On-Line Warehouse View Maintenance*

Dallan Quass
Computer Science Department
Stanford University
quass@cs.stanford.edu

Jennifer Widom
Computer Science Department
Stanford University
widom@cs.stanford.edu

Abstract

Data warehouses store materialized views over base data from external sources. Clients typically perform complex read-only queries on the views. The views are refreshed periodically by *maintenance transactions*, which propagate large batch updates from the base tables. In current warehousing systems, maintenance transactions usually are isolated from client read activity, limiting availability and/or size of the warehouse. We describe an algorithm called *2VNL* that allows warehouse maintenance transactions to run concurrently with readers. By logically maintaining two versions of the database, no locking is required and serializability is guaranteed. We present our algorithm, explain its relationship to other multi-version concurrency control algorithms, and describe how it can be implemented on top of a conventional relational DBMS using a query rewrite approach.

1 Introduction

Data warehouses collect information from one or more data sources and integrate it into a single database where it can be queried by clients (*readers*) of the warehouse. The relations stored at the warehouse represent materialized views over the data at the sources [LW95]. Because data warehouses often are used primarily for *decision support*, queries at the warehouse tend to be long and complex. Thus, a warehouse may contain many materialized views in order to speed up query processing [HRU96].

As changes are made to the data at the sources, the views at the warehouse become out of date. In order to make the views consistent again with the source data, the views can be *incrementally maintained* [GL95] by propagating changes from the source data to the warehouse views. In current commercial warehousing systems, usually changes to the source data are queued and propagated periodically (*e.g.*, once a day) to the warehouse views in a large batch update transaction, called a *maintenance transaction*. The periodic maintenance transaction is typically the only transaction to update the warehouse views—all other transactions performed at the warehouse are read-only queries.

An important problem in data warehousing is how to execute queries and the periodic maintenance transaction so that they do not block one another, especially because both queries and maintenance transactions can be long running. One approach to avoid blocking is to violate serializability and allow readers to see an inconsistent database state. However, often such inconsistency is not acceptable. In fact, readers may want to see data that is consistent across a sequence of queries executed over a period of several minutes or hours while analyzing the data. During analysis it would be unacceptable to have the results change from query to query. We term such a long-running sequence of queries a *reader session*.

How can readers be guaranteed to read a consistent database state without blocking when the maintenance transaction is running? Conventional two-phase locking algorithms can't be used because they require readers to block if they attempt to read a data item that is modified by an active (uncommitted) maintenance transaction, and a maintenance transaction will block if it attempts to modify a data item that is read by an active reader. Since in data warehouses both readers and maintenance transactions often access significant portions of the database, blocking would occur frequently.

1.1 Current approach

The approach most commonly used in commercial warehousing systems for guaranteeing consistency without blocking is to maintain the warehouse at night, during which time the warehouse is unavailable to readers. Since readers and the maintenance transaction never execute at the same time: (i) they execute without blocking, (ii) readers are guaranteed to read a consistent database state, and (iii) neither readers nor the maintenance transaction need to place any locks.

This current approach is illustrated in Figure 1. The figure shows three maintenance transactions executing on three different nights. The reader sessions take place only during the day when the maintenance transactions are not running. Unfortunately, there are two major problems with this approach:

- As corporations become globalized there is no longer any nighttime common to all corporate sites during which it is convenient to make the warehouse unavailable to readers.

*This work was supported by Rome Laboratories under Air Force Contract F30602-94-C-023 and by equipment grants from Digital and IBM Corporations.

- Since the maintenance transaction must be complete by the next morning, the time available for view maintenance can be a limiting factor in the number and size of views that can be materialized at the warehouse.

1.2 Our approach

In this paper we propose an algorithm that: (i) allows readers and the maintenance transaction to execute concurrently without blocking, (ii) allows readers to see a consistent database state throughout an entire session (*i.e.*, readers and the maintenance transaction are serializable), and (iii) allows readers and the maintenance transaction to execute without the overhead of placing locks. Using our algorithm it is possible to make a warehouse available to readers 24 hours a day. The algorithm is a type of multi-version concurrency control algorithm that is especially suited to view maintenance in a data warehousing environment. Data warehousing environments are distinct from typical database environments because at most one update (maintenance) transaction is active at a time. This property allows us to develop a new algorithm that is better tuned to data warehousing, and is simpler and easier to implement than existing multi-version algorithms. We call our algorithm *two-version no locking* (2VNL) because up to two versions of each tuple are available simultaneously, and readers and the maintenance transaction do not need to place locks. The algorithm has the following advantages in data warehousing environments over other multi-version concurrency control algorithms:

- Little extra storage is required to maintain the extra version information.
- The overhead in terms of additional I/O's required for readers and the maintenance transaction is small.
- The overhead of locking can be eliminated for both readers and the maintenance transaction.
- 2VNL can be implemented on top of existing database management systems through query rewrite, without the need to modify the DBMS's existing concurrency control or storage systems.

1.3 Paper outline

The remainder of the paper proceeds as follows. Section 2 introduces a running example and motivates the 2VNL algorithm. Section 3 specifies the details of the algorithm. Section 4 shows how the 2VNL algorithm can be implemented on top of current DBMS's through query rewrite. Section 5 explains how to extend the 2VNL algorithm to the general case of nVNL ($n \geq 2$). In Section 6 we compare the 2VNL algorithm to *two-version two-phase locking* (2V2PL) and *multi-version two-phase locking* (MV2PL) algorithms [BHG87] and discuss the advantages of 2VNL for data warehousing environments. Conclusions and areas for future research are presented in Section 7.

2 Example and Motivation

EXAMPLE 2.1 Consider a warehouse of sales data for a chain of sporting goods stores. Let the warehouse contain the following relation (materialized view) aggregating total daily sales by city and product line.

DailySales(tt city, state, product_line, date,
total_sales)

The DailySales relation is an example of a *summary table*, because it summarizes (aggregates) the base sales data. Summary tables are used commonly in data warehouses to speed up the evaluation of aggregate queries [HRU96].

Suppose that an analyst wanted to find the total sales made by stores in each city. The analyst would issue the following query on the DailySales relation:

```
SELECT city, state, SUM(total_sales)
FROM    DailySales
GROUP BY city, state
```

A common subsequent action for the analyst would be to "drill down" in some particular area in order to get more detail. For example, if sales in San Jose, California weren't as high as the analyst thought they ought to be, the analyst could get a breakdown of sales made in San Jose in each of the product lines by issuing the following query:

```
SELECT product_line, SUM(total_sales)
FROM    DailySales
WHERE city = "San Jose" and state = "CA"
GROUP BY product_line
```

It is important that the analyst see a consistent state of the database across both queries. For example, it would be disconcerting if the sum of the San Jose sales broken down by product line that was returned by the second query didn't add up to the overall San Jose sales returned by the first query. Therefore, if a maintenance transaction needs to update the DailySales relation to include the effect of the current day's sales while the data is being analyzed, then either the maintenance transaction should block until the analyst's session is over, or the analyst should be able to continue reading the state of the database as it was before the maintenance transaction took effect; *i.e.*, the session and the maintenance transaction should be serializable.

We will revisit the DailySales example throughout the paper. □

2.1 Transactions and sessions under 2VNL

As mentioned in the introduction, using conventional locking to achieve serializability is undesirable. Because reader sessions and maintenance transactions tend to be long and complex, blocking could introduce significant delays, while at the same time locking introduces significant overhead. The current solution used by most commercial warehousing systems is to only execute maintenance transactions at night, when the warehouse is unavailable to readers. This scenario was illustrated in Figure 1.

Figure 2 shows a possible warehouse operation when the 2VNL algorithm is applied. (Ignore the "Database Versions" information for now.) Note that the maintenance transaction can execute concurrently with reader transactions, which means that maintenance transactions can be longer and/or more frequent, and readers need not be disabled during warehouse modifications.[1]

A reader session accesses the state of the database that was current as of the commit of the most recent previous maintenance transaction. Call this transaction t_1. The reader can continue to read this state throughout a subsequent maintenance transaction, t_2, until t_2 commits and

[1] Figure 2 illustrates one, perhaps extreme, pattern of maintenance transactions allowable using 2VNL: very long maintenance transactions with only short gaps between them. In practice, maintenance transactions might be shorter and gaps might be longer. The only case in which 2VNL is inappropriate is when both maintenance transactions and gaps are very short, an unlikely scenario in a data warehouse.

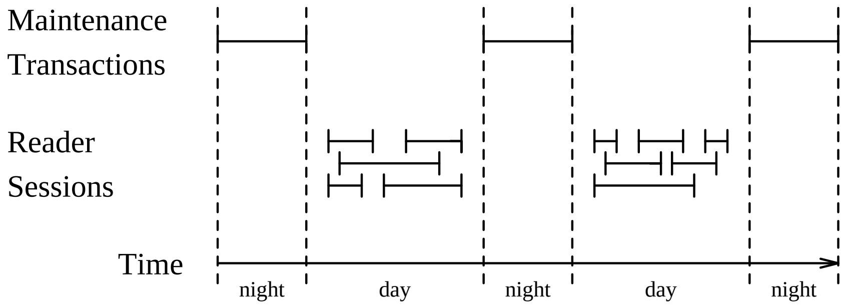

Figure 1: Current approach to warehouse querying and maintenance

another maintenance transaction, t_3, begins. Once t_3 begins, since only up to two versions of a tuple are available in the database, the reader can no longer be guaranteed to read a consistent database state. We then say that the session has "expired," and the reader is notified to begin a new session if consistency is desired.

Figure 2 illustrates a policy where a maintenance transaction is started each day at 9am. All updates sent to the warehouse during that day are applied within the scope of this maintenance transaction. The transaction is committed at 8am the following morning. A session beginning after 8am will therefore see the effects of that maintenance transaction, and is guaranteed to access a consistent database state until 9am the following morning, at which point the session expires and a new session must be begun.

Executing a single maintenance transaction once a day is just one policy that can be used with the 2VNL algorithm. A potential problem with this policy (or any other policy with small gaps between maintenance transactions) is that sessions beginning just before 8am expire very quickly, at 9am the same day. This problem is alleviated somewhat by executing maintenance transactions on a regular schedule, since readers can anticipate when their sessions will expire, but alternative solutions are also possible. One possibility is to commit the maintenance transaction only when no reader sessions are active. The disadvantage of this approach is that it is possible for readers to "starve" the maintenance transaction, *i.e.*, the maintenance transaction might wait a very long time to commit, but the advantage is that reader sessions never expire. Another possibility is to extend the 2VNL algorithm to the more general nVNL ($n \geq 2$) case. The nVNL algorithm is considered in Section 5.

2.2 Intuition for 2VNL

We now give the intuition behind the 2VNL algorithm. We assume that an external protocol limits maintenance transactions to execute one at a time. We also assume that readers and the maintenance transaction do not place any locks, or that if the maintenance transaction does place locks, readers ignore the locks. In SQL92 [MS93], readers can be instructed to ignore write locks by setting the transaction isolation level to "read uncommitted," and several commercial DBMS's, such as Informix, support this capability.

The additional concurrency in Figure 2 is achieved by making two versions of the database logically available simultaneously. The method for making two versions available simultaneously is explained in Section 3. Even though a reader and the maintenance transaction may access the same tuples, they do not interfere with each other because conceptually each operates on a different database version. For this discussion we classify a database version as either a *future version*, a *current version*, or a *previous version*. At a given point in time the versions represented in the database are either a future version and a current version (when there is an active maintenance transaction), or a current version and a previous version (when there is not an active maintenance transaction).

Maintenance transactions always operate on a future version. A reader is associated with the version that is current at the beginning of the session, and continues to read that version even if it becomes a previous version, until the version expires. Figure 2 illustrates when the various versions are available to readers. For example, during the first maintenance transaction in Figure 2 a current version (labeled version 1) is available to readers while the maintenance transaction creates a future version. When the first maintenance transaction commits, the current version (version 1) becomes a previous version and the future version that was created by the maintenance transaction becomes the new current version (version 2). Readers can continue to read version 1 even though it is now a previous version. When the second maintenance transaction begins, the previous version (version 1) *expires*; that is, it becomes unavailable for reading, and readers still reading this version are notified to begin a new session. The second maintenance transaction creates a new future version, with the current version (version 2) still available to readers. It is important to note that the process of switching versions is logical: the tuples in the database are not modified in order to switch.

3 2VNL Algorithm

In this section we specify the 2VNL algorithm. First we describe the general algorithm, then we describe each component in detail.

As alluded to in Figure 2, each version of the database is associated with a unique version number. A global variable, *currentVN*, records the current version number. Variable *currentVN* is 1 initially. In addition, a global flag *maintenanceActive* records whether a maintenance transaction is currently active. (We assume a simple latching mechanism is used to read and update these global variables, or they can be implemented as a record in the DBMS as described in Section 4.)

Each tuple in the database is extended to include: (i) a *tupleVN* attribute that records the version of the database when the tuple was last modified by a maintenance transaction, (ii) an *operation* attribute that records the operation $\in$ {*insert*, *update*, *delete*} last performed on the tuple, and (iii) a set of *pre-update attributes* that contain the previous values of the updatable attributes of the tuple (*i.e.*, those attributes that could be changed by an *update* oper-

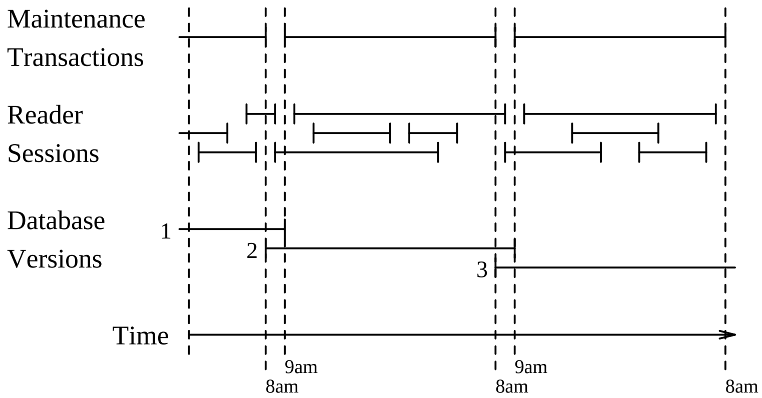

Figure 2: Warehouse querying and maintenance scenario with 2VNL

ation in the maintenance transaction). In the worst case, storing the previous values of updatable attributes requires approximately doubling the size of the warehouse, but as we will see in Section 3.1, for summary tables the overhead often is much less. By recording the *tupleVN*, *operation*, and previous values of updatable attributes, each tuple contains sufficient information so that two versions of the tuple are available: the version of the tuple that was current as of database version *tupleVN*, and the version of the tuple that was current in database version *tupleVN*−1.

When a maintenance transaction begins, it reads the global *currentVN* variable and sets a local *maintenanceVN* variable to *currentVN*+1. In addition, it sets the global *maintenanceActive* flag to true.

A maintenance transaction always reads the latest version of tuples in the database. When the maintenance transaction modifies a tuple in the database, the tuple must be modified in such a way that readers can later read the pre-update version of the tuple. The maintenance transaction therefore updates the *tupleVN* and *operation* attributes in the tuple with its *maintenanceVN* and the logical operation (*insert*, *update*, or *delete*) that is performed on the tuple. In addition, the previous values of updatable attributes are saved in the pre-update tuple attributes. Note that in order not to lose information about the previous state of a tuple when it is modified by a maintenance transaction, the *physical operation* performed on the tuple is not always the same as the *logical operation* specified in the maintenance transaction. For example, a logical tuple delete may be translated to a physical tuple update in order that the previous version of the tuple still be available (see Section 3.3).

At commit of the maintenance transaction it updates the global *currentVN* with its *maintenanceVN* and sets the *maintenanceActive* flag to false, indicating that maintenance is complete and there is a new current version of the database.

When a reader session begins, it reads *currentVN* and copies it into its local *sessionVN* variable. This is the version of the database that the reader will access throughout the session. When a reader reads a tuple, it reads the most recent version of the tuple that is $\leq$ *sessionVN*. If the tuple has not been modified since the session began, then the current version of the tuple is read. On the other hand, if the tuple has been modified by an uncommitted maintenance transaction or by a maintenance transaction that committed after the start of the reader session, then the pre-update version of the tuple is read. Thus readers are guaranteed to read a consistent database state without placing read locks. The maintenance transaction need not place write locks on the tuples it modifies, or if it does, the reader can (and should) ignore the write locks in order to continue without blocking.

A reader must be able to detect if its session has expired, which can be done in one of two ways. A reader can determine that its session has expired when it attempts to read a tuple that has been modified by more than one maintenance transaction since the reader session began. In this case the proper tuple version cannot be read because the current and pre-update tuple versions are maintained only for the most recent update. Alternatively, a reader can perform a more "global" but pessimistic check, in which the reader's *sessionVN* is compared with the global *currentVN*. Both approaches are discussed in more detail in Sections 3.2 and 4.1 below.

3.1 Modifying the relation schema

In this section we explain how a relation schema is modified to represent two versions of each tuple.

Each of the relations at the warehouse needs to be extended with additional attributes. Let $\mathcal{A} = \{A_1, A_2, \ldots, A_n\}$ be the initial set of attributes for a relation R, and let $\mathcal{A}' = \{A_1, A_2, \ldots, A_k\}$ be the subset of attributes in $\mathcal{A}$ that are updatable. Then the extended schema of R is $\{$*tupleVN*, *operation*, $A_1, A_2, \ldots, A_n,\ A_1^p, A_2^p, \ldots, A_k^p\ \}$, where A_i^p is used to denote the pre-update attribute corresponding to updatable attribute A_i. Attribute *tupleVN* contains the *maintenanceVN* of the maintenance transaction that most recently modified the tuple, *operation* contains the logical operation performed by that maintenance transaction, attributes $\{A_1, A_2, \ldots, A_n\}$ contain the current values of the tuple attributes, and pre-update attributes $\{A_1^p, A_2^p, \ldots, A_k^p\}$ contain the values of the updatable attributes before being updated by the maintenance transaction. (In the case of insert operations the pre-update attributes are null, and in the case of delete operations they contain the pre-delete values.)

In the worst case, when every attribute is updatable, representing two versions of each tuple requires approximately doubling the storage space of the warehouse. However, it is often the case in data warehouses that many attributes are not updatable. For example, data warehouses often contain many summary tables, which can be thought of as `select-from-where-groupby` aggregate views. Although tuples can be inserted into or removed from a summary table by a maintenance transaction, the values of the group-by attributes are never updated. Only the attributes represent-

tupleVN	*operation*	city	state	product_line	date	total_sales	*pre*_total_sales
4	1	20	2	12	4	4	4

Figure 3: Modified schema for `DailySales` relation

tupleVN	*operation*	city	state	product_line	date	total_sales	*pre*_total_sales
3	insert	San Jose	CA	golf equip	10/14/96	10,000	null
4	insert	San Jose	CA	golf equip	10/15/96	1,500	null
4	update	Berkeley	CA	racquetball	10/14/96	12,000	10,000
4	delete	Novato	CA	rollerblades	10/13/96	8,000	8,000

Figure 4: Example `DailySales` relation with modified schema

ing the results of aggregate functions are updatable. Hence, for summary tables the storage overhead required by the 2VNL algorithm is small.

EXAMPLE 3.1 Given the `DailySales` relation of Example 2.1, Figure 3 shows the relation schema after being extended with the additional attributes necessary for the 2VNL algorithm. In the figure the attribute lengths are shown under each attribute. Before modification, the `DailySales` relation required 42 bytes per tuple. After modification it requires 51 bytes, an increase of approximately 20%. □

3.2 Algorithm for readers

In this section we give the algorithm for extracting the correct version of a tuple so that a reader sees a consistent state of the database. The general idea is that by looking at *tupleVN*, the reader can tell whether it should read the current version of the tuple or the pre-update version of the tuple. By looking at *operation* and the current and pre-update attributes the proper state for either tuple version can be extracted.

A reader always reads the version of the tuple that was current during database version *sessionVN*, meaning the version that includes the effects of all maintenance transactions with $maintenanceVN \leq sessionVN$, and no other maintenance transactions. Recall that *tupleVN* contains the *maintenanceVN* of the last maintenance transaction to modify the tuple. There are three cases to consider:

1. $sessionVN \geq tupleVN$: Read the current version of the tuple.
2. $sessionVN = tupleVN - 1$: Read the pre-update version of the tuple.
3. $sessionVN < tupleVN - 1$: The session has expired.

The current version of the tuple is the state that was current as of database version *tupleVN*. The pre-update version of the tuple is the state that was current in database version $tupleVN-1$. The reason in case (3) that the session has expired is that there is no way of determining the tuple's state at database version $tupleVN-2$ or earlier, since only two states of the tuple are available. In this case the reader can be notified to begin a new session.

Table 1 specifies how to extract the current or pre-update version of the tuple, depending upon the *operation*. For example, in the case where the pre-update version of the tuple is to be read and the *operation* = insert, the tuple should be ignored. Note that when the table specifies to read pre-update attribute values, the current attribute values are read for non-updatable attributes since they cannot change.

EXAMPLE 3.2 Assume that our example `DailySales` relation, extended as described in Section 3.1, contains tuples as illustrated in Figure 4. If a reader with $sessionVN = 3$ reads the relation, the following tuples would be returned according to Table 1.

city	state	product_line	date	total_sales
San Jose	CA	golf equip	10/14/96	10,000
Berkeley	CA	racquetball	10/14/96	10,000
Novato	CA	rollerblades	10/13/96	8,000

□

In Section 4.1 we will show how the decision procedure of Table 1 can be implemented in SQL, so that the correct version of each tuple can be extracted through a query rewrite mechanism.

3.3 Algorithm for maintenance transactions

When a maintenance transaction reads a tuple, it always reads the current version. Thus, it always follows the first line of Table 1 for tuple reads.

When a maintenance transaction inserts, deletes, or updates a tuple, several actions need to take place so that both the current and pre-update tuple versions are maintained in the tuple:

- In some cases the current attribute values are moved to the pre-update attribute values so that the pre-update version of the tuple is preserved.
- The current attribute values are set to the values specified by the maintenance operation.
- *tupleVN* is set to *maintenanceVN*.
- *operation* is set to the logical operation (insert, delete, or update) that is performed by the maintenance transaction on the tuple. Note that the logical operation performed by the maintenance transaction may not be the same as the physical operation effected on the tuple. For example, when the logical operation is deletion, the tuple usually is not physically deleted from the database because the pre-update version of the tuple might be needed by readers. Once logically-deleted tuples are no longer needed by readers, they can be garbage collected by periodically running a process to physically delete them. We plan to examine garbage collection in more detail in future work.

In addition, the operation recorded in *operation* needs to represent the *net effect* [SP89] of all operations performed by the maintenance transaction on the tuple.

	operation Insert	Update	Delete
Current Version	read current attribute values	read current attribute values	ignore tuple
Pre-Update Version	ignore tuple	read pre-update attribute values	read pre-update attribute values

Table 1: Decision table for extracting the current or pre-update tuple version

	Previous *operation*: Insert	Update	Delete
$tupleVN < maintenanceVN$	impossible	impossible	Update tuple: $PV \leftarrow$nulls, $CV \leftarrow MV$, $tupleVN \leftarrow maintenanceVN$, $operation \leftarrow$insert
$tupleVN = maintenanceVN$	impossible	impossible	Update tuple: $CV \leftarrow MV$, $operation \leftarrow$update
No Conflicting Tuple	Insert tuple: $PV \leftarrow$nulls, $CV \leftarrow MV$, $tupleVN \leftarrow maintenanceVN$, $operation \leftarrow$insert		

Table 2: Decision table for insert maintenance operation

	Previous *operation*: Insert	Update	Delete
$tupleVN < maintenanceVN$	Update tuple: $PV \leftarrow CV$, $CV \leftarrow MV$, $tupleVN \leftarrow maintenanceVN$, $operation \leftarrow$update	Update tuple: $PV \leftarrow CV$, $CV \leftarrow MV$, $tupleVN \leftarrow maintenanceVN$, $operation \leftarrow$update	impossible
$tupleVN = maintenanceVN$	Update tuple: $CV \leftarrow MV$	Update tuple: $CV \leftarrow MV$	impossible

Table 3: Decision table for update maintenance operation

	Previous *operation*: Insert	Update	Delete
$tupleVN < maintenanceVN$	Update tuple: $PV \leftarrow CV$, $tupleVN \leftarrow maintenanceVN$, $operation \leftarrow$delete	Update tuple: $PV \leftarrow CV$, $tupleVN \leftarrow maintenanceVN$, $operation \leftarrow$delete	impossible
$tupleVN = maintenanceVN$	Delete tuple	Update tuple: $operation \leftarrow$delete	impossible

Table 4: Decision table for delete maintenance operation

	city	state	product_line	date	total_sales
insert:	San Jose	CA	golf equip	10/16/96	11,000
insert:	Novato	CA	rollerblades	10/13/96	6,000
update:	San Jose	CA	golf equip	10/14/96	10,000 → 10,200
delete:	Berkeley	CA	racquetball	10/14/96	12,000

Figure 5: Example maintenance transaction

tupleVN	*operation*	city	state	product_line	date	total_sales	*pre*_total_sales
5	update	San Jose	CA	golf equip	10/14/96	10,200	10,000
4	insert	San Jose	CA	golf equip	10/15/96	1,500	null
5	delete	Berkeley	CA	racquetball	10/14/96	12,000	12,000
5	insert	Novato	CA	rollerblades	10/13/96	6,000	null
5	insert	San Jose	CA	golf equip	10/16/96	11,000	null

Figure 6: Result of DailySales after maintenance transaction

For example, if a maintenance transaction inserts a tuple and then updates the same tuple in the same transaction, the net effect is still an insert. If *operation* were incorrectly set to update, readers looking for the pre-update version of the tuple would try to read the pre-update attribute values, instead of correctly ignoring the tuple.

- The appropriate physical operation is performed on the tuple.

The exact actions to perform on the tuple depend upon the maintenance operation, the tuple's *tupleVN* value, and the tuple's *operation* value. Tables 2, 3, and 4 give the actions to perform for an insertion, update, and deletion maintenance operation respectively. Given *maintenanceVN* for the maintenance transaction and the tuple's existing values for *tupleVN* and *operation*, the tables show the correct physical operation to perform on the tuple so that the current and pre-update tuple versions are preserved. In the tables,

- CV denotes the current attribute values of the tuple,
- PV denotes the pre-update attribute values of the tuple, and
- MV denotes the attribute values specified in the maintenance operation (if the maintenance operation is an insert or an update).

Thus, the expression "$PV \leftarrow CV$" means set the pre-update attribute values of the tuple equal to the corresponding current attribute values, and "$CV \leftarrow MV$" means to set the current attribute values of the tuple equal to the attribute values specified in the maintenance operation.

Note that since only one maintenance transaction executes at a time, we know that $tupleVN \leq maintenanceVN$. The first row in each table specifies the actions to take when the tuple's version number is less than the maintenance transaction's version number. When $tupleVN = maintenanceVN$, the tuple has been modified previously by the same maintenance transaction. The actions to take in this case are specified in the second row in each table. As mentioned previously, the *operation* assigned in the second row represents the net effect of all operations performed on the tuple by the maintenance transaction.

In Table 2 for insertions, the first two rows are for the case when tuples have unique keys and a tuple having the same key as the tuple being inserted by the maintenance transaction is found in the database. (This case can occur only when a tuple with the same key was previously deleted.) The third row in Table 2 describes the actions to take in the more common case when a conflicting tuple is not found in the database. For tuples that do not have unique keys, the actions in the third row are always followed.

Some of the table cells simply specify "impossible." These cells represent sequences of operations that are not possible in a valid transaction. For example, it is not possible to update or delete an already-deleted tuple (Tables 3 and 4), and if tuples have unique keys then it is not possible to insert a tuple with the same key value as a previously inserted or updated tuple (Table 2). We can however insert a tuple with the same key as a previously deleted tuple (Table 2), and the net effect of a delete and insert in the same maintenance transaction is an update.

EXAMPLE 3.3 Assume again that our `DailySales` relation contains tuples as illustrated in Figure 4. As is always the case with summary tables, the key of the relation is the set of group-by attributes, which in our case are `city`, `state`, `product_line`, and `date`. Suppose we execute a maintenance transaction with a *maintenanceVN* of 5 containing the operations shown in Figure 5. Executing the maintenance transaction results in the extended `DailySales` relation having the tuples shown in Figure 6. □

In Section 4.2 we will show how the decision procedures of Tables 2, 3, and 4 can be implemented in SQL, so that the current and pre-update versions of each tuple are preserved by a maintenance operation.

4 Implementing 2VNL

The 2VNL algorithm can be implemented in a data warehousing system either by modifying the internals of the system, or by using a *query rewrite* [Sto75,SJGP90] approach in which the system itself need not be modified at all. Certainly building 2VNL into the system is likely to yield better performance, but in many cases it is impractical or impossible to modify the internals of an existing DBMS. A useful property of 2VNL is that it can be implemented entirely outside of an existing DBMS by automatically modifying the relation schema as specified in Section 3.1 and rewriting the maintenance and query operations. We will specify the rewriting process by showing how to implement the decision tables of Sections 3.2 and 3.3 using SQL.

To implement 2VNL on top of an existing DBMS by query rewrite we require that the DBMS have the following two characteristics:

- During the time that a tuple is in the process of being modified, a latch (short-duration lock) is held on the tuple or the page to keep readers from accessing a partly-modified tuple. The latch is released as soon as the tuple has been modified, without waiting for the transaction to commit. A write lock is not obtained on the modified tuple, or if it is, the write lock is ignored by readers. In SQL-92 it is possible to set the transaction isolation level to "read uncommitted" to tell readers to ignore write locks, and at least some commercial DBMS's support this capability.

- When a physical tuple update is performed, the update is performed in place so that the new state of the tuple (containing information regarding the current and pre-update tuple versions) replaces the old tuple on the page. Performing updates in place makes it impossible for a reader scanning through a relation to read two different physical records for the same tuple. Most DBMS's perform updates in place. If updates are not performed in place by the DBMS, then an update must be issued instead as a deletion and an insertion, which may result in key values no longer being unique since several physical records may have the same key value but different versions. If we chose instead to build the algorithm into a DBMS, which we plan to explore as future work, the requirement to update in place would not be necessary.

In order to implement the global variables *currentVN* and *maintenanceActive* using the DBMS, they can be stored in a single-tuple, two-attribute `Version` relation that is read by readers and updated by maintenance transactions. At the beginning of a maintenance transaction the *maintenanceActive* flag in the tuple is set to true. Just before the commit of the maintenance transaction *maintenanceVN* is written to the tuple as the new value of *currentVN* and *maintenanceActive* is set to false. Note that if after writing *maintenanceVN* into *currentVN* the maintenance transaction aborts, readers who read the updated *currentVN* can see an inconsistent database state while the maintenance transaction is backing out. A solution to this problem would be to update *currentVN* in a separate transaction that runs just after the maintenance transaction commits.

4.1 Query rewrite for readers

We now explain how to rewrite reader queries to access the correct tuple version according to the decision table of Section 3.2. In the rewriting, the SQL-92 `CASE` expression [MS93] is used to access the current or pre-update attributes as appropriate. Any time an updatable attribute is referenced in a query it is replaced with a `CASE` expression that returns the current or pre-update attribute value depending upon the tuple's *tupleVN* and the reader's *sessionVN*. Additionally, a condition is added to the `where` clause so that the appropriate tuples are ignored. We illustrate the rewriting with an example; the general case follows directly.

EXAMPLE 4.1 Returning again to our `DailySales` example relation, suppose that an analyst wanted to find the total sales made by stores in each city. The following query would be issued:

```
SELECT city, state, SUM(total_sales)
FROM   DailySales
GROUP BY city, state
```

Since `total_sales` is the only updatable attribute, after rewriting the query is translated to the following. We use :*sessionVN* as a placeholder for the reader's *sessionVN* value.

```
SELECT city, state,
       SUM(CASE WHEN :sessionVN ≥ tupleVN
            THEN total_sales ELSE pre_total_sales END)
FROM   DailySales
WHERE  (:sessionVN ≥ tupleVN AND operation <> "delete")
  OR   (:sessionVN < tupleVN AND operation <> "insert")
GROUP BY city, state
```

□

Readers also need to detect when they have expired. As discussed earlier, one approach is to detect whenever a tuple is read with $sessionVN < tupleVN-1$ and raise an exception, but this approach cannot always be implemented by query rewrite. Alternatively, the reader can determine whether it *may* have read such tuples by checking whether, since the reader started, a maintenance transaction has committed and another begun. This check is performed by evaluating the following condition (which can be implemented by reading the single-tuple `Version` relation):

$(sessionVN = currentVN)$ or
$((sessionVN = currentVN - 1)$ and
$maintenanceActive = \text{false})$

If the condition returns false, then the session is expired.

4.2 Modifying maintenance transactions

Insert, delete, and update statements issued by a maintenance transaction are rewritten similarly to queries issued by readers. Each type of statement is considered separately below.

Note that in the decision tables for insert and delete operations (Tables 2 and 4 respectively), logical insertions and deletions sometimes translate to physical insertions and deletions respectively, and sometimes translate to physical tuple updates. Furthermore, in the decision table for update operations (Table 3), sometimes the current attribute values are preserved by copying into the pre-update attributes, and sometimes not. It is thus not possible to rewrite an SQL insert, update, or delete statement into a single corresponding statement. Either the statements can be rewritten into two corresponding statements, one for each type of physical operation that might be performed, or cursors can be used so that the decision of which physical operation to perform can be made on a tuple by tuple basis. We will explain the latter approach since cursors are likely to be used in the maintenance transaction even before the rewriting.

4.2.1 Insert statement

Following the third row of Table 2, an insert statement must be modified to add values for *tupleVN* and *operation* (setting them to *maintenanceVN* and "insert" respectively), and to set the pre-update attribute values to null. If the relation has no unique key, then these are the only changes. When the relation has a unique key, it is possible to encounter a key conflict upon insertion if the tuple was previously deleted by the same maintenance transaction, or if it was logically deleted earlier but not garbage collected. In the case of a key conflict, instead of inserting the new tuple, the existing tuple must be updated to reflect the values of the logically-inserted tuple. We illustrate the rewriting of an insert statement by an example; the general case follows directly.

EXAMPLE 4.2 Consider insertions into `DailySales`, which has as a unique key ⟨`city`, `state`, `product_line`, `date`⟩. The following pseudocode describes the insertion process. For each tuple, first the insert is attempted. If a unique key conflict occurs, the conflicting tuple is updated instead. Note that the second `select` statement in the pseudocode always returns a single tuple r since it selects on the key.

```
For each tuple t to insert
    INSERT INTO DailySales VALUES
                                        % line 3 in Table 2
      (:maintenanceVN, "insert", t.city, t.state,
       t.product_line, t.date, t.total_sales, null)
    If insert failed due to a unique key conflict,
        Let r =               % r has same key value as t
            (SELECT *
            FROM DailySales
            WHERE city = t.city AND state = t.state
              AND  product_line = t.product_line
              AND  date = t.date)
        If t.tupleVN < :maintenanceVN,
                                        % line 1 in Table 2
          Update r
               set r.pre_total_sales = null
               set r.total_sales = t.total_sales
               set r.tupleVN = :maintenanceVN
               set r.operation = "insert"
        Else                            % line 2 in Table 2
          Update r
               set r.total_sales = t.total_sales
               set r.operation = "update"
```

□

4.2.2 Update statement

The specific physical update operation corresponding to a logical update operation on a tuple depends upon whether the tuple has already been modified (inserted or updated) by the maintenance transaction. We can test this property for each tuple using a cursor approach, as shown below. If the tuple has not been modified by the maintenance transaction, then the existing values of updatable attributes are preserved by copying them into the pre-update attributes. If the tuple has been modified by the maintenance transaction, then the updatable attributes should not be copied. We illustrate the rewriting of an update statement with an example; again, the general case follows directly.

EXAMPLE 4.3 Suppose we were to add 1,000 to the `total_sales` of all tuples in our `DailySales` relation having a city of "San Jose" and a date of "10/13/96." The SQL update statement appears below.

```
UPDATE DailySales
        SET total_sales = total_sales + 1000
WHERE city = "San Jose" AND date = "10/13/96"
```

The following pseudocode illustrates how a cursor approach can be used to implement the above update statement.

```
For each tuple r in
        (SELECT *
        FROM DailySales
        WHERE city = "San Jose"
          AND  date = "10/13/96")
    If r.tupleVN < :maintenanceVN, % line 1 in Table 3
        Update r
            set r.pre_total_sales = r.total_sales
            set r.total_sales = r.total_sales + 1000
            set r.tupleVN = :maintenanceVN
            set r.operation = "update"
    Else                              % line 2 in Table 3
        Update r
            set r.total_sales = t.total_sales + 1000
```

□

4.2.3 Delete statement

A logical delete operation usually corresponds to a physical update operation, unless the tuple was previously inserted in the same transaction (in which case the tuple is physically deleted). Similar to an update statement, an SQL delete statement can be implemented using a cursor approach by testing for each tuple the value of *tupleVN* and *operation* and performing the appropriate action according to Table 4.

EXAMPLE 4.4 Suppose we were to delete all tuples in `DailySales` having a city of "San Jose" and a date of "10/13/96." The SQL delete statement appears below.

```
DELETE FROM DailySales
WHERE city = "San Jose" AND date = "10/13/96"
```

The following pseudocode illustrates how a cursor approach can be used to implement the above delete statement.

```
For each tuple r in
        (SELECT *
        FROM DailySales
        WHERE city = "San Jose"
          AND  date = "10/13/96")
    If r.tupleVN < :maintenanceVN, % line 1 in Table 4
        Update r
            set r.pre_total_sales = r.total_sales
            set r.tupleVN = :maintenanceVN
            set r.operation = "delete"
    Else                              % line 2 in Table 4
        If r.operation = "insert"
            Delete r
        Else
            Update r
                set r.operation = "delete"
```

□

4.3 2VNL and indexing

Traditional indexes can still be used under the 2VNL algorithm. Indexes on non-updatable attributes are not affected by the algorithm. Note especially that for the summary tables commonly found in data warehouses, indexes are usually built on the group-by attributes; since the group-by attributes are not updatable these indexes are not affected by the use of 2VNL.

Indexes on updatable attributes could also be built, but using 2VNL there are two physical attributes, a current and a pre-update attribute, for each logical attribute. Indexes could be built on both the current and pre-update attributes. However, in our rewrite approach, updatable attributes always appear inside `CASE` expressions in the rewritten queries. Thus, to use indexes the query optimizer would need to be able to use indexes on attributes appearing inside `CASE` expressions. It is doubtful that current query optimizers have this capability.

We plan to investigate indexing issues in the context of 2VNL in more depth as future work.

5 Extending 2VNL to n versions

As explained earlier, if during a reader session one maintenance transaction commits and another one starts, then the reader session can expire—the 2VNL algorithm only guarantees that a reader sees a consistent state as long as the reader overlaps at most one maintenance transaction. This overlap property is likely to hold for most reader sessions in data warehousing environments, since maintenance transactions tend to be long, and the "gaps" between them tend to be relatively long as well. Nevertheless, it is conceivable that expiration could become a problem, and in Section 2.1 we discussed several alternatives for avoiding it. In this section we elaborate on the nVNL solution.

The nVNL algorithm for a given $n > 2$ is the natural extension of the 2VNL algorithm to make n versions of the database available at the same time. Whereas the 2VNL algorithm guarantees that a reader sees a consistent state as long as it overlaps at most one maintenance transaction, the nVNL algorithm guarantees that a reader sees a consistent state as long as it overlaps at most $n-1$ maintenance transactions.

Increasing n allows us to increase the length of reader sessions that are guaranteed never to expire. For example, assuming i is the length of the minimum time interval between two maintenance transactions, and m is the length of the shortest maintenance transaction, then 2VNL guarantees that reader sessions lasting up to i will never expire. 3VNL, on the other hand, guarantees that reader sessions lasting up to $2i + m$ will never expire. In general, nVNL guarantees that reader sessions lasting up to $(n-1)*(i+m)-m$ never expire. Thus, n can be tuned for the expected pattern of reader sessions and maintenance transactions in a data warehouse in order to avoid expiration. Of course the higher n is, the more overhead we incur in storage and run-time costs.

Note that we are still assuming that maintenance transactions run one at a time: The purpose of nVNL is not to allow multiple concurrent writers, but rather to allow readers to (sequentially) overlap multiple maintenance transactions. Consequently, nVNL still differs from other multiversion algorithms in significant ways, as discussed earlier and in Section 6 below.

For nVNL the relation schema is modified as described in Section 3.1, except there are $n-1$ *tupleVN* attributes ($tupleVN_1, \ldots, tupleVN_{n-1}$), $n-1$ corresponding *operation* attributes, and $n-1$ corresponding sets of pre-update attributes. Attribute $tupleVN_i$ contains the *maintenanceVN* of the i-th most recent maintenance transaction to modify the tuple, $operation_i$ contains the logical operation performed by that modification, and the corresponding set of pre-update attributes contains the values of the updatable attributes before being modified.

As with 2VNL, we want a reader to see the version of a tuple that was current during database version *sessionVN*, meaning the version that includes the effects of all maintenance transactions with *maintenanceVN* $\leq$ *sessionVN*, and no other maintenance transactions. Recalling that $tupleVN_1$ denotes the largest (most recent) *maintenanceVN*, and $tupleVN_{n-1}$ denotes the smallest (least recent) *maintenanceVN*, the cases for a reader are:

1. $sessionVN \geq tupleVN_1$: Read the current version of the tuple.
2. $tupleVN_{n-1} - 1 \leq sessionVN < tupleVN_1$: Read the pre-update version of the tuple for the least $tupleVN_j >$ *sessionVN*.
3. $sessionVN < tupleVN_{n-1} - 1$: The session has expired.

To read the correct values for the tuple, Table 1 for 2VNL is used. For case (1) above, *operation* in Table 1 is $operation_1$ and the first row in the table is followed. For case (2), *operation* is $operation_j$, the second row is followed, and the j-th set of pre-update attribute values are used.

For maintenance transactions, we follow the same decision tables as for 2VNL (Tables 2–4), except instead of simply overwriting *tupleVN*, *operation*, and the pre-update attribute values, we "push back" attributes, eliminating the existing nth version in order to make room for a new 1st version. That is, we set $tupleVN_{i+1} \leftarrow tupleVN_i$ for $1 \leq i \leq n-2$, and similarly for the $operation_i$'s and the pre-update attributes. We then set $tupleVN_1$, $operation_1$, and the first set of pre-update attributes according to Tables 2–4. There are some exceptions in the tables, such as new inserts (where most attributes are set to null) and multiple updates in the same maintenance transaction (where rather than "pushing back" values we overwrite the most recent ones). Due to space constraints we do not enumerate all the cases here; the generalization from Tables 2–4 is straightforward.

EXAMPLE 5.1 Consider our example `DailySales` relation using nVNL with $n = 4$. We'll examine a tuple for golf equipment sales in San Jose, CA on 10/14/96. Suppose that a maintenance transaction with *maintenanceVN* = 3 inserts the tuple with `total_sales` = 10,000, then a maintenance transaction with *maintenanceVN* = 5 updates the `total_sales` to 10,200, then a maintenance transaction with *maintenanceVN* = 6 deletes the tuple. After the deletion, the tuple in the extended relational schema appears as shown in Figure 7.

Using the cases described above along with Table 1, we see that a reader with *sessionVN* ≥ 6 will ignore the tuple. A reader with *sessionVN* < 6 but ≥ 2 will read the pre-update version of the tuple for the least *tupleVN* $>$ *sessionVN*; that is, a reader with *sessionVN* $\in \{3, 4\}$ will see the logical tuple with `total_sales` = 10,000, and a reader with *sessionVN* = 2 will ignore the tuple. A reader with *sessionVN* < 2 will have expired. □

6 Relationship to 2V2PL and MV2PL algorithms

A large body of research has been performed on *two-version two-phase locking* (2V2PL) algorithms and *multi-version two-phase locking* (MV2PL) algorithms (also called *transient versioning* algorithms). In fact, several commercial DBMS's have implemented various forms of multi-version algorithms [BBG+95,BHG87]. Multi-version algorithms can be differentiated in several ways:

- how many previous versions are kept,
- how long previous versions are stored, and
- which versions are read by readers.

city	state	product_line	date	total_sales	*tupleVN*$_1$	*operation*$_1$	*pre*_total_sales$_1$	···
San Jose	CA	golf equip	10/14/96	10,200	6	delete	10,200	···

···	*tupleVN*$_2$	*operation*$_2$	*pre*_total_sales$_2$	*tupleVN*$_3$	*operation*$_3$	*pre*_total_sales$_3$
···	5	update	10,000	3	insert	null

Figure 7: Example 4VNL tuple

In 2V2PL algorithms [BHR80,SR81], when a writer modifies a tuple a new version of the tuple is created. While the writer transaction is active, concurrent readers read the previous tuple version. Because writers write a different version than the version read by readers, readers are never blocked by writers. Unlike our 2VNL (or nVNL) algorithm, however, 2V2PL algorithms delete the previous version of modified tuples when the writer transaction commits. Deleting the previous version of modified tuples causes readers to delay writers because the writer cannot commit until all readers that have read the previous version of modified tuples have committed—otherwise, repeatable reads would be sacrificed.

MV2PL algorithms [AS89,BC92a,BC92b,CFL+82,Wei87, MPL92,WYC93] guarantee that readers and writers never block each other.[2] MV2PL algorithms maintain sufficient previous versions of each tuple (or page, if versioning is at the page level) that readers are always guaranteed to have the correct tuple version available. Previous versions may be garbage collected when it is guaranteed that they are no longer needed by any reader. With some MV2PL algorithms, readers read the latest version of the tuple that is less than the reader's *begin-timestamp* [CFL+82]. These algorithms require maintaining possibly many previous versions of each tuple. By carefully choosing the previous tuple versions that are made available to readers, other MV2PL algorithms [MPL92,WYC93] guarantee that readers and writers never block each other with a maximum of three or four tuple versions. Note that in contrast, 2VNL requires only two versions of each tuple to be available. This reduction in the number of versions required is due to the fact that in data warehousing environments, maintenance transactions are known to run one at a time.

All MV2PL algorithms require overhead to access and maintain previous tuple versions. For example, the approach in [CFL+82] stores previous versions in a special "version pool" on disk, where they are chained together and to the current version. This means that readers might have to perform several I/O's to access the correct version of a tuple. Also, tuple writes involve an additional I/O for copying the existing current version to the version pool. The approach in [BC92b] improves on this design by reserving a portion of each data page for a version pool cache so that recent tuple versions are stored on the same page. But reserving a portion of each page for a version pool cache requires storage overhead, and the global version pool must still be accessed if the version pool cache on the page overflows. By storing all information in a single tuple, even in the case when most tuples in a relation are updated by maintenance transactions, the 2VNL algorithm requires very little overhead, especially in the case of summary tables where only a few of the attributes are updatable. Furthermore, in 2VNL additional I/O's for reading and modifying tuples are never required, although there could be more I/O's altogether in a scan, say, since fewer tuples fit on a page.

[MPL92] considers *incremental versioning*, storing only the values of changed attributes in previous tuple versions. They mention that in order to access the correct tuple version multiple incremental versions must be read and combined, but the algorithms for creating and combining the incremental versions are not given. More importantly, implementing the algorithm of [MPL92], as with all multiversion algorithms of which we are aware, requires significant changes to the DBMS's underlying storage and transaction management systems. We believe 2VNL is the first algorithm that can be implemented on top of current DBMS's using a query rewrite approach. Also, by assuming characteristics common in a data warehouse—that maintenance transactions run one at a time and only a few attributes in summary tables are updatable—our 2VNL algorithm provides concurrency for warehouse readers and the maintenance transaction with very little overhead.

7 Conclusions and Future Work

We have presented a concurrency control algorithm that is especially suited to view maintenance in a data warehousing environment. Using our algorithm, readers of the warehouse can execute concurrently with a view-maintenance transaction without blocking, readers and the maintenance transaction are serializable (*i.e.*, both access consistent versions of the database), and readers and the maintenance transaction do not need to place any locks. Allowing the maintenance transaction to execute concurrently with readers has significant advantages, including making the warehouse available to readers 24 hours a day and allowing maintenance transactions to be longer and/or more frequent.

Our algorithm is a type of multi-version algorithm especially suited to the data warehousing environment. It does not suffer from the problem of readers delaying writer commit that 2V2PL algorithms suffer from. Nor does it suffer from the additional I/O's required to access and manage multiple separate versions of each tuple that most MV2PL algorithms suffer from, because in 2VNL both versions of each tuple are stored together in the same physical location. Our approach is very space-efficient when only a few attributes are updatable by maintenance transactions, which is often the case in data warehousing environments, even if a large percentage of tuples are updated. Finally, we have shown how our algorithm can be implemented easily on top of current DBMS's through query rewrite as long as the DBMS supports the "read-uncommitted" transaction isolation level and physical tuple updates are performed in-place.

We intend to pursue the following areas as future work.

- Building the algorithm into a DBMS would probably yield better performance than implementing it through

[2] Not all multi-version algorithms are MV2PL algorithms. That is, not all use two-phase locking to synchronize writers. But since synchronizing writers is not the focus of this paper (we assume maintenance transactions are the only writers, and they are limited to executing one at a time by an external protocol), and since all multi-version algorithms use essentially the same technique for synchronizing readers, we describe only MV2PL algorithms here.

query rewrite. Thus, we would like to implement and compare both approaches, both from a performance and software engineering standpoint.

- We want to compare performance in terms of number of I/O's and storage space required between our 2VNL algorithm and MV2PL algorithms presented in the literature.
- Tuples that have been marked deleted can be removed once $tupleVN < sessionVN-1$ for all active readers. We intend to explore efficient methods for "garbage collecting" old tuples.
- Because tuples that have been modified by the maintenance transaction contain sufficient information to extract their previous version, maintenance transactions can execute without the need to log before-images. We plan to explore ways to allow rolling back maintenance transactions without logging by reverting to the previous version of tuples in the database.

Acknowledgments

We thank the one thorough SIGMOD reviewer for his or her useful comments and corrections, Janet Wiener for helpful comments on a previous version of the paper, and Hector Garcia-Molina for pointers to related work.

References

[AS89] D. Agrawal and S. Sengupta. Modular synchronization in multiversion databases: Version control and currency control. In *Proceedings of ACM SIGMOD 1989 International Conference on Management of Data*, pages 408–417, 1989.

[BBG+95] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of ACM SIGMOD 1995 International Conference on Management of Data*, pages 1–10, 1995.

[BC92a] P. Bober and M. Carey. Multiversion query locking. In *Proceedings of the Eighteenth International Conference on Very Large Databases (VLDB)*, pages 497–510, 1992.

[BC92b] P. Bober and M. Carey. On mixing queries and transactions via multiversion locking. In *Proceedings of the Eighth International Conference on Data Engineering*, pages 535–545, 1992.

[BHG87] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[BHR80] R. Bayer, H. Heller, and A. Reiser. Parallelism and recovery in database systems. *ACM Transactions on Database Systems*, 5(2):139–156, June 1980.

[CFL+82] A. Chan, S. Fox, W. Lin, A. Nori, and D. Ries. The implementation of an integrated concurrency control and recovery scheme. In *Proceedings of ACM SIGMOD 1982 International Conference on Management of Data*, pages 184–191, 1982.

[GL95] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In M. Carey and D. Schneider, editors, *Proceedings of ACM SIGMOD 1995 International Conference on Management of Data*, pages 328–339, San Jose, CA, May 23-25 1995.

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

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

[MPL92] C. Mohan, H. Pirahesh, and R. Lorie. Efficient and flexible methods for transient versioning of records to avoid locking by read-only transactions. In *Proceedings of ACM SIGMOD 1992 International Conference on Management of Data*, pages 124–133, 1992.

[MS93] J. Melton and A. Simon. *Understanding the New SQL: A Complete Guide.* Morgan Kaufmann, 1993.

[SJGP90] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching and views in data base systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 281–290, 1990.

[SP89] A. Segev and J. Park. Updating distributed materialized views. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):173–184, June 1989.

[SR81] R. Stearns and D. Rosenkrantz. Distributed database concurrency controls using before-values. In *Proceedings of ACM SIGMOD 1981 International Conference on Management of Data*, pages 74–83, 1981.

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

[Wei87] W. Weihl. Distributed version management for read-only actions. *IEEE Transactions on Software Engineering*, SE-13(1):55–64, January 1987.

[WYC93] K. Wu, P. Yu, and M. Chen. Dynamic finite versioning: An effective versioning approach to concurrent transaction and query processing. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 557–586, 1993.