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As changes are made to the data at the sources, the views at the warehouse become out of date.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 F30602-84-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 ≥ 2). In Section 6 we compare the 2VNL algorithm to two-version two-phase locking (2V2PL) and multi-version two-phase locking (MV2PL) algorithms [BHGS87] 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.

\[
\text{DailySales} \left( \text{cit}y, \text{state}, \text{product line}, \text{date}, \text{total sales} \right)
\]

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:

\[
\text{SELECT city, state, SUM(totalSales)} \\
\text{FROM DailySales} \\
\text{GROUP BY city, state}
\]

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:

\[
\text{SELECT product line, SUM(totalSales)} \\
\text{FROM DailySales} \\
\text{WHERE city = “San Jose” and state = “CA”} \\
\text{GROUP BY product line}
\]

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

¹Figure 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.
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 \( n \)VNL (\( n \geq 2 \)) case. The \( n \)VNL algorithm is considered in Section 5.

### 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, \( \text{current VN} \), records the current version number. Variable \( \text{current VN} \) is 1 initially. In addition, a global flag \( \text{maintenance Active} \) 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 \( \text{tuple VN} \) attribute that records the version of the database when the tuple was last modified by a maintenance transaction, (ii) an \( \text{operation} \) attribute that records the operation \( \{ \text{insert, update, delete} \} \) last performed on the tuple, and (iii) a set of \( \text{pre-update attributes} \) that contain the previous values of the updatable attributes of the tuple (i.e., those attributes that could be changed by an update oper-
atation 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 tuple VN, 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 tuple VN, and the version of the tuple that was current in database version tuple VN-1.

When a maintenance transaction begins, it reads the global current VN variable and sets a local maintenance VN variable to current VN+1. In addition, it sets the global maintenance Active 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 tuple VN and operation attributes in the tuple with its maintenance VN 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 delete 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 current VN with its maintenance VN and sets the maintenance Active 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 current VN and copies it into its local session VN 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 <= session VN. 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 session VN is compared with the global current VN. 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 A = {A1, A2, ..., An} be the initial set of attributes for a relation R, and let A' = {A1, A2, ..., An, AP, AP2, ..., APn} be the subset of attributes in A that are updatable. Then the extended schema of R is {tuple VN, operation, A1, A2, ..., An, AP, AP2, ..., APn}, where AP is used to denote the pre-update attribute corresponding to updatable attribute A1. Attribute tuple VN contains the current VN of the maintenance transaction that most recently modified the tuple, operation contains the logical operation performed by that maintenance transaction, attributes {A1, A2, ..., An} contain the current values of the tuple attributes, and pre-update attributes {AP, AP2, ..., APn} 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-group-by 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-
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 ≤ 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 ≤ 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.

<table>
<thead>
<tr>
<th>city</th>
<th>state</th>
<th>product_line</th>
<th>date</th>
<th>total_sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>San Jose</td>
<td>CA</td>
<td>golf equip</td>
<td>10/14/96</td>
<td>10,000</td>
</tr>
<tr>
<td>Berkeley</td>
<td>CA</td>
<td>racquetball</td>
<td>10/14/96</td>
<td>12,000</td>
</tr>
<tr>
<td>Novato</td>
<td>CA</td>
<td>rollerblades</td>
<td>10/13/96</td>
<td>8,000</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>operation</th>
<th>insert</th>
<th>update</th>
<th>delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current Version</td>
<td>read current attribute values</td>
<td>read current attribute values</td>
<td>ignore tuple</td>
</tr>
<tr>
<td>Pre-Update Version</td>
<td>ignore tuple</td>
<td>read pre-update attribute values</td>
<td>read pre-update attribute values</td>
</tr>
</tbody>
</table>

Table 2: Decision table for insert maintenance operation

<table>
<thead>
<tr>
<th>Previous operation</th>
<th>insert</th>
<th>update</th>
<th>delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>tupleVN &lt; maintenanceVN</td>
<td>impossible</td>
<td>impossible</td>
<td>Update tuple: PV&lt;-&gt;nulls CV&lt;-&gt;MV tupleVN&lt;-&gt;maintenanceVN operation&lt;-&gt;insert</td>
</tr>
<tr>
<td>tupleVN = maintenanceVN</td>
<td>impossible</td>
<td>impossible</td>
<td>Update tuple: CV&lt;-&gt;MV operation&lt;-&gt;update</td>
</tr>
<tr>
<td>No Conflicting Tuple</td>
<td>insert tuple: PV&lt;-&gt;nulls CV&lt;-&gt;MV tupleVN&lt;-&gt;maintenanceVN operation&lt;-&gt;insert</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Decision table for update maintenance operation

<table>
<thead>
<tr>
<th>Previous operation</th>
<th>insert</th>
<th>update</th>
<th>delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>tupleVN = maintenanceVN</td>
<td>Update tuple: CV&lt;-&gt;MV</td>
<td>impossible</td>
<td></td>
</tr>
</tbody>
</table>

Table 4: Decision table for delete maintenance operation

<table>
<thead>
<tr>
<th>Previous operation</th>
<th>insert</th>
<th>update</th>
<th>delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>tupleVN &lt; maintenanceVN</td>
<td>Update tuple: PV&lt;-&gt;CV tupleVN&lt;-&gt;maintenanceVN operation&lt;-&gt;delete</td>
<td>Update tuple: PV&lt;-&gt;CV tupleVN&lt;-&gt;maintenanceVN operation&lt;-&gt;delete</td>
<td>impossible</td>
</tr>
<tr>
<td>tupleVN = maintenanceVN</td>
<td>Delete tuple</td>
<td>Update tuple: operation&lt;-&gt;delete</td>
<td>impossible</td>
</tr>
</tbody>
</table>
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 maintenance VN for the maintenance transaction and the tuple’s existing values for tuple VN 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 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 $city$, $state$, $productLine$, 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,SI95] 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

| insert: | San Jose | CA | golf equip | 10/16/96 | 10,000 |
| update: | San Jose | CA | rollerblades | 10/14/96 | 10,000 |
| delete: | Berkeley | CA | racquetball | 10/14/96 | null |

**Figure 5: Example maintenance transaction**

<table>
<thead>
<tr>
<th>tupleVN</th>
<th>operation</th>
<th>city</th>
<th>state</th>
<th>productLine</th>
<th>date</th>
<th>totalSales</th>
<th>prevTotalSales</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>update</td>
<td>San Jose</td>
<td>CA</td>
<td>golf equip</td>
<td>10/14/96</td>
<td>10,200</td>
<td>10,000</td>
</tr>
<tr>
<td>4</td>
<td>insert</td>
<td>San Jose</td>
<td>CA</td>
<td>golf equip</td>
<td>10/15/96</td>
<td>1,500</td>
<td>null</td>
</tr>
<tr>
<td>5</td>
<td>delete</td>
<td>Berkeley</td>
<td>CA</td>
<td>racquetball</td>
<td>10/14/96</td>
<td>12,000</td>
<td>12,000</td>
</tr>
<tr>
<td>5</td>
<td>insert</td>
<td>Novato</td>
<td>CA</td>
<td>rollerblades</td>
<td>10/15/96</td>
<td>6,000</td>
<td>null</td>
</tr>
<tr>
<td>5</td>
<td>insert</td>
<td>San Jose</td>
<td>CA</td>
<td>golf equip</td>
<td>10/16/96</td>
<td>11,000</td>
<td>null</td>
</tr>
</tbody>
</table>

**Figure 6: Result of DailySales after maintenance transaction**
rewriting process by showing how to implement the decision tables of Sections 3.2 and 3.3 using SQL.

To implement 2VN 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 prev_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 = 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 since it selects on the key.

For each tuple t to insert

```
INSERT INTO DailySales VALUES
  (:maintenance VN, "insert", t.city, t.state,
  t.product_line, t.date, t.total_sales, null)
```

If insert failed due to a unique key conflict,

Let \( r = t \) \( \% r \) has same key value as \( t \)

(SEELECT * FROM DailySales
  WHERE city = t.city AND state = t.state
  AND product_line = t.product_line
  AND date = t.date)

If \( t\) tuple VN < :maintenance VN, \% line 1 in Table 2

Update \( r \)

```
set r.pre_total_sales = null
set r.total_sales = r.total_sales + t.total_sales
set r.tuple VN = :maintenance VN
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 transac-
tion, 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 transac-
tion, 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 hav-
ing 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 \( r\) tuple VN < :maintenance VN, \% 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.tuple VN = :maintenance VN
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 phsyical
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 tuple VN 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 \( r\) tuple VN < :maintenance VN, \% line 1 in Table 4

Update \( r \)

```
set r.pre_total_sales = r.total_sales
set r.tuple VN = :maintenance VN
set r.operation = "delete"
```

Else \% line 2 in Table 4

```
set 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 algo-
rithm. Indexes on non-updatable attributes are not affected
by the algorithm. Note especially that for the summary ta-
bles commonly found in data warehouses, indexes are usu-
ally 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. In-
dices 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 $n$VNL solution.

The $n$VNL 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 $n$VNL 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. $n$VNL, on the other hand, guarantees that reader sessions lasting up to $i + m$ will never expire. In general, $n$VNL 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 $n$VNL is not to allow multiple concurrent writers, but rather to allow readers to (sequentially) overlap multiple maintenance transactions. Consequently, $n$VNL still differs from other multi-version algorithms in significant ways, as discussed earlier and in Section 6 below.

For $n$VNL the relation schema is modified as described in Section 3.1, except there are $n - 1$ tuple $VN$ attributes (tuple $VN_1, \ldots, \text{tuple } VN_{n-1}$), $n - 1$ corresponding operation attributes, and $n - 1$ corresponding sets of pre-update attributes. Attribute tuple $VN_i$ contains the maintenance $VN$ of the $i$-th most recent maintenance transaction to modify the tuple. operation, 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 session $VN$, meaning the version that includes the effects of all maintenance transactions with maintenance $VN \leq$ session $VN$, and no other maintenance transactions. Recalling that tuple $VN_i$ denotes the largest (most recent) maintenance $VN$, and tuple $VN_{n-1}$ denotes the smallest (least recent) maintenance $VN$, the cases for a reader are:

1. session $VN \geq$ tuple $VN$: Read the current version of the tuple.
2. $\text{tuple } VN_{n-1} - 1 \leq$ session $VN < \text{tuple } VN$: Read the pre-update version of the tuple for the least tuple $VN_j >$ session $VN$.
3. session $VN < \text{tuple } VN_{n-1} - 1$: 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, and the first row in the table is followed. For case (2), operation is operation, 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 tuple $VN$, operation, and the pre-update attribute values, we “push back” attributes, eliminating the existing $n$th version in order to make room for a new 1st version. That is, we set tuple $VN_{i+1} \leftarrow$ tuple $VN_i$ for $1 \leq i \leq n - 2$, and similarly for the operation’s and the pre-update attributes. We then set tuple $VN_1$, operation, 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 $n$VNL 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 maintenance $VN = 3$ inserts the tuple with total sales = 10,000, then a maintenance transaction with maintenance $VN = 5$ updates the total sales to 10,200, then a maintenance transaction with maintenance $VN = 6$ 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 session $VN \geq 6$ will ignore the tuple. A reader with session $VN < 6$ but $\geq 2$ will read the pre-update version of the tuple for the least tuple $VN_j >$ session $VN$ that is, a reader with session $VN \in \{3, 4\}$ will see the logical tuple with total sales = 10,000, and a reader with session $VN = 2$ will ignore the tuple. A reader with session $VN < 2$ 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 [BBG95,BHGS87]. 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.
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 2VN) 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, BC92, BC92b, CFL82, Wei87, MPL92, WYC93] guarantee that readers and writers never block each other. MV2PL algorithms maintain sufficient previous versions of each tuple (or page, if versioning is at the page level) that readers are 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 [CFL82]. 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 [CFL82] 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/Os 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 multi-version 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.

### Figure 7: Example 2VNL tuple

<table>
<thead>
<tr>
<th>city</th>
<th>state</th>
<th>product line</th>
<th>date</th>
<th>total sales</th>
<th>tuple VN1</th>
<th>operations</th>
<th>pre_total sales</th>
<th>tuple VN2</th>
<th>operations</th>
<th>pre_total sales</th>
</tr>
</thead>
<tbody>
<tr>
<td>San Jose</td>
<td>CA</td>
<td>golf equip</td>
<td>10/14/96</td>
<td>10,200</td>
<td>delete</td>
<td>10,200</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 commits 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
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 tuple VN < session VN=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


