5

MULTIVERSION CONCURRENCY CONTROL

5.1 INTRODUCTION

In a multiversion concurrency control algorithm, each Write on a data item x produces a new copy (or *version*) of x. The DM that manages x therefore keeps a list of versions of x, which is the history of values that the DM has assigned to x. For each Read(x), the scheduler not only decides when to send the Read to the DM, but it also tells the DM which one of the versions of x to read.

The benefit of multiple versions for concurrency control is to help the scheduler avoid rejecting operations that arrive too late. For example, the scheduler normally rejects a Read because the value it was supposed to read has already been overwritten. With multiversions, such old values are never overwritten and are therefore always available to tardy Reads. The scheduler can avoid rejecting the Read simply by having the Read read an old version.

Maintaining multiple versions may not add much to the cost of concurrency control, because the versions may be needed anyway by the recovery algorithm. As we'll see in the next chapter, many recovery algorithms have to maintain some before image information, at least of those data items that have been updated by active transactions; the recovery algorithm needs those before images in case any of the active transactions abort. The before images of a data item are exactly its list of old versions. It is a small step for the DM to make those versions explicitly available to the scheduler.

An obvious cost of maintaining multiple versions is storage space. To control this storage requirement, versions must periodically be purged or

archived. Since certain versions may be needed by active transactions, purging versions must be synchronized with respect to active transactions. This purging activity is another cost of multiversion concurrency control.

We assume that if a transaction is aborted, any versions it created are destroyed. In our subsequent discussion, the term "version" will refer to the value of a data item produced by a transaction that's either active or committed. Thus, when the scheduler decides to assign a particular version of x to Read(x), the value returned is not one produced by an aborted transaction. If the version read is one produced by an active transaction, recoverability requires that the reader's commitment be delayed until the transaction that produced the version has committed. If that transaction actually aborts (thereby invalidating its version), the reader must also be aborted.

The existence of multiple versions is only visible to the scheduler and DM, not to user transactions. Transactions still reference data items, such as x and y. Users therefore expect the DBS to behave as if there were only one version of each data item, namely, the last one that was written from that user's perspective. The scheduler may use multiple versions to improve performance by rejecting operations less frequently. But it must not change the system's functionality over a single version view of the database.

There are many applications of databases in which users *do* want to explicitly access each of the multiple versions of a data item. For example, a user may wish to maintain several versions of a design database: the last design released for manufacturing, the last design checked for correctness, and the most recent working design. The user may update each version of the design independently. Since the existence of these multiple versions is not transparent to the user, such applications are not appropriate for the multiversion concurrency control algorithms described in this chapter.

Analyzing Correctness

To analyze the correctness of multiversion concurrency control algorithms, we need to extend serializability theory. This extension requires two types of histories: multiversion (MV) histories that represent the DM's execution of operations on a multiversion database, and single version (1V) histories that represent the interpretation of MV histories in the users' single version view of the database. Serial 1V histories are the histories that the user regards as correct. But the system actually produces MV histories. So, to prove that a concurrency control algorithm is correct, we must prove that each of the MV histories that it can produce is equivalent to a serial 1V history.

What does it mean for an MV history to be equivalent to a 1V history? Let's try to answer this by extending the definition of equivalence of 1V histories that we used in Chapters 2–4. To attempt this extension, we need a little

notation. For each data item x, we denote the versions of x by x_i, x_j, ..., where the subscript is the index of the transaction that wrote the version. Thus, each Write in an MV history is always of the form $w_i[x_i]$, where the version subscript equals the transaction subscript. Reads are denoted in the usual way, such as $r_i[x_j]$.

Suppose we adopt a definition of equivalence that says an MV history H_{MV} is equivalent to a 1V history H_{1V} if every pair of conflicting operations in H_{MV} is in the same order in H_{1V}. Consider the MV history

$$H_1 = w_0[x_0]\ c_0\ w_1[x_1]\ c_1\ r_2[x_0]\ w_2[y_2]\ c_2.$$

The only two operations in H_1 that conflict are $w_0[x_0]$ and $r_2[x_0]$. The operation $w_1[x_1]$ does not conflict with either $w_0[x_0]$ or $r_2[x_0]$, because it operates on a different version of x than those operations, namely x_1. Now consider the 1V history

$$H_2 = w_0[x]\ c_0\ w_1[x]\ c_1\ r_2[x]\ w_2[y]\ c_2.$$

We constructed H_2 by mapping each operation on versions x_0, x_1, and y_2 in H_1 into the same operation on the corresponding data items x and y. Notice that the two operations in H_1 that conflict, $w_0[x_0]$ and $r_2[x_0]$, are in the same order in H_2 as in H_1. So, according to the definition of equivalence just given, H_1 is equivalent to H_2. But this is not reasonable. In H_2, T_2 reads x from T_1, whereas in H_1, T_2 reads x from T_0.[1] Since T_2 reads a different value of x in H_1 and H_2, it may write a different value in y.

This definition of equivalence based on conflicts runs into trouble because MV and 1V histories have slightly different operations — version operations versus data item operations. These operations have different conflict properties. For example, $w_1[x_1]$ does not conflict with $r_2[x_0]$, but their corresponding 1V operations $w_1[x]$ and $r_2[x]$ *do* conflict. Therefore, a definition of equivalence based on conflicts is inappropriate.

To solve this problem, we need to return to first principles by adopting the more fundamental definition of view equivalence developed in Section 2.6. Recall that two histories are *view equivalent* if they have the same reads-from relationships and the same final writes. Comparing histories H_1 and H_2, we see that T_2 reads x from T_0 in H_1, but T_2 reads x from T_1 in H_2. Thus, H_1 is not view equivalent to H_2.

Now that we have a satisfactory definition of equivalence, we need a way of showing that every MV history H produced by a given multiversion concurrency control algorithm is equivalent to a serial 1V history. One way would be to show that SG(H) is acyclic, so H is equivalent to a serial MV history. Unfortunately, this doesn't help much, because not every serial MV history is equivalent to a serial 1V history. For example, consider the serial MV history

---

[1]Recall from Section 2.4 that T_i *reads x from* T_j in H if (1) $w_j[x] < r_i[x]$, (2) $a_j \ll r_i[x]$, and (3) if there is some $w_k[x]$ such that $w_j[x] < w_k[x] < w_i[x]$, then $a_k < r_i[x]$.

$$H_3 = w_0[x_0]\ w_0[y_0]\ c_0\ r_1[x_0]\ r_1[y_0]\ w_1[x_1]\ w_1[y_1]\ c_1\ r_2[x_0]\ r_2[y_1]\ c_2.$$

If we treat the versions of x and y as independent data items, then we get

$$SG(H_3) = T_0 \rightarrow T_1 \rightarrow T_2.$$

Although H_3 is serial and $SG(H_3)$ is acyclic, H_3 is not equivalent to a serial 1V history. For example, consider the 1V history

$$H_4 = w_0[x]\ w_0[y]\ c_0\ r_1[x]\ r_1[y]\ w_1[x]\ w_1[y]\ c_1\ r_2[x]\ r_2[y]\ c_2.$$

We can show that H_3 is not view equivalent to H_4 by showing that they do not have the same reads-from relationships. In H_4, T_2 reads x and y from T_1. But in H_3, T_2 reads x from T_0 and reads y from T_1. Since T_2 reads different values in H_3 and H_4, the two histories are not equivalent. Similarly, H_3 is not equivalent to the 1V history

$$H_5 = w_0[x]\ w_0[y]\ c_0\ r_2[x]\ r_2[y]\ c_2\ r_1[x]\ r_1[y]\ w_1[x]\ w_1[y]\ c_1.$$

Clearly, H_3 is not equivalent to any 1V serial history over the same set of transactions.

Only a subset of serial MV histories, called 1-serial MV histories, are equivalent to serial 1V histories. Intuitively, a serial MV history is *1-serial* if for each reads-from relationship, say T_i reads x from T_j, T_j is the last transaction preceding T_i that writes *any* version of x. Notice that H_3 is not 1-serial because T_2 reads x from T_0, not T_1, which is the last transaction preceding T_2 that writes x.

All 1-serial MV histories are equivalent to serial 1V histories, so we can define 1-serial histories to be correct. To prove that a multiversion concurrency control algorithm is correct, we must show that its MV histories are equivalent to 1-serial MV histories. We will do this by defining a new graph structure called a multiversion serialization graph (MVSG). An MV history is equivalent to a 1-serial MV history iff it has an acyclic MVSG. Now proving multiversion concurrency control algorithms correct is just like standard serializability theory. We simply prove that its histories have acyclic MVSGs. We now proceed with a formal development of this line of proof.

5.2 *MULTIVERSION SERIALIZABILITY THEORY[2]

Let $T = \{T_0, \ldots T_n\}$ be a set of transactions, where the operations of T_i are ordered by $<_i$ for $0 \leq i \leq n$. To process operations from T, a multiversion scheduler must translate T's operations on (single version) data items into operations on specific versions of those data items. We formalize this transla-

[2]This section requires reading Section 2.6 as a prerequisite. We recommend skipping this and other starred sections of this chapter on the first reading, to gain some intuition for multiversion algorithms before studying their serializability theory.

tion by a function h that maps each $w_i[x]$ into $w_i[x_i]$, each $r_i[x]$ into $r_i[x_j]$ for some j, each c_i into c_i, and each a_i into a_i.

A *complete multiversion (MV) history* H over T is a partial order with ordering relation $<$ where

1. $H = h(\bigcup_{i=0}^{n} T_i)$ for some translation function h;
2. for each T_i and all operations p_i, q_i in T_i, if $p_i <_i q_i$, then $h(p_i) < h(q_i)$;
3. if $h(r_j[x]) = r_j[x_i]$, then $w_i[x_i] < r_j[x_i]$;
4. if $w_i[x] <_i r_i[x]$, then $h(r_i[x]) = r_i[x_i]$; and
5. if $h(r_j[x]) = r_j[x_i]$, $i \neq j$, and $c_j \in H$, then $c_i < c_j$.

Condition (1) states that the scheduler translates each operation submitted by a transaction into an appropriate multiversion operation. Condition (2) states that the MV history preserves all orderings stipulated by transactions. Condition (3) states that a transaction may not read a version until it has been produced.[3] Condition (4) states that if a transaction writes into a data item x before it reads x, then it must read the version of x that it previously created. This ensures that H is consistent with the implied semantics of the transactions over which it is defined. If H satisfies condition (4), we say that it *preserves reflexive reads-from relationships*. Condition (5) says that before a transaction commits, all the transactions that produced versions it read must have already committed. If H satisfies this condition we say it is *recoverable*.

An *MV history* H is a prefix of a complete MV history. We say that an MV history *preserves reflexive reads-from relationships* (or is *recoverable*) if it is the prefix of a complete MV history that does so. As in 1V histories, a transaction T_i is *committed* (respectively *aborted*) in an MV history H if c_i (respectively a_i) is in H. Also, the *committed projection* of an MV history H, denoted $C(H)$, is defined as for 1V histories; that is, $C(H)$ is obtained by removing from H the operations of all but the committed transactions. It is easy to check that if H is an MV history then $C(H)$ is a complete MV history, i.e., $C(H)$ satisfies conditions (1) – (5) (see Exercise 5.2).

For example, given transactions $\{T_0, T_1, T_2, T_3, T_4\}$,

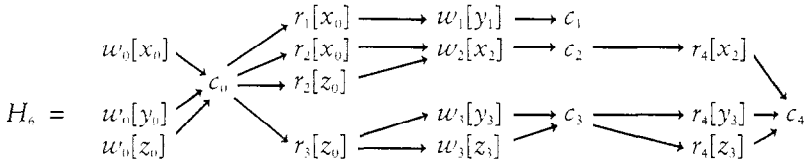
$$T_0 = \begin{matrix} w_0[x] \\ w_0[y] \\ w_0[z] \end{matrix} \rightarrow c_0 \qquad T_3 = r_3[z] \rightarrow \begin{matrix} w_3[y] \\ w_3[z] \end{matrix} \rightarrow c_3$$

$$T_1 = r_1[x] \rightarrow w_1[y] \rightarrow c_1 \qquad T_4 = \begin{matrix} r_4[x] \\ r_4[y] \\ r_4[z] \end{matrix} \rightarrow c_4$$

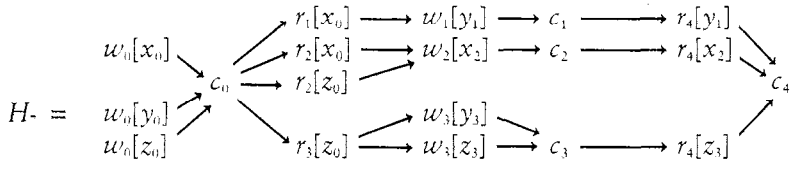
$$T_2 = \begin{matrix} r_2[x] \\ r_2[z] \end{matrix} \rightarrow w_2[x] \rightarrow c_2$$

[3]To ensure condition (3), we will normally include in our examples an initializing transaction, T_0, that creates the initial version of each data item.

the following history, H_6, is a complete MV history over $\{T_0, T_1, T_2, T_3, T_4\}$.



All complete MV histories over a set of transactions must have the same Writes, but they need not have the same Reads. For example, H- has $r_4[y_1]$ instead of $r_4[y_3]$.



MV History Equivalence

Two 1V histories over the same transactions are *view* equivalent if they contain the same operations, have the same reads-from relationships, and the same final writes. However, for MV histories, we can safely drop "and the same final writes" from the definition. If two histories are over the same transactions, then they have the same Writes. Since no versions are overwritten, all Writes are effectively final writes. Thus, if two MV histories over the same transactions have the same operations and the same reads-from relationships, then they have the same final writes and are therefore view equivalent.

To formalize the definition of equivalence, we must formalize the notion of reads-from in MV histories. To do this, we replace the notion of *data item* by *version* in the ordinary definition of reads-from for 1V histories. Transaction T_j *reads x from* T_i in MV history H if T_j reads the version of x produced by T_i. Since the version of x produced by T_i is x_i, T_j reads x from T_i in H iff T_j reads x_i, that is, iff $r_j[x_i] \in H$.

Two MV histories over a set of transactions T are *equivalent*, denoted $\equiv$, if they have the same operations and the same reads-from relationships. In view of the preceding discussion, having the same reads-from relationships amounts to having the same Read operations. Therefore, equivalence of MV histories reduces to a trivial condition, as stated in the following proposition.

Proposition 5.1: Two MV histories over a set of transactions are equivalent iff the histories have the same operations. □

Next we want to define the equivalence of an MV history H_{MV} to a 1V history H_{1V}. We will only be interested in such an equivalence if H_{1V} is a valid one version view of H_{MV}. That is, H_{1V} and H_{MV} must be over the same set of transactions and their operations must be in one-to-one correspondence. More precisely, there must be a bijective (one-to-one and onto) function from the operations of H_{1V} to those of H_{MV}, mapping c_i to c_i, a_i to a_i, $r_i[x]$ to $r_i[x_j]$ for some version x_j of x and $w_i[x]$ to $w_i[x_i]$.

Given that the operations of H_{MV} and H_{1V} are in one-to-one correspondence, we can talk about their reads-from relationships being the same. We need not worry about final writes; all of the final writes in H_{1V} must be part of the state produced by H_{MV}, because H_{MV} retains all versions written in it. So, just like MV histories, an MV history and 1V history are *equivalent* if they have the same reads-from relationships.[4]

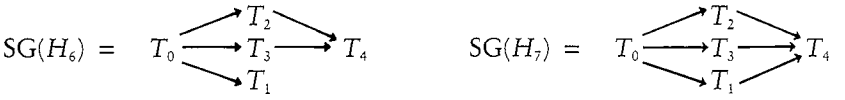
Serialization Graphs

Two operations in an MV history *conflict* if they operate on the same version and one is a Write. Only one pattern of conflict is possible in an MV history: if $p_i < q_j$ and these operations conflict, then p_i is $w_i[x_i]$ and q_j is $r_j[x_i]$ for some data item x. Conflicts of the form $w_i[x_i] < w_j[x_i]$ are impossible, because each Write produces a unique new version. Conflicts of the form $r_j[x_i] < w_i[x_i]$ are impossible, because T_j cannot read x_i until it has been produced. Thus, all conflicts in an MV history correspond to reads-from relationships.

The serialization graph for an MV history is defined as for a 1V history. But since only one kind of conflict is possible in an MV history, SGs are quite simple. Let H be an MV history. SG(H) has nodes for the committed transaction in H and edges $T_i \rightarrow T_j$ ($i \neq j$) whenever for some x, T_j reads x from T_i. That is, $T_i \rightarrow T_j$ is present iff for some x, $r_j[x_i]$ ($i \neq j$) is an operation of C(H). This gives us the following proposition.

Proposition 5.2: Let H and H' be MV histories. If $H \equiv H'$, then SG(H) = SG(H'). □

The serialization graphs of H_6 and H_7 follow.



[4]Two 1V histories can be equivalent in this sense without being view equivalent to each other, because they don't have the same final writes.

One Copy Serializability

A complete MV history is *serial* if for every two transactions T_i and T_j that appear in H, either all of T_i's operations precede all of T_j's or vice versa. Not all serial MV histories behave like ordinary serial 1V histories, for example,

$$H_3 = w_0[x_0]\ w_0[y_0]\ c_0\ r_1[x_0]\ r_1[y_0]\ w_1[x_1]\ w_1[y_1]\ c_1\ r_2[x_0]\ r_2[y_1]\ c_2.$$

The subset of serial MV histories that are equivalent to serial 1V histories is defined as follows.

A serial MV history H is *one-copy serial* (or *1-serial*) if for all i, j, and x, if T_i reads x from T_j, then $i = j$, or T_j is the last transaction preceding T_i that writes into *any* version of x. Since H is serial, the word *last* in this definition is well defined. History H_3 is not 1-serial because T_2 reads x from T_0 but $w_0[x_0] < w_1[x_1] < r_2[x_0]$. History H_8, which follows, *is* 1-serial.

$$H_8 = w_0[x_0]\ w_0[y_0]\ w_0[z_0]\ c_0\ r_1[x_0]\ w_1[y_1]\ c_1\ r_2[x_0]\ r_2[z_0]\ w_2[x_2]\ c_2\ r_3[z_0]\ w_3[y_3]\ w_3[z_3]\ c_3\ r_4[x_2]\ r_4[y_3]\ r_4[z_3]\ c_4$$

An MV history is *one-copy serializable* (or *1SR*) if its committed projection is equivalent to a 1-serial MV history.[5] For example, H_6 is 1SR because $C(H_6) = H_6$ is equivalent to H_8, which you can verify by Proposition 5.1. $C(H_7) = H_7$ is equivalent to no 1-serial history, and thus H_7 is not 1SR.

A serial history can be 1SR even though it is not 1-serial. For example,

$$H_9 = w_0[x_0]\ c_0\ r_1[x_0]\ w_1[x_1]\ c_1\ r_2[x_0]\ c_2$$

is not 1-serial since T_2 reads x from T_0 instead T_1. But it *is* 1SR, because it is equivalent to

$$H_{10} = w_0[x_0]\ c_0\ r_2[x_0]\ c_2\ r_1[x_0]\ w_1[x_1]\ c_1.$$

To justify the value of one-copy serializability as a correctness criterion, we need to show that the committed projection of every 1SR history is equivalent to a serial 1V history.

Theorem 5.3: Let H be an MV history over T. $C(H)$ is equivalent to a serial, 1V history over T iff H is 1SR.

Proof: (If) Since H is 1SR, $C(H)$ is equivalent to a 1-serial MV history H_s. Translate H_s into a serial 1V history H_s', by translating each $w_i[x_i]$ into $w_i[x]$ and $r_j[x_i]$ into $r_j[x]$. To show $H_s \equiv H_s'$, consider any reads-from relationship in H_s, say T_j reads x from T_i. Since H_s is 1-serial, no $w_k[x_k]$ lies between $w_i[x_i]$ and $r_j[x_i]$. Hence no $w_k[x]$ lies between $w_i[x]$ and $r_j[x]$ in H_s'.

---

[5] It turns out that this is a prefix commit-closed property. Unlike view serializability, we need not require that the committed projection of *every prefix* of an MV history be equivalent to a 1-serial MV history. This follows from the fact that the committed projection of the history itself is equivalent to a 1-serial MV history (see Exercise 5.4).

Thus T_j reads x from T_i in H_s'. Now consider a reads-from relationship in H_s', T_j reads x from T_i. If $r_j[x]$ was translated from $r_j[x_i]$ in H_s, then T_j reads x from T_i in H_s and we're done. So assume instead that $r_j[x]$ was translated from $r_j[x_k]$, $k \neq i$. If $i = j$, then $k = i$ by condition (4) in the definition of MV history and we're done. If $i \neq j$, then since H_s is 1-serial, either $w_i[x_i] < w_k[x_k]$ or $r_j[x_k] < w_i[x_i]$. But then, translating these operations into H_s' implies that T_j does not read x from T_i in H_s', a contradiction. Thus T_j reads x from T_i in H_s. This establishes $H_s' \equiv H_s$. Since $H_s \equiv C(H)$, $C(H) \equiv H_s'$ follows by transitivity of equivalence.

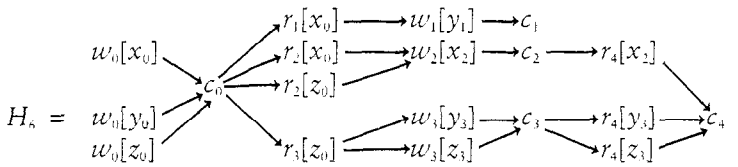
(Only if) Let H_s' be the hypothesized serial 1V history equivalent to $C(H)$. Translate H_s' into a serial MV history H_s by translating each c_i into c_i, $w_i[x]$ into $w_i[x_i]$, and $r_j[x]$ into $r_j[x_i]$ such that T_j reads x from T_i in H_s'. We must show that H_s is indeed a complete MV history. It is immediate that it satisfies conditions (1) and (2) of the complete MV history definition. For condition (3), it is enough to show that each $r_j[x]$ is preceded by some $w_i[x]$ in H_s'. Since H is an MV history, each $r_j[x_k]$ in $C(H)$ is preceded by $w_k[x_k]$. Since H_s' has the same reads-from relationships as the MV history $C(H)$, every Read in H_s', must be preceded by a Write on the same data item, as desired. To show H_s satisfies condition (4), note that if $w_j[x] < r_j[x]$ in H_s', then since H_s' is serial, T_j reads x from T_j in H_s' and $r_j[x]$ is translated into $r_j[x_j]$ in H_s. Finally, for condition (5) we must show that if $r_j[x_i]$ $(i \neq j)$ is in H_s then $c_i < c_j$. If $r_j[x_i]$ is in H_s then T_j reads x from T_i in H_s'. Since H_s' is serial and T_i, T_j are committed in it, we have $c_i < c_j$ in H_s'. By the translation then, it follows that $c_i < c_j$ in H_s, as wanted. This concludes the proof that H_s is indeed an MV history. Since the translation preserves reads-from relationships, so $H_s \equiv H_s'$. By transitivity, $C(H) \equiv H_s$.

It remains to prove that H_s is 1-serial. So consider any reads-from relationship in H_s, say T_j reads x from T_i, where $i \neq j$. Since H_s' is a 1V history, no $w_k[x]$ lies between $w_i[x]$ and $r_j[x]$. Hence no $w_k[x_k]$ lies between $w_i[x_i]$ and $r_j[x_i]$ in H_s. Thus, H_s is 1-serial, as desired. □

The 1-Serializability Theorem

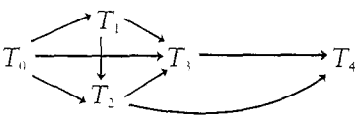
To determine if a multiversion concurrency control algorithm is correct, we must determine if all of its histories are 1SR. To do this, we use a modified SG. The modification is motivated by the fact that all known multiversion concurrency control algorithms sort the versions of each data item into a total order. We use this total order of versions to define an appropriately modified SG.

Given an MV history H and a data item x, a *version order*, $\ll$, for x in H is a total order of versions of x in H. A *version order* for H is the union of the version orders for all data items. For example, a possible version order for H_6 is $x_0 \ll x_2$, $y_0 \ll y_1 \ll y_3$, and $z_0 \ll z_3$.



Given an MV history H and a version order $\ll$, the *multiversion serialization graph for H and* $\ll$, MVSG(H, $\ll$), is SG(H) with the following *version order edges* added: for each $r_k[x_j]$ and $w_i[x_i]$ in C(H) where i, j, and k are distinct, if $x_i \ll x_j$ then include $T_i \rightarrow T_j$, otherwise include $T_k \rightarrow T_i$.[6] (Note that there is no version order edge if $j = k$, that is, if a transaction reads from itself.) For example, given the preceding version order for H_s,

MVSG(H_s, $\ll$) =



The version order edges that are in MVSG(H_s, $\ll$) but not in SG(H) are $T_1 \rightarrow T_2$, $T_1 \rightarrow T_3$, and $T_2 \rightarrow T_3$. Except for $T_0 \rightarrow T_1$, all edges in SG(H) are also version order edges.

Given an MV history H, suppose SG(H) is acyclic. We know that a serial MV history H_s obtained by topologically sorting SG(H) may not be equivalent to any serial 1V history. The reason is that some of H_s's reads-from relationships may be changed by mapping version operations into data item operations. The purpose of version order edges is to detect when this happens. If $r_k[x_j]$ and $w_i[x_i]$ are in C(H), then the version order edge forces $w_i[x_i]$ to either precede $w_j[x_j]$ or follow $r_k[x_j]$ in H_s. That way, when operations on x_i and x_j are mapped to operations on x when changing H_s to a 1V history, the reads-from relationship is undisturbed. This ensures that we can map H_s into an equivalent 1V history. Of course, all of this is possible only if SG(H) is still acyclic after adding version order edges. This observation leads us to the following theorem, which is our main tool for analyzing multiversion concurrency control algorithms.

Theorem 5.4: An MV history H is 1SR iff there exists a version order $\ll$ such that MVSG(H, $\ll$) is acyclic.

Proof: (If) Let H_s be a serial MV history $T_{i_1} T_{i_2} \ldots T_{i_n}$, where T_{i_1}, T_{i_2}, ... T_{i_n} is a topological sort of MVSG(H, $\ll$). Since C(H) is an MV history, it follows that H_s is as well. Since H_s has the same operations as C(H), by Proposition 5.1, $H_s \equiv$ C(H).

[6] Recall that the nodes of SG(H) and, therefore, of MVSG(H, $\ll$) are the *committed* transactions in H.

It remains to prove that H_s is 1-serial. Consider any reads-from relationship in H_s, say T_k reads x from T_j, $k \neq j$. Let $w_i[x_i]$ ($i \neq j$ and $i \neq k$) be any other Write on x. If $x_i \ll x_j$, then MVSG(H, $\ll$) includes the version order edge $T_i \rightarrow T_j$, which forces T_j to follow T_i in H_s. If $x_j \ll x_i$, then MVSG(H, $\ll$) includes the version order edge $T_k \rightarrow T_i$, which forces T_k to precede T_i in H_s. Therefore, no transaction that writes x falls in between T_j and T_k in H_s. Thus, H_s is 1-serial.

(Only if) Given H and $\ll$, let MV(H, $\ll$) be the graph containing *only* version order edges. Version order edges depend only on the operations in H and $\ll$; they do *not* depend on the *order* of operations in H. Thus, if H and H' are MV histories with the same operations, then MV(H, $\ll$) = MV(H', $\ll$) for all version orders $\ll$.

Let H_s be a 1-serial MV history equivalent to C(H). All edges in SG(H_s) go "left-to-right;" that is, if $T_i \rightarrow T_j$ then T_i precedes T_j in H_s. Define $\ll$ as follows: $x_i \ll x_j$ only if T_i precedes T_j in H_s. All edges in MV(H_s, $\ll$) are also left-to-right. Therefore all edges in MVSG(H_s, $\ll$) = SG(H_s) $\cup$ MV(H_s, $\ll$) are left-to-right. This implies MVSG(H_s, $\ll$) is acyclic.

By Proposition 5.1, C(H) and H_s have the same operations. Hence MV(C(H), $\ll$) = MV(H_s, $\ll$). By Proposition 5.1 and 5.2 SG(C(H)) = SG(H_s). Therefore MVSG(C(H), $\ll$) = MVSG(H_s, $\ll$). Since MVSG(H_s, $\ll$) is acyclic, so is MVSG(C(H), $\ll$), which is identical to MVSG(H, $\ll$). □

5.3 MULTIVERSION TIMESTAMP ORDERING

We can define schedulers for multiversion concurrency control based on each of the three basic types of schedulers: 2PL, TO, and SGT. We begin with a multiversion scheduler based on TO because it is the easiest to prove correct.

As for all TO schedulers, each transaction has a unique timestamp, denoted ts(T_i). Each operation carries the timestamp of its corresponding transaction. Each version is labeled by the timestamp of the transaction that wrote it.

A *multiversion TO* (*MVTO*) scheduler processes operations first-come-first-served. It translates operations on data items into operations on versions to make it appear as if it processed these operations in timestamp order on a single version database. The scheduler processes $r_i[x]$ by first translating it into $r_i[x_k]$, where x_k is the version of x with the largest timestamp less than or equal to ts(T_i), and then sending $r_i[x_k]$ to the DM. It processes $w_i[x]$ by considering two cases. If it has already processed a Read $r_j[x_k]$ such that ts(T_k) < ts(T_i) < ts(T_j), then it rejects $w_i[x]$. Otherwise, it translates $w_i[x]$ into $w_i[x_i]$ and sends it to the DM. Finally, to ensure recoverability, the scheduler must delay the processing of c_i until it has processed c_j for all transactions T_j that wrote versions read by T_i.

To understand MVTO, it is helpful to compare its effect to an execution, say H_{1V}, on a single version database in which operations execute in timestamp order. In H_{1V}, each Read, $r_i[x]$, reads the value of x with the largest timestamp less than or equal to $ts(T_i)$. This is the value of the version that the MVTO scheduler selects when it processes $r_i[x]$.

Since MVTO need not process operations in timestamp order, a Write could arrive whose processing would invalidate a Read that the scheduler already processed. For example, suppose $w_0[x_0] < r_2[x_0]$ represents the execution so far, where $ts(T_i) = i$ for all transactions. Now if $w_1[x]$ arrives, the scheduler has a problem. If it translates $w_1[x]$ into $w_1[x_1]$ and sends it to the DM, then it produces a history that no longer has the same effect as a TO execution on a single version database. For in such an execution, $r_2[x]$ would have read the value of x written by T_1, but in the execution $w_0[x_0]\ r_2[x_0]\ w_1[x_1]$, it reads the value written by T_0. In this case, we say that $w_1[x]$ would have *invalidated* $r_2[x_0]$. To avoid this problem, the scheduler rejects $w_1[x]$ in this case. In general, it rejects $w_i[x]$ if it has already processed a Read $r_j[x_k]$ such that $ts(T_k) < ts(T_i) < ts(T_j)$. This is exactly the situation in which processing $w_i[x]$ would invalidate $r_j[x_k]$.

To select the appropriate version to read and to avoid invalidating Reads, the scheduler maintains some timestamp information about operations it has already processed. For each version, say x_i, it maintains a timestamp interval, denoted interval(x_i) = [wts, rts], where wts is the timestamp of x_i and rts is the largest timestamp of any Read that read x_i; if no such Read exists, then rts = wts. Let intervals(x) = {interval(x_i) | x_i is a version of x}.

To process $r_i[x]$, the scheduler examines intervals(x) to find the version x_j whose interval, interval(x_j) = [wts, rts], has the maximal wts less than or equal to $ts(T_i)$. If $ts(T_i) >$ rts, then it sets rts to $ts(T_i)$.

To process $w_i[x]$, the scheduler again examines intervals(x) to find the version x_j whose interval [wts, rts] has the maximal wts less than $ts(T_i)$. If rts $> ts(T_i)$, then it rejects $w_i[x]$. Otherwise, it sends $w_i[x_i]$ to the DM and creates a new interval, interval(x_i) = [wts, rts], where wts = rts = $ts(T_i)$.

Eventually, the scheduler will run out of space for storing intervals, or the DM will run out of space for storing versions. At this point, old versions and their corresponding intervals must be deleted. To avoid incorrect behavior, it is essential that versions be deleted from oldest to newest. To see why, consider the following history,

$$H_{11} = w_0[x_0]\ c_0\ r_2[x_0]\ w_2[x_2]\ c_2\ r_4[x_2]\ w_4[x_4]\ c_4,$$

where $ts(T_i) = i$ for $0 \leq i \leq 4$. Suppose the system deleted x_2 but not x_0. If $r_3[x]$ now arrives, the scheduler will incorrectly translate it into $r_3[x_0]$. Suppose instead that the system deleted x_0. If $r_1[x]$ now arrives, the scheduler will find no version whose interval has a wts $< ts(T_1)$. This condition indicates that the DBS has deleted the version that $r_1[x]$ has to read, so the scheduler must reject $r_1[x]$.

*Proof of Correctness

To prove MVTO correct, we must describe it in serializability theory. As usual, we do this by inferring properties that all histories produced by MVTO will satisfy. Using these properties as our formal specification of the algorithm, we prove that all histories produced by MVTO have an acyclic MVSG and hence are 1SR.

The following properties describe the essential characteristics of every MVTO history H over $\{T_0, \ldots T_n\}$.

$MVTO_1$. For each T_i, there is a unique timestamp $ts(T_i)$; that is, $ts(T_i) = ts(T_j)$ iff $i = j$.

$MVTO_2$. For every $r_k[x_j] \in H$, $w_j[x_j] < r_k[x_j]$ and $ts(T_j) \leq ts(T_k)$.

$MVTO_3$. For every $r_k[x_j]$ and $w_i[x_i] \in H$, $i \neq j$, either (a) $ts(T_i) < ts(T_j)$ or (b) $ts(T_k) < ts(T_i)$ or (c) $i = k$ and $r_k[x_j] < w_i[x_i]$.

$MVTO_4$. If $r_j[x_i] \in H$, $i \neq j$, and $c_j \in H$, then $c_i < c_j$.

Property $MVTO_1$ says that transactions have unique timestamps. Property $MVTO_2$ says that each transaction T_k only reads versions with timestamps smaller than $ts(T_k)$. Property $MVTO_3$ states that when the scheduler processes $r_k[x_j]$, x_j is the version of x with the largest timestamp less than or equal to $ts(T_k)$. Moreover, if the scheduler later receives $w_i[x_i]$, it will reject it if $ts(T_j) < ts(T_i) < ts(T_k)$. $MVTO_4$ states that H is recoverable.

These conditions ensure that H preserves reflexive reads-from relationships. To see this, suppose not, that is, $w_k[x_k] < r_k[x_j]$ and $j \neq k$. By $MVTO_2$ and $j \neq k$, $ts(T_j) < ts(T_k)$. By $MVTO_3$, either (a) $ts(T_k) < ts(T_j)$, (b) $ts(T_k) < ts(T_k)$, or (c) $r_k[x_j] < w_k[x_k]$. All three cases are impossible, a contradiction.

We now prove that any history satisfying these properties is 1SR. In other words, MVTO is a correct scheduler.

Theorem 5.5: Every history H produced by MVTO is 1SR.

Proof: Define a version order as follows: $x_i \ll x_j$ iff $ts(T_i) < ts(T_j)$. We now prove that $MVSG(H, \ll)$ is acyclic by showing that for every edge $T_i \rightarrow T_j$ in $MVSG(H, \ll)$, $ts(T_i) < ts(T_j)$.

Suppose $T_i \rightarrow T_j$ is an edge of $SG(H)$. This edge corresponds to a reads-from relationship. That is, for some x, T_j reads x from T_i. By $MVTO_2$, $ts(T_i) \leq ts(T_j)$. By $MVTO_1$, $ts(T_i) \neq ts(T_j)$. So, $ts(T_i) < ts(T_j)$ as desired.

Let $r_k[x_j]$ and $w_i[x_i]$ be in H where i, j, and k are distinct, and consider the version order edge that they generate. There are two cases: (1) $x_i \ll x_j$, which implies $T_i \rightarrow T_j$ is in $MVSG(H, \ll)$; and (2) $x_j \ll x_i$, which implies $T_k \rightarrow T_i$ is in $MVSG(H, \ll)$. In case (1), by definition of $\ll$, $ts(T_i) < ts(T_j)$. In case (2), by $MVTO_3$, either $ts(T_i) < ts(T_j)$ or $ts(T_k) < ts(T_i)$. The first option is impossible, because $x_j \ll x_i$ implies $ts(T_j) < ts(T_i)$. So, $ts(T_k) < ts(T_i)$ as desired.

Since all edges in MVSG(H, $\ll$) are in timestamp order, MVSG(H, $\ll$) is acyclic. By Theorem 5.4, H is 1SR. □

5.4 MULTIVERSION TWO PHASE LOCKING

Two Version 2PL

In 2PL, a write lock on a data item x prevents transactions from obtaining read locks on x. We can avoid this locking conflict by using two versions of x. When a transaction T_i writes into x, it creates a new version x_i of x. It sets a lock on x that prevents other transactions from reading x_i or writing a new version of x. However, other transactions *are* allowed to read the previous version of x. Thus, Reads on x are not delayed by a concurrent writer of x. In the language of Section 4.5, we are using 2PL for ww synchronization and version selection for rw synchronization. As we will see, there is also certification activity involved.

To use this scheme, the DM must store one or two versions of each data item. If a data item has two versions, then only one of those versions was written by a committed transaction. Once a transaction T_i that wrote x commits, its version of x becomes the unique committed version of x, and the previous committed version of x becomes inaccessible.

The two versions of each data item could be the same two versions used by the DM's recovery algorithm. If T_i wrote x but has not yet committed, then the two versions of x are T_i's before image of x and the value of x it wrote. As we will see, T_i's before image can be deleted once T_i commits. So, an old version becomes dispensable for both concurrency control and recovery reasons at approximately the same time.[7]

A *two version 2PL (2V2PL)* scheduler uses three types of locks: read locks, write locks, and certify locks. These locks are governed by the compatibility matrix in Fig. 5–1. The scheduler sets read and write locks at the usual time, when it processes Reads and Writes. When it learns that a transaction has terminated and is about to commit, it converts all of the transaction's write locks into certify locks. We will explain the handling and significance of certify locks in a moment.

When a 2V2PL scheduler receives a Write, $w_i[x]$, it attempts to set $wl_i[x]$. Since write locks conflict with certify locks and with each other, the scheduler delays $w_i[x]$ if another transaction already has a write or certify lock on x. Otherwise, it sets $wl_i[x]$, translates $w_i[x]$ into $w_i[x_i]$, and sends $w_i[x_i]$ to the DM.

When the scheduler receives a Read, $r_i[x]$, it attempts to set $rl_i[x]$. Since read locks only conflict with certify locks, it can set this lock as long as no

[7] This fits especially well with the shadow page recovery techniques used, for example, in the no-undo/no-redo algorithm of Section 6.7.

	Read	Write	Certify
Read	y	y	n
Write	y	n	n
Certify	n	n	n

FIGURE 5-1
Compatibility Matrix for Two Version 2PL

transaction already owns a certify lock on x. If T_i already owns $wl_i[x]$ and has therefore written x_i, then the scheduler translates $r_i[x]$ into $r_i[x_i]$, which it sends to the DM. Otherwise, it waits until it can set a read lock, and then sets the lock, translates $r_i[x]$ into $r_i[x_j]$, where x_j is the most recently (and therefore only) committed version of x, and sends $r_i[x_j]$ to the DM. Note that since only committed versions may be read (except for versions produced by the reader itself), the scheduler avoids cascading aborts and, *a fortiori*, ensures that the MV histories it produces are recoverable.

When the scheduler receives a Commit, c_i, indicating that T_i has terminated, it attempts to convert T_i's write locks into certify locks. Since certify locks conflict with read locks, the scheduler can only do this lock conversion on those data items that have no read locks owned by other transactions. On those data items where such read locks exist, the lock conversion is delayed until all read locks are released. Thus, the effect of certify locks is to delay T_i's commitment until there are no active readers of data items it is about to overwrite.

Note that lock conversions can lead to deadlock just as in standard 2PL. For example, suppose T_i has a read lock on x and T_j has a write lock on x. If T_i tries to convert its read lock to a write lock and T_j tries to convert its write lock to a certify lock, then the transactions are deadlocked. We can use any deadlock detection or prevention technique: cycle detection in a WFG, timestamp-based prevention, etc.

Since a transaction may deadlock while trying to convert its write locks, it may be aborted during this activity. Therefore, it must not release its locks or be committed until it has obtained all of its certify locks.

Certify locks in 2V2PL behave much like write locks in ordinary 2PL. Since the time to certify a transaction is usually much less than the total time to execute it, 2V2PL's certify locks delay Reads for less time than 2PL's write locks delay Reads. However, since existing read locks delay a transaction's certification in 2V2PL, the improved concurrency of Reads comes at the expense of delaying the certification and therefore the commitment of update transactions.

Using More than Two Versions

The only purpose served by write locks is to ensure that only two versions of a data item exist at a time. They are not needed to attain 1-serializability. If we relax the conflict rules so that write locks do not conflict, then a data item may have many *uncertified* versions (i.e., versions written by uncommitted transactions). However, if we follow the remaining 2V2PL locking rules, then only the most recently certified version may be read.

If we are willing to cope with cascading aborts, then we can be a little more flexible by allowing a transaction to read any of the uncertified versions. (We could make the same allowance in 2V2PL, in which there is at most one uncertified version to read.) To get the same correct synchronization behavior as 2V2PL, we have to modify the scheduler in two ways. First, a transaction cannot be certified until all of the versions it read (except for ones it wrote itself) have been certified. And second, the scheduler can only convert a write lock on x into a certify lock if there are no read locks on *certified* versions of x.

Essentially, the scheduler is ignoring a read lock on an uncertified version until either that version is certified or the transaction that owns the read lock tries to become certified. This is just like delaying the granting of that read lock until after the version to be read is certified. The only difference is that cascading aborts are now possible. If the transaction that produced an uncertified version aborts, then transactions that read that version must also abort.

*Correctness of 2V2PL

To list the properties of histories produced by executions of 2V2PL, we need to include the operation f_i, denoting the certification of T_i.

Let H be a history over $\{T_0, \ldots T_n\}$ produced by 2V2PL. Then H must satisfy the following properties.

2V2PL$_1$. For every T_i, f_i follows all of T_i's Reads and Writes and precedes T_i's commitment.

2V2PL$_2$. For every $r_k[x_j]$ in H, if $j \neq k$, then $c_j < r_k[x_j]$; otherwise $w_k[x_k] < r_k[x_k]$.

2V2PL$_3$. For every $w_k[x_k]$ and $r_k[x_j]$ in H, if $w_k[x_k] < r_k[x_j]$, then $j = k$.

Property 2V2PL$_2$ says that every Read $r_k[x_j]$ either reads a certified version or reads a version written by itself (i.e., T_k). Property 2V2PL$_3$ says that if T_k wrote x before the scheduler received $r_k[x]$, then it translates $r_k[x]$ into $r_k[x_k]$.

2V2PL$_4$. If $r_k[x_j]$ and $w_i[x_i]$ are in H, then either $f_i < r_k[x_j]$ or $r_k[x_j] < f_i$.

Property 2V2PL$_4$ says that $r_k[x_j]$ is strictly ordered with respect to the certification operation of every transaction that writes x. This is because each

transaction T_i that writes x must obtain a certify lock on x. For each transaction T_k that reads x, either T_i must delay its certification until T_k has been certified (if it has not already been so), or else T_k must wait for T_i to be certified before it can set its read lock on x and therefore read x.

2V2PL$_5$. For every $r_k[x_j]$ and $w_i[x_i]$ (i, j, and k distinct), if $f_i < r_k[x_j]$, then $f_i < f_j$.

Property 2V2PL$_5$, combined with 2V2PL$_2$, says that each Read $r_k[x_j]$ either reads a version written by T_k or reads the most recently certified version of x.

2V2PL$_6$. For every $r_k[x_j]$ and $w_i[x_i]$, $i \neq j$ and $i \neq k$, if $r_k[x_j] < f_i$, then $f_k < f_i$.

Property 2V2PL$_6$ says that a transaction T_i that writes x cannot be certified until all transactions that previously read a version of x have already been certified. This follows from the fact that certify locks conflict with read locks.

2V2PL$_7$. For every $w_i[x_i]$ and $w_j[x_j]$, either $f_i < f_j$ or $f_j < f_i$.

Property 2V2PL$_7$ says that the certification of every two transactions that write the same data item are atomic with respect to each other.

Theorem 5.6: Every history H produced by a 2V2PL scheduler is 1SR.

Proof: By 2V2PL$_1$, 2V2PL$_2$ and 2V2PL$_3$, H preserves reflexive reads-from relationships and is recoverable, and therefore is an MV history. Define a version order $\ll$ by $x_i \ll x_j$ only if $f_i < f_j$. By 2V2PL$_7$, $\ll$ is indeed a version order. We will prove that all edges in MVSG(H, $\ll$) are in certification order. That is, if $T_i \rightarrow T_j$ in MVSG(H, $\ll$), then $f_i < f_j$.

Let $T_i \rightarrow T_j$ be in SG(H). This edge corresponds to a reads-from relationship, such as T_j reads x from T_i. By 2V2PL$_2$, $f_i < r_j[x_i]$. By 2V2PL$_1$, $r_j[x_i] < f_j$. Hence, $f_i < f_j$.

Consider a version order edge induced by $w_i[x_i]$, $w_j[x_j]$, and $r_k[x_j]$ (i, j, and k distinct). There are two cases: $x_i \ll x_j$ and $x_j \ll x_i$. If $x_i \ll x_j$, then the version order edge is $T_i \rightarrow T_j$, and $f_i < f_j$ follows directly from the definition of $\ll$. If $x_j \ll x_i$, then the version order edge is $T_k \rightarrow T_i$. Since $x_j \ll x_i$, $f_j < f_i$. By 2V2PL$_4$, either $f_i < r_k[x_j]$ or $r_k[x_j] < f_i$. In the former case, 2V2PL$_5$ implies $f_i < f_j$, contradicting $f_j < f_i$. Thus $r_k[x_j] < f_i$, and by 2V2PL$_6$, $f_k < f_i$ as desired.

This proves that all edges in MVSG(H, $\ll$) are in certification order. Since the certification order is embedded in a history, which is acyclic by definition, MVSG(H, $\ll$) is acyclic too. So, by Theorem 5.4, H is 1SR. □

5.5 A MULTIVERSION MIXED METHOD

As we have seen, multiversions give the scheduler more flexibility in scheduling Reads. If the scheduler knows in advance which transactions will *only* issue Reads (and no Writes), then it can get even more concurrency among transactions. Recall from Section 1.1 that transactions that issue Reads but no Writes are called *queries*, while those that issue Writes (and possibly Reads as well) are called *updaters*. In this section we'll describe an algorithm that uses MVTO to synchronize queries and Strict 2PL to synchronize updaters.

When a transaction begins executing, it tells its TM whether it's an updater or a query. If it's an updater, then the TM simply passes that fact to the scheduler, which executes operations from that transaction using Strict 2PL. When the TM receives the updater's Commit, indicating that the updater has terminated, the TM assigns a timestamp to the updater, using the timestamp generation method of Section 4.5 for integrating Strict 2PL and TWR. This ensures that updaters have timestamps that are consistent with their order in the SG.

Unlike Basic 2PL, in this method each Write produces a new version. The scheduler tags each version with the timestamp of the transaction that wrote it. The scheduler uses these timestamped versions to synchronize Reads from queries using MVTO.

When a TM receives the first operation from a transaction that identified itself as a query, it assigns to that query a timestamp smaller than that of any committed or active updater (and therefore, of any future updater as well). When a scheduler receives an $r_i[x]$ from a query T_i, it finds the version of x with the largest timestamp less than $ts(T_i)$. By the timestamp assignment rule, this version was written by a committed transaction. Moreover, by the same rule, assigning this version of x to $r_i[x]$ will not invalidate the Read at any time in the future (so future Writes need never be rejected).

Note that a query does not set any locks. It is therefore never forced to wait for updaters and never causes updaters to wait for it. The scheduler can always process a query's Read without delay.

In a centralized DBS, selecting the timestamp of a query is easy, because active updaters are not assigned timestamps until they terminate. In a distributed DBS, TMs can ensure that each query has a sufficiently small timestamp by deliberately selecting an *especially* small timestamp. Suppose that the local clocks at any two TMs are known to differ by at most δ. If a TM's clock reads t, then it is safe to assign a new query any timestamp less than $t - \delta$. Any updater that terminates after this point in real time will be assigned a timestamp of at least $t - \delta$, so the problem of the previous paragraph cannot arise.

We can argue the correctness of this by using MVSGs as follows.[8] Let H be a history produced by the method. Define the version order for H as in

[8]This paragraph requires an understanding of Section 5.2, on Multiversion Serializability Theory, a starred section.

MVTO: $x_i \ll x_j$ iff $ts(T_i) < ts(T_j)$. We show that MVSG(H, $\ll$) is acyclic by showing that for each of its edges $T_i \rightarrow T_j$, $ts(T_i) < ts(T_j)$. First, consider an edge $T_i \rightarrow T_j$ in SG(H). Each such edge is due to a reads-from relationship. If T_j is an updater, then by the way timestamps are assigned to updaters, $ts(T_i)$; $< ts(T_j)$ (cf. Section 4.5). If T_j is a query, then by MVTO version selection, $ts(T_i) < ts(T_j)$. Now consider a version order edge in MVSG(H, $\ll$) that arises because T_j reads x from T_i and T_k writes x (i, j, k distinct). If $x_k \ll x_i$, then we have the edge $T_k \rightarrow T_i$ in MVSG(H, $\ll$) and $ts(T_k) < ts(T_i)$. Otherwise, we have the edge $T_j \rightarrow T_k$, so we must show $ts(T_j) < ts(T_k)$. If T_j is an updater, then T_j released $rl_j[x]$ before T_k obtained $wl_k[x]$, so by the timestamp assignment method, $ts(T_j) < ts(T_k)$. If T_j is a query, then it is assigned a timestamp smaller than all active or future updaters. So again $ts(T_j) < ts(T_k)$. Thus, all edges in MVSG(H, $\ll$) are in timestamp order, and MVSG(H, $\ll$) is acyclic. By Theorem 5.4, H is 1SR.

To avoid running out of space, the scheduler must have a way of deleting "old" versions. Any committed version may be eliminated as soon as the scheduler can be assured that no query will need to read that copy in the future. For this reason, the scheduler maintains a non-decreasing value *min*, which is the minimum timestamp that can be assigned to a query. Whenever the scheduler wants to release some space used by versions, it sets min to be the smallest timestamp assigned to any active query. It can then discard a committed version x_i if $ts(T_i) <$ min and there is another committed version x_j, such that $ts(T_i) < ts(T_j)$.

The main benefit of this method is that queries and updaters never delay each other. A query can always read the data it wants without delay. Although updaters may delay each other, queries set no locks and therefore never delay updaters. This is in sharp contrast to 2PL, where a query may set many locks and thereby delay many updaters. This delay is also inherent in multiversion 2PL and 2V2PL, since an updater T_i cannot commit until there are no read locks held by other transactions on T_i's writeset.

The main disadvantages of the method are that queries may read out-of-date data and that the tagging and interpretation of timestamps on versions may add significant scheduling overhead. Both problems can be mitigated by using the methods described next.

Replacing Timestamps by Commit Lists

Tagging versions with timestamps may be costly because when a scheduler processes $w_i[x]$ by creating a new version of x, it doesn't know T_i's timestamp. Only after T_i terminates can the scheduler learn T_i's timestamp. However, by this time, the version may already have been moved to disk; it needs to be reread in order to be tagged, and then subsequently rewritten to disk.

One can avoid timestamps altogether by using instead a list of identifiers of committed transactions, called the *commit list*. When a query begins execut-

ing, the TM makes a copy of the commit list and associates it with the query. It attaches the commit list to every Read that it sends to the scheduler, essentially treating the list like a timestamp. When the scheduler receives $r_i[x]$ for a query T_i, it finds the most recently committed version of x whose tag is in T_i's copy of the commit list. To do this efficiently, all versions of a data item are kept in a linked list, from newest to oldest. That is, whenever a new version is created, it is added to the *top* of the version list. Since updaters use Strict 2PL, two transactions may not concurrently create new versions of the same data item. Thus, the order of a data item's versions (and hence the version list) is well defined.

Given this organization for versions, to process $r_i[x]$ for a query T_i, the scheduler scans the version list of x until it finds a version written by a transaction that appears in the commit list associated with T_i. This is just like reading the most recently committed version of x whose timestamp is less than ts(T_i) (if T_i had a timestamp). This technique is used in DBS products by Prime Computer, and in the Adaplex DBS by Computer Corporation of America.

The problem with this scheme is the size and structure of commit lists. First, each list must be small. In a centralized system, every query will have a copy of the list consuming main memory. In a distributed system, every Read sent to a DM will have a copy of the list, which consumes communication bandwidth. Second, since the scheduler must search the list on every Read from a query, the list should be structured to make it easy to determine whether a given transaction identifier is in the list.

A good way to accomplish these goals is to store the commit list as a bit map. That is, the commit list is an array, CL, where CL[i] = 1 if T_i is committed; otherwise CL[i] = 0. Using the bit map, the scheduler can easily tell whether a version's tag is in the list. It simply looks up the appropriate position in the array. However, as time goes on, the list grows without limit. So we need a way to keep the list small.

We can shorten the list by observing that old transaction identifiers eventually become useless. A transaction identifier is only needed as long as there is a version whose tag is that identifier. Suppose we know that all versions whose tags are less than n (where n is a transaction identifier) have either been committed or discarded before all active queries began. Then when the scheduler reads a version whose tag is less than n, it may assume that n is in the commit list. Only transactions whose identifiers are greater than or equal to n need to be kept in the list.

The commit list can be kept short as follows. When the list has exceeded a certain size, the scheduler asks the TM for a transaction identifier, n, that is smaller than that which has been assigned to any active query or updater, or will be assigned to any future query or updater. The scheduler can then discard the prefix of the commit list through transaction identifier n, thereby shortening the list. To process $r_i[x]$ of some query T_i, the scheduler returns the first version in the version list of x written by a transaction whose identifier is either in, or smaller than any identifier in, the commit list given to T_i when it started.

We are assuming here, as always, that when a transaction aborts, all versions it has produced are removed from the version lists.

When the scheduler receives n from the TM for the purpose of reducing the size of the commit list, it can also garbage collect versions. In particular, it can discard a committed version of x, provided there is a more recent committed version of x whose identifier is less than n.

Distributed Commit Lists

In a distributed DBS, using a commit list in place of timestamps requires special care, because the commit lists maintained at different sites may not be instantaneously identical. For example, suppose an updater T_1 commits at site A, where it updated x, and is added to CL_A, the commit list at site A; but suppose T_1 has not yet committed at site B, where it updated y. Next, suppose an updater T_2 starts executing at A, reads the version of x written by T_1, writes a new version of z at site B, and commits, thereby adding its transaction identifier to CL_A and CL_B. (T_1 still hasn't committed at site B). Now suppose a query starts executing at site B, reads CL_B (which contains T_2 but not T_1), and reads y and z at site B. It will read the version of z produced by T_2 (which read x from T_1) but not the version of y produced by T_1. The result is not 1SR.

We can avoid this problem by ensuring that whenever a commit list at a site contains a transaction T_i, then it also contains all transactions from which T_i read a data item (at the same site or any other site). To do this, before an updater transaction T_j commits, it reads the commit lists at all sites where it read data items and takes the union of those commit lists along with $\{T_j\}$, producing a temporary commit list CL_{temp}. Then, instead of merely adding T_j to the commit list at every site where it wrote, it unions CL_{temp} into those commit lists. Using this method in the example of the previous paragraph, T_2 would read CL_A, which includes T_1, and would union it into CL_B. The query that reads CL_B now reads T_1's version of y, as required to be 1SR.

Using this method, each query reads a database that was effectively produced by a serial execution of updaters. However, executions may not be 1SR in the sense that two different queries may see mutually inconsistent views. For example, suppose TM_1 and TM_2 supervise the execution of queries T_1 and T_2, respectively, both of which read data items x and y stored, respectively, at sites A and B. Consider now the following sequence of events:

1. TM_1 reads CL_A.
2. TM_2 reads CL_B.
3. T_3 writes x at site A and commits, thereby adding T_3 to CL_A.
4. T_4 writes y at site B and commits, thereby adding T_4 to CL_B.
5. TM_1 reads CL_B.
6. TM_2 reads CL_A.

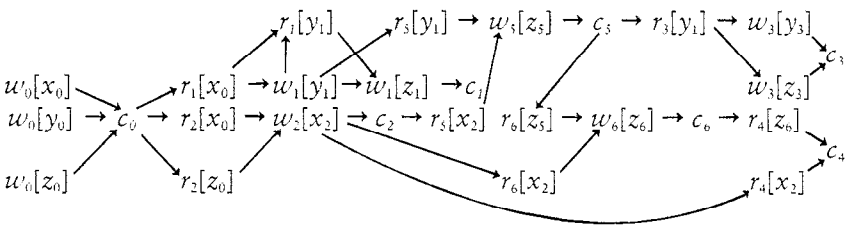
Now, T_1 reads a database state that includes T_4's Write on y but not T_3's Write on x, while T_2 reads a database state that includes T_3's Write on x but not T_4's Write on y. Thus, from T_1's viewpoint, transactions executed in the order $T_4\ T_1\ T_3$, but from T_2's viewpoint, transactions executed in the order $T_3\ T_2\ T_4$. There is no serial 1V history including all four transactions that is equivalent to this execution. Yet, the execution consisting only of updaters is 1SR, and in a sense, each query reads consistent data. We leave the proof of these properties as an exercise (see Exercise 5.22).

BIBLIOGRAPHIC NOTES

The serializability theoretic model of multiversion concurrency control is from [Bernstein, Goodman 83]. Other theoretical aspects are explored in [Hadzilacos, Papadimitriou 85], [Ibaraki, Kameda 83], [Lausen 83], and [Papadimitriou, Kanellakis 84]. The two version 2PL algorithm in Section 5.3 is similar to that of [Stearns, Rosenkrantz 81], which uses timestamp-based deadlock prevention. A similar method that uses SGT certification for rw synchronization is described in [Bayer et al. 80] and [Bayer, Heller, Reiser 80]. A multiversion tree locking algorithm appears in [Silberschatz 82]. Multiversion TO was introduced in [Reed 78], [Reed 79], and [Reed 83]. Multiversion mixed methods like those in Section 5.5 are described in [Bernstein, Goodman 81], [Chan et al. 82], [Chan, Gray 85], [Dubourdieu 82], and [Weihl 85]. [Dubourdieu 82] describes a method used in a product of Prime Computer. [Lai, Wilkinson 84] describes a multiversion 2PL certifier, where queries are never delayed, and each updater T_i is certified by checking its readset and writeset against the writeset of all transactions that committed after T_i starts.

EXERCISES

5.1* Consider the following history:



a. Prove that this satisfies the definition of MV history.
b. Is this history serializable?
c. Is it one-copy serializable? If so, give a version order that produces an acyclic MVSG.
d. Suppose we add the operation $r_4[y_3]$ (where $w_3[y_3] < r_4[y_3]$) to the history. Answer (c) for this new history.

5.2* Give a careful proof of the fact that if H is an MV history then $C(H)$ is a complete MV history. Suppose in the definition of MV histories we

required only conditions (1) – (4), but not recoverability. Prove that in that case, C(H) would not necessarily be a complete MV history. (Incidentally, this is the reason for making recoverability part of the *definition* of MV histories, whereas in 1V serializability theory we treated recoverability as a property that some histories have and others do not.)

5.3* Prove Proposition 5.2.

5.4* Prove that if H is a 1SR MV history, then so is any prefix of H.

5.5* Suppose no transaction ever reads a data item that it previously wrote. Then we can redefine MV history, such that it need not preserve reflexive reads-from relationships (since they cannot exist). Using this revised definition prove Theorem 5.3, making as many simplifications as possible.

5.6 MVTO can reject transactions whose Writes arrive too late. Design a conservative MVTO scheduler that never rejects Reads or Writes. Prove it correct. To show why your conservative MVTO is not worse than single version conservative TO, characterize those situations in which the latter will delay operations while the former will not. Are there situations where the opposite is true?

5.7 In MVTO, suppose that we store timestamp intervals in the data items themselves rather than in a separate table. For example, suppose the granularity of data items is a fixed size page and that each page has a header containing timestamp interval information. How does this organization affect the efficiency with which the MVTO scheduler processes operations? How does it affect the way the scheduler garbage collects old versions?

5.8 Since MVTO doesn't use locks, we need to add a mechanism for preventing transactions from reading uncommitted data and thereby avoiding cascading aborts. Propose such a mechanism. How much concurrency do you lose through this mechanism? Compare the amount of concurrency you get with the one you proposed for Exercise 5.6.

5.9 Show that there does or does not exist a sequence of Reads and Writes in which

a. Basic TO rejects an operation and MVTO does not;
b. Basic TO delays an operation and MVTO does not;
c. MVTO rejects an operation and Basic TO does not; and
d. MVTO delays an operation and Basic TO does not.

That is, for each situation, either give an example sequence with the desired property, or prove that such a sequence does not exist.

5.10 Modify MVTO so that it correctly handles transactions that write into a data item more than once.

5.11 Describe the precise conditions under which MVTO can safely discard a version without affecting any future transaction.

5.12 It is incorrect to use MVTO for rw synchronization and TWR for ww synchronization. Explain why.

5.13 Assume no transaction ever reads a data item that it previously wrote. Consider the following variation of standard 2PL, called *2PL with delayed writes*. Each TM holds all writes used by a transaction until the transaction terminates. It then sends all those held Writes to the appropriate DMs. DMs use standard 2PL. Compare the behavior of 2V2PL to 2PL with delayed writes.

5.14* Let H be the set of all 1V histories equivalent to the MV histories produced by 2V2PL. Is H identical to the set of histories produced by 2PL? Prove your answer.

5.15 Suppose we modify multiversion 2PL as follows. As in Section 5.5, we distinguish queries from updaters. Updaters set certify locks in the usual way. Queries set no (read) locks. To read a data item x, a query reads the most recently certified version of x. Does this algorithm produce 1SR executions? If so, prove it. If not, give a counterexample.

5.16 Suppose no transaction ever reads a data item that it previously wrote. Use this knowledge to simplify the 2V2PL algorithm. Does your simplification improve performance?

5.17 Show how to integrate timestamp-based deadlock prevention into 2V2PL. If most write locks will eventually be converted into certify locks (i.e., if very few transactions spontaneously abort), is it better to perform the deadlock prevention early using write locks or later using certify locks?

5.18* Prove the correctness of the extension to 2V2PL that uses more than two versions, described at the end of Section 5.4.

5.19 Compare the behavior of the multiple version extension to 2V2PL to standard 2V2PL. How would you expect them to differ in the number of delays and aborts they induce?

5.20* Prove that the mixed method of Strict 2PL and "MVTO" that uses commit lists for queries in a centralized DBS (in Section 5.5) is correct.

5.21 Consider the distributed Strict 2PL and "MVTO" mixed method in Section 5.5 that uses commit lists for queries. The method only guarantees that any execution of updaters is 1SR, and that each query reads consistent data. Propose a modification to the algorithm that ensures that queries do not read mutually inconsistent data; that is, any execution of updaters and queries is 1SR. Compare the cost of your method to the cost of the one in the chapter.

5.22* Prove that the distributed Strict 2PL and "MVTO" mixed method in Section 5.5 that uses commit lists for queries is correct, in the sense that any execution of updaters and one query is 1SR.

5.23 Design a multiversion concurrency control algorithm that uses SGT certification for rw synchronization and 2PL for ww synchronization. Prove that your algorithm is correct.