using net effect, the inserts and deletes are combined and applied as a single change to the view that drops the count by two. If the count drops to zero, then there are no sales records for Wisconsin, so the Wisconsin record is deleted from `StateCount`.

The net effect avoids replaying history, which simplifies the IPDA process, particularly in the presence of constraints on the view. Consider the following materialized view:

```
WICust = SELECT  c.custId, c.name
```

```

FROM    Customer c
WHERE   c.state = 'WI'

```

The `custId` field is a key, so we can build a unique index on it. Now say a particular customer, “Paul’s Petunias”, is in the view, but then Paul retires and sells the business to Peter. Peter discovers that he is deathly allergic to petunias and promptly sells the business to Patty. Since updates to the base table appear as delete–insert pairs,  $\Delta$ WICust will contain the following records:

xid	timestamp	count	custId	name
5	10	-1	52	Paul's Petunias
5	20	+1	52	Peter's Petunias
9	30	-1	52	Peter's Petunias
9	40	+1	52	Patty's Petunias

We cannot apply these records to the view in any arbitrary order. For example, if we first applied the second record, we would violate the uniqueness constraint on `custId`. If we applied the third record first, we would not find the record for Peter’s Petunias, which should never happen during apply. Also, if we just ignored the error, then we would add Peter’s Petunias later, and it would not be deleted. So we have no choice but to apply the records in the proper sequence.

The net effect of the above delta table eliminates all but the first and last record. The net effect of a non-aggregate view always results in at most two records for any particular key of the view. The first record is a deletion of the existing record, and the second is an insertion of a new record. The net effect of aggregate views is always one record per group.

Therefore, by using the net effect, we can avoid ordering records of the delta table that are applied together. In particular, we can avoid ordering changes from a single transaction. For aggregate views, each group appears only once, so all the records are independent. For non-aggregate views, we know that the deletion record must occur before the insertion for a particular key because any remaining deletion is for the existing record with that key in the view, and the insertion is the value that should be in the view for that key. Therefore, if we perform all the deletions before the insertions, we know that we will not violate any uniqueness constraints, and we will not attempt to delete any non-existent records.

Another benefit of using the net effect is that we can reduce the space required to store the delta table. For example, if the changes to a view are applied at the end of the month, we could replace the contents of delta table with its net effect at the end of each day, which could result in a significantly smaller delta table. Also, we can choose to compress a range of records based on the commit timestamps of the transactions instead of the entire delta.

The net effect of a delta table eliminates the `xid` and `timestamp` fields from the delta. We can think of the net effect as a delta for a single transaction where the `xid` is any `xid` in the delta, and the commit timestamp for `xid` is the maximum timestamp in the delta table, or the high timestamp used select the range.

When we compress a range of a delta table (or the entire delta table) with net effect in this manner, we cannot allow the apply process to roll a view to a time that falls in the compressed range, otherwise the view might not be consistent with other views rolled to the same time. For example, if we compressed the range of records from 3:00 to 4:00, then changes that really occurred at 3:15 would be labeled as occurring at 4:00, or eliminated altogether by another change that occurred at 3:45. If we rolled the view to 3:30, the view would not reflect the database state at 3:30 because the compressed view delta eliminated the details on transactions between 3:00 and 4:00. On the other hand, if we roll the view to 3:00 or 4:00 the view will be consistent.

The use of net effect has one caveat: it changes the semantics of referential integrity constraints and triggers on the materialized view. For example, if a view has a delete-cascade constraint, a delete-insert pair in the delta table could be cancelled out, and the delete-cascade will not happen. For triggers, obviously the net effect can cause the insertion and deletion triggers to be fired fewer times.

On the other hand, just deferring the apply phase already changes the semantics of referential integrity constraints and triggers. Consider the following case: First, a deletion is added to the delta table. Second, a record is added to a table that is referred to by the delete-cascade that matches the previous deletion. Third, the changes are applied and the newly inserted record gets deleted.

## 5.1 Correctness of Net Effect

In this section, we formalize the preceding arguments, and prove that applying the net effect of summary and non-summary delta tables produces the same result as applying the delta table in the exact order that it was created. We also prove that the net effect of a non-summary delta table for a view with a key produces at most two records for each key.

**Lemma 9** *The result of applying the net effect of a summary delta table  $\Delta V$  is the same as applying the  $\Delta V$  directly.*

**Proof** Let  $V$  be the original view,  $V'$  be the result of applying  $\Delta V$  without the net effect, and  $V''$  be the result of applying the net effect of  $\Delta V$ . The apply process ensures that no group (value of the GROUP BY columns) appears more than once in either  $V'$  or  $V''$ . If  $V' \neq V''$  then at least one of the following must be true:

1. We have a group  $g$  in both  $V'$  and  $V''$  that has a different value for some aggregate column  $a$ . Let  $f$  be the aggregate function used to compute  $a$ . Let  $a_0$  be the value of column  $a$  for group  $g$  in the original view  $V$ , if it exists. Let  $a_1, \dots, a_n$  be the  $a$  values of records in  $\Delta V$  that have GROUP BY value  $g$ . Since we assume that all aggregate functions are distributive,  $f(\{a_0, a_1, \dots, a_n\}) = f(\{a_0, f(\{a_1, \dots, a_n\})\})$  if  $a_0$  exists, and  $f(\{a_1, \dots, a_n\}) = f(\{f(\{a_1, \dots, a_n\})\})$  if  $a_0$  does not exist. So the value of column  $a$  must be the same in both  $V'$  and  $V''$ , which is a contradiction.

2. We have a group  $g$  in  $V''$  that does not exist in  $V'$ , or vice versa. Since the net effect of  $\Delta V$  performs a GROUP BY on  $\Delta V$ , it cannot create groups or lose groups (and we do not drop zero count records from a summary delta table). Therefore, the cause must be that the final count for group  $g$  in  $V'$  (respectively  $V''$ ) must have dropped to zero, but be greater than zero in  $V''$  ( $V'$ ). The preceding case showed that the result of aggregates, which includes the count must be the same — a contradiction.

■

**Lemma 10** *The result of applying the net effect of a non-summary delta table  $\Delta V$  is the same as applying  $\Delta V$  in the proper order.*

**Proof** Let  $V$  be the original view,  $V'$  be the result of applying  $\Delta V$  to  $V$  without the net effect, and  $V''$  be the result of applying the net effect of  $\Delta V$  to  $V$ . All count values in  $\Delta V$  are either +1 (for an insertion) or -1 (for a deletion).

Let  $r_1 \dots r_n$  be the records of  $\Delta V$  in the order they were applied to compute  $V'$ . Let  $V'_0 = V''_0 = V$ . Let  $V'_i$  be the result of applying the first  $i$  records of  $\Delta V$  to  $V$  without the net effect, and let  $V''_i$  be the result of applying the net effect of the first  $i$  records to  $V$ .  $V'_i$  is the same as applying  $r_i$  to  $V'_{i-1}$ . Let  $c_{r,i}$  be the count of the number of records in  $\{r_1 \dots r_i\}$  that have the same view attribute values as the record  $r$ .

Assume  $V'_{i-1} = V''_{i-1}$ . Consider any record  $r$  in  $V'_{i-1}$ . Assume  $r$  does not have the same view attribute values as  $r_i$ . Then the number of records with the same values as  $r$  is unchanged from  $V'_{i-1}$  to  $V'_i$ . In addition,  $c_{r,i} = c_{r,i-1}$ , so the number of  $r$  records is unchanged from  $V''_{i-1}$  to  $V''_i$ .

Otherwise,  $r$  and  $r_i$  have the same view values. If the count of  $r_i$  is +1, then the number of records with the same values as  $r$  increases by one in  $V'_i$ . Also,  $c_{r,i} = c_{r,i-1} + 1$ , so the number of  $r$  records in  $V''_i$  increases by one. If the count is -1, then the number of  $r$  records goes down by one in both  $V'_i$  and  $V''_i$ . Therefore,  $V' = V'_n = V''_n = V''$ . ■

**Lemma 11** *The result of applying the net effect of a non-summary delta table will have at most two records for any particular key: a deletion of an existing record, and an insertion of a new record.*

**Proof** We use the same definitions as Lemma 10. Consider any key  $k$  present in  $\Delta V$ . There can be at most one record in  $V$  with key  $k$ . Let  $r'_1 \dots r'_m$  be the records in  $\Delta V$  that have key  $k$ .

Case 1: No record with key  $k$  is in  $V$ .  $r'_1$  must be an insertion because there no record to delete in  $V$ .  $r'_2$ , if it exists, must be a deletion of  $r'_1$ , otherwise two records with key  $k$  would be in the view. Similarly for any odd  $i < m$ ,  $r'_i$  is an insertion because there is no record to delete, and  $r'_{i+1}$ , if it exists, must be a deletion of  $r'_i$  because otherwise two records with the same key would be in the view. Therefore, if  $m$  is odd,  $r'_m$  is the only insertion that is not cancelled, and if  $m$  is even, all the insertions are cancelled. Therefore the net effect of  $\Delta V$  has at most one insertion.

Case 2: One record with key  $k$  exists in  $V$ . Then  $r'_1$  must be a deletion of the record in  $V$  with key  $k$  because an insertion would violate the key constraint. Using an argument similar to Case 1, for even  $i$ ,  $r'_i$  is

an insertion, and  $r'_{i+1}$  is a deletion of  $r'_i$ . If  $m$  is odd, then all the insertions are cancelled, and  $r_0$  is the sole deletion. If  $m$  is even,  $r'_m$  is the one insertion that survives, and  $r_0$  the deletion, unless  $r'_m$  matches  $r'_1$ , in which case, no insertion or deletion survives. ■

## 6 Valid Time

A major advantage of IPDA (and DPDA) is its ability to bring a collection of views to a mutually consistent state. We can extend this single time point to an time window that allows base tables and materialized views with differing maintenance polices to be combined together. A similar problem is addressed in [4], but they statically determine which views should be mutually consistent and they enforce it by maintaining all the at the same time. The solution described here allows views to be maintained independently, and then dynamically decides to combine multiple views and base tables if it produces a consistent answer.

Every view can maintain a time interval for which the view is current, called the *valid time* of the view. Each base table can also maintain the time of its most recent change. The valid time of a base table is from its most recent change to the current time. Our notion of valid time is closely related to transaction time in temporal databases [13]. The major difference is that each table or materialized view has only one valid time that is accessible at any given moment, and that we associate a time interval with the entire table instead of with each row.

If the valid time intervals of two or more (base or materialized view) tables overlap, then they can be safely combined to provide a consistent answer a query. Tracking valid time gives the system greater flexibility in answering queries using materialized views. For example, a user can ask for a consistent answer to a query that is current within the past five days. The system can also return the valid time of the answer.

A valid time is *conservative* if the start time is later than the true start time or the end time is earlier than the true end time, or both. When the valid times are conservative, the system will not combine tables and views, even though the the combination would provide a consistent answer. When this occurs, the system still produces a correct answer, but fewer plans are considered (i.e., it is safe to use conservative valid times).

The tracking of valid time for views varies depending on the view maintenance policy. Immediate refresh views, like base tables, are valid from their most recent change to the current time. Deferred maintenance views are valid starting (conservatively) from their refresh time (after the computation of snapshot views, the specified refresh time for IPDA/DPDA) until the first change that affects them. The start time for snapshot views can actually be set to the latest start time of all of its input tables because the view could not have changed if none of its inputs changed. For IPDA/DPDA, the start time can be the most recent time that appears in the delta table before the refresh time. If no such record exists, then the start time is the most recent start time of all its input tables. However, simply using the refresh time is conservative, and it is easier to implement.

The system conservatively maintains the end time by the following procedure. When a change is made to a base table, B, the end time of each materialized view with deferred maintenance, V, that depends on B is checked. If the end time of V is not set (i.e., V is still valid at the current time), then the end time is set to the current time. If the end time is already set, then nothing is done. For IPDA/DPDA views, when the apply process is performed, if any commit records with a timestamp later than the refresh time are available, then the end time is set to the earliest of these timestamps.

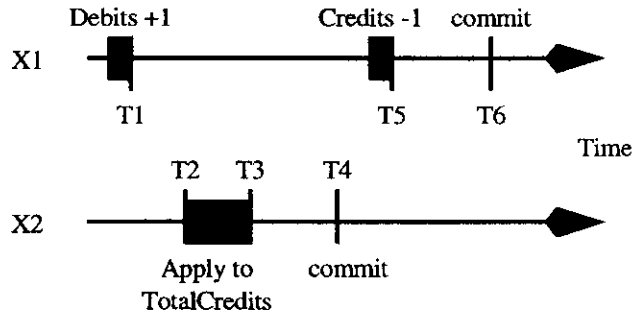


Figure 3: Debit Credit Workload

The timestamp used for valid times should be the commit time of the transaction, rather than the time of the update because the commit time is when the change officially occurred. Consider the following accounting database:

```

Debits(date, value)
Credits(date, value)
TotalCredits = SELECT sum(value) AS total FROM Credits

```

To keep the books balanced, every transaction places an equal total value into both Debits and Credits. Assume that the database is empty when we run the workload depicted in Figure 3. If the time of the last update to a relation is used as the relation's start time, the valid times would be:

Table	valid time	
	start	end
Debits	$t_1$	$\infty$
Credits	$t_5$	$\infty$
TotalCredits	$t_3$	$t_5$

So the query:

```

Q1 = SELECT 'debits', sum(value) FROM Debits UNION
      SELECT 'credits', sum(value) FROM Credits

```

can be rewritten as:

```
Q1 = SELECT 'debits', sum(value) FROM Debits UNION
      SELECT 'credits', total FROM TotalCredits
```

with a valid time of  $(t_3, t_5)$ . The result of the rewritten query is:

debits	1
credits	0

which is an inconsistent answer. If we instead used the commit time of each transaction, then the valid times become:

Table	valid time	
	start	end
Debits	$t_6$	$\infty$
Credits	$t_6$	$\infty$
TotalCredits	$t_4$	$t_5$

Therefore, `TotalCredits` can no longer be combined with `Debits`, the query will not be rewritten, and the correct result will be produced:

debits	1
credits	1

Actually, it is safe to use any time between the acquisition of the last lock and the release of the first lock because all the effects of the transaction will have the same timestamp, and no dependent transaction will use a timestamp in this range (see Lemma 3). By definition of two-phase locking, no dependent transaction could have completed all of its changes before the original commits.

The use of valid time in query processing has a major stumbling block: the state of valid times is dynamic. Query plans cannot be precompiled and used at a later point because the (base and materialized view) tables that were legally combined at compile-time might not be legal to combine at run-time. Moreover, while a query is running the valid times of some of its tables may change. For example, assume `Debits` and `TotalCredits` are mutually consistent when we start query  $Q$ . After we read `TotalCredits` and some of `Debits`, a record is added to `Debits` and committed. Subsequently,  $Q$  sees this new record, which means that  $Q$  returns an invalid answer. The problem is that even read-only transactions are serialized by their commit order.

One solution is to ensure the data that the query examines does not change while the query is running by placing table-level read locks on all the (base and materialized view) tables that the query will use before it accesses any of them. Unfortunately, this has the potential of locking much more data than necessary, which will reduce concurrency. However, if the queries usually access large portions of the tables, or if the workload is mostly read-only, then the loss of concurrency should be mitigated.

A second solution is to optimistically run the query, and double-check that the answer returned is still valid. If the valid times of the tables used to answer the query still overlap at the end of the query, then

we can assert that the answer is consistent. If the valid times no longer overlap, then the answer *might* be inconsistent. For instance, a record could be inserted into a table but not satisfy a predicate in the query. When the query result is in doubt, the system can return an error code, and the user can rerun the query if they need to ensure consistency. The downside is invalid answers might be returned to the user. However, in a read-mostly environment, few in-doubt results will be produced.

An improvement to the above solutions is to avoid mixing tables that have a strong chance of becoming inconsistent with each other. In a system with frequent updates to base tables, the system can mix immediate refresh views with base tables, and it can mix all types of deferred views (snapshot, IPDA, DPDA), but it should not mix frequently updated base tables and their immediate refresh views with deferred views.

## 7 Summary

We introduced the Immediate Propagate Deferred Apply (IPDA) materialized view policy, and described its relationship to other maintenance policies. IPDA improves update concurrency by reducing the conflicts induced by immediate refresh, but still incrementally maintains views. IPDA supports point-in-time refresh, which enables a view to be made consistent with some time in the past. By independently rolling multiple views to the same point in time, the views become mutually consistent, which means that the views can be combined to answer a query consistently.

We argued that the naive implementation of IPDA is riddled with consistency problems. We provided a corrected version of IPDA, and proved that it produces consistent answers. We described the net-effect of a delta table, and how it can be used to avoid replaying history and to condense the the delta table. We discussed how the valid-times of tables can be used to allow base tables and materialized views to be combined consistently.

In summary, we presented a detailed discussion of the issues and benefits of IPDA that we discovered during our prototype implementation of IPDA in DB2.

## References

- [1] R. Bello, K. Dias, A. Downing, J. Feenan, J. Finnerty, W. Norcott, H. Sun, A. Witkowski, and M. Ziauddin. Materialized views in Oracle. In *Proceedings of the International Conference of Very Large Data Bases*, pages 659–664, 1998.
- [2] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently updating materialized views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 61–71, 1986.
- [3] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 469–480, 1996.

- [4] L. S. Colby, A. Kawaguchi, D. F. Lieuwen, I. S. Mumick, and K. A. Ross. Supporting multiple view maintenance policies. In J. Peckham, editor, *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 405–416. ACM Press, 1997.
- [5] A. Gupta, H. V. Jagadish, and I. S. Mumick. Data integration using self-maintainable views. In *EDBT*, pages 140–144, 1996.
- [6] A. Gupta and I. S. Mumick. Maintaining views incrementally. *Bulletin of the IEEE Technical Committee on Data Engineering*, 18(2):3–19, 1995.
- [7] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 157–167, 1993.
- [8] I. S. Mumick, D. Quass, and B. S. Mumick. Maintenance of data cubes and summary tables in a warehouse. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 100–111, 1997.
- [9] D. Quass, A. Gupta, I. S. Mumick, and J. Widom. Making views self-maintainable for data warehousing. In *International Conference on Distributed Information Systems*, pages 158–169, 1996.
- [10] D. Quass and J. Widom. On-line warehouse view maintenance. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 393–404, 1997.
- [11] K. Salem, K. S. Beyer, B. Lindsay, and R. Cochrane. How to roll a join: Asynchronous incremental view maintenance. Submitted to SIGMOD'00.
- [12] O. Shmueli and A. Itai. Maintenance of views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 240–255, 1984.
- [13] R. T. Snodgrass and I. Ahn. A taxonomy of time in databases. In S. B. Navathe, editor, *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data, Austin, Texas, May 28-31, 1985*, pages 236–246. ACM Press, 1985.
- [14] Y. Zhuge, H. G. Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 316–327, 1995.

