Chapter 5

Multiversion Concurrency Control

"A book is a version of the world. If you do not like it, ignore it; or offer your own version in return." (Salmon Rushdie)

5.1 Goal and Overview of the Chapter

In this chapter we will consider an aspect of transaction processing that is both theoretically attractive as well as practically relevant. So far the underlying assumption has been that data items exist in exactly one copy each. As a result, write operations overwrite existing data items, and read-from relationships between distinct transactions may fail to observe the transactional isolation requirement. Now imagine *more than one* copy of each data item were allowed! This would, for example, render it possible to keep the "old" version of a data item that is subject to overwriting, at least until the transaction that writes a "new" version commits. More generally, distinct transactions could be given distinct versions of the same data item to read, or to overwrite.

In this chapter we will study the impact of allowing multiple versions on the various components of the framework we have developed so far: schedules, serializability, concurrency control protocols. It will turn out that versioning demands some reasonable extensions of serializability theory, and, as will be seen in Part III, is connected to recovery, as versions of modified data ("before images" or "after images", or both) are sometimes maintained in some form by a recovery manager anyway. We will additionally look at the storage space needed to hold multiple versions. While theoretical considerations tend to assume space to be an unlimited

resource, it turns out that the picture changes significantly should this not be valid anymore. In particular, we will study how multiversion serializability is affected by limiting the (total) number of versions around.

An important assumption we will make in this chapter is that versioning is *transparent* to the outside world. In other words, users or applications will not be aware of the fact that data items can appear in multiple versions. We mention that such an assumption is different from versioning in design environments such as CAD (Computer-Aided Design) or CASE (Computer-Aided Software Engineering), where database users need to "play around" with versions of data items in order to develop, change, alternate, or experiment with their designs. This type of versioning that users are aware of is also of increasing importance in the context of *repositories* for change and configuration management.

In this book we restrict ourselves to transparent versioning of data items. We will first derive, in Section 5.2, a suitable abstraction for such a setting from the read-write model. Intuitively, the idea is that, because of the transparency requirement, correctness of histories and schedules "looks and feels" as if versioning were not applied, at least from a user's perspective. In Section 5.3, we will generalize our serializability notions to the augmented setting. After that, in Section 5.4 we take an in-depth look at the assumption, made in the very beginning, that the number of versions a system is capable of keeping around is unbounded; we will indicate how the picture changes if the total number of versions stored is limited. Finally, in Section 5.5 we look at concurrency control protocols and generalize some of those presented in the previous chapter to the management of versions.

The discussion in this chapter continues to consider unaborted transactions only.

5.2 Multiversion Schedules

In order to motivate the studies of this chapter more concretely, we start with an example that shows that the presence of multiple versions of data items can indeed affect the correctness of histories or schedules.

Example 5.1

Consider the following history:

$$s = r_1(x)w_1(x)r_2(x)w_2(y)r_1(y)w_1(z)c_1c_2$$

Obviously, $s \notin$ CSR, since there is a cyclic conflict between t_1 and t_2. Informally, the reason for this is the fact that y has already been overwritten by t_2 when step $r_1(y)$ arrives at the scheduler; in other words, $r_1(y)$ is "too late" for making the schedule that already has a conflict (on x) between t_1 and t_2 an acceptable one.

Now suppose that the old value of y (in this case the initial value of y) was still available when $r_1(y)$ arrives. Then it would be possible to create a history equivalent to

$$s' = r_1(x)w_1(x)r_1(y)r_2(x)w_2(y)w_1(z)c_1c_2$$

which would be correct since $s' \in$ CSR. □

The existence of multiple versions of data items is based on the assumption that write operations are no longer executed "in–place". Instead, the result of each and every write operation is individually kept as a new version of the data item in question. In Example 5.1, this would imply that old values would still be accessible for read operations that arrive "too late". Clearly, a scheduler would still have to decide which version of a data item is read by a particular read operation. We will look at this and other questions in what follows.

Our first goal will be an appropriate formalization of histories and schedules in the presence of multiple versions of data items. Since we are concerned with transparent versioning only, the transaction model can remain unchanged; only the interpretation of a transaction changes. Basically, a read step of the form $r(x)$ reads an (existing) version of x, and a write step of the form $w(x)$ (always) creates a new version of x (or overwrites an existing one). For simplicity of notation, we assume that each transaction writes every data item at most once; thus, if t_j contains operation $w_j(x)$, we can denote the version of x created by this write by x_j. Now we can define:

version function

Definition 5.2 (Version function)

Let s be a history with initialization transaction t_0 and final transaction t_∞. A *version function* for s is a function h which associates with each read step of s a previous write step on the same data item, and which is the identity on write steps.

Since we have already assumed that a step $w_j(x)$ creates version x_j of x, we may write the following when using a version function h in the context of a schedule s:

1. $h(r_i(x)) = w_j(x)$ for some $w_j(x) <_s r_i(x)$, and $r_i(x)$ reads x_j,
2. $h(w_i(x)) = w_i(x)$, and $w_i(x)$ writes x_i.

Thus, $h(r_i(x))$ essentially denotes the version of x that has been assigned to be read by $r_i(x)$. In slight abuse of the notation, we also write $w_j(x_j)$ for the write step that creates x_j, and we write $r_i(x_j)$ for the read step that has been associated with

$w_j(x_j)$ by version function h. So in the sequel, the result of applying a version function to a read operation could be any of (1) the associated write operation, (2) the version assigned for reading, or (3) the "versioned" read operation. Since all three possibilities are basically equivalent, this should not cause any confusion.

A version function translates each write step into a version-creation step, and each read step into a version-read step. We are now ready to introduce the formal notion of a multiversion schedule, which is essentially a schedule (in the sense used so far) together with a version function:

multiversion schedule

Definition 5.3 (Multiversion schedule)

Let $T = \{t_1, \dots, t_n\}$ be a (finite) set of transactions.

(i) A *multiversion history* (or complete multiversion schedule) for T is a pair $m = (\mathrm{op}(m), <_m)$, where $<_m$ is an order on $\mathrm{op}(m)$ and

(1) $\mathrm{op}(m) = h(\bigcup_{i=1}^{n} \mathrm{op}(t_i))$ for some version function h (here we assume that h has been canonically extended from single operations to sets of operations),

(2) for all $t \in T$ and all operations $p, q \in \mathrm{op}(t)$ the following holds:
$p <_t q \Rightarrow h(p) <_m h(q)$,

(3) if $h(r_j(x)) = r_j(x_i)$, $i \neq j$, and c_j is in m, then c_i is in m and $c_i <_m c_j$.

(ii) A *multiversion schedule* is a prefix of a multiversion history.

In words, a multiversion history contains "versioned" read and write operations for its transactions whose ordering respects the individual transaction orderings. We mention that Condition (i)(3) is needed here since otherwise CP(m), the committed projection of a schedule m, is not necessarily complete (i.e., not a history).

Example 5.4

The following is a totally ordered multiversion schedule for the history from Example 5.1 with a version function h that assigns, for example, version y_0 to the read step $r_1(y)$, i.e., $h(r_1(y)) = w_0(y)$ where t_0 is the initialization transaction (which is not mentioned explicitly in the schedule).

$$m = r_1(x_0)w_1(x_1)r_2(x_1)w_2(y_2)r_1(y_0)w_1(z_1)c_1c_2$$

□

Conventional schedules, as we have discussed them in the previous chapters, can be viewed as a special case of multiversion schedules. Such a "monoversion" schedule is obtained by restricting the version function of a multiversion schedule so that each read step reads the version produced by the *last* preceding write step. In other words, the effect of a monoversion schedule is the same as if only a single version of each data item is kept.

Definition 5.5 (Monoversion schedule)

A multiversion schedule is called a monoversion schedule if its version function maps each read step to the last preceding write step on the same data item.

An example of a monoversion schedule is:

$$m = r_1(x_0)w_1(x_1)r_2(x_1)w_2(y_2)r_1(y_2)w_1(z_1)c_1c_2$$

Notice that the example differs from the previous multiversion example only in that now we have $h(r_1(y)) = w_2(y)$. Because the version function of a monoversion schedule is uniquely determined by the order of steps, we can also write m as if it were a conventional schedule, omitting version subscripts:

$$s = r_1(x)w_1(x)r_2(x)w_2(y)r_1(y)w_1(z)c_1c_2$$

In the sequel we will often use monoversion schedules as in previous chapters as well as multiversion schedules in the sense just introduced; in order to be able to distinguish them, we will continue to use s, s' etc. for the former, and we will use m, m' etc. for the latter.

5.3 Multiversion Serializability

We next study various types of multiversion serializability. To this end, we will begin by following our previous practice, namely devise a notion of view serializability and investigate whether it is feasible. Not surprisingly, it will again exhibit an NP-complete decision problem, which motivates to look for a notion of conflict serializability. In addition, another important aspect will be to limit the number of versions that can be stored, and hence kept around, at the same time; as we will see in the next Section 5.4, such a limit drives home an entire hierarchy of classes of serializable schedules.

5.3.1 Multiversion View Serializability

As we have mentioned already, when making serializability precise for multiversion schedules we have to keep in mind that we are here talking about *transparent* versioning, i.e., versioning that is not visible to the outside world, in particular to application programs or even the user. Thus, from a user's point of view a "correct" multiversion schedule should be equivalent to an ordinary serial schedule without versions. To make equivalence precise, conflict equivalence of the usual style (i.e., according to Chapter 3) is not an appropriate candidate, as the following example indicates.

Example 5.6

Consider the serial (monoversion) schedule $s = w_0(x)c_0w_1(x)c_1r_2(x)w_2(y)c_2$. A multiversion schedule exhibiting the same "step syntax" is

$$m = w_0(x_0)c_0w_1(x_1)c_1r_2(x_0)w_2(y_2)c_2.$$

Notice that while s shows three different conflicts (between $w_0(x)$ and $w_1(x)$, between $w_0(x)$ and $r_2(x)$, and between $w_1(x)$ and $r_2(x)$), m exhibits only one conflict, the one between $w_0(x_0)$ and $r_2(x_0)$, since t_1 now writes a new version of x. Although we could define that a multiversion schedule m is conflict equivalent to a serial monoversion schedule s if all pairs of conflicting operations occur in m in the same order as in s (so that the two schedules shown above are indeed equivalent), the problem remains that the respective reads-from relations may be different, which is intuitively undesirable. Already in this example we find that in s transaction t_2 reads data item x from t_1, while in m transaction t_2 reads the version of x produced by t_0; in other words, under the no-interpretation schedule semantics adopted so far (i.e., the Herbrand semantics, see Chapter 3), we should assume that both schedules yield distinct values of item y.
□

Considerations like the one just shown motivate the use of view equivalence for defining serializability in this context, suitably adapted to the presence of versions.

reads-from relation

Definition 5.7 (Reads-from relation)

Let m be a multiversion schedule, $t_i, t_j \in \text{trans}(m)$.

The *reads-from relation* of m is defined by

$$\text{RF}(m) := \{(t_i, x, t_j) \mid r_j(x_i) \in \text{op}(m)\}$$

Now, following our general methodology from Chapter 3 in developing correctness criteria, we can directly obtain an appropriate notion of equivalence based on the reads-from relations of schedules.

view equivalence of multiversion schedules

Definition 5.8 (View equivalence)

Let m and m' be two multiversion schedules s.t. $\text{trans}(m') = \text{trans}(m)$. m and m' are *view equivalent*, abbreviated $m \approx_v m'$, if $\text{RF}(m) = \text{RF}(m')$.

A difference between this notion of view equivalence and the one previously introduced in Chapter 3 for monoversion schedules is that a consideration of final write operations is now irrelevant, for the simple reason that two multiversion schedules with identical sets of transactions *always* have the same final writes (since writes produce versions that are not erased any more). As will be seen shortly, this statement no longer holds, however, if the number of versions that can be stored is limited.

Example 5.9

Let $m = w_0(x_0)w_0(y_0)c_0r_3(x_0)w_3(x_3)c_3w_1(x_1)c_1r_2(x_1)w_2(y_2)c_2$ and
$m' = w_0(x_0)w_0(y_0)c_0w_1(x_1)c_1r_2(x_1)r_3(x_0)w_2(y_2)w_3(x_3)c_3c_2$.
Then $m \approx_v m'$. □

For defining (view-) serializability, it is not sufficient to require that a given multiversion schedule m is view equivalent to a *serial* multiversion schedule m'; as the following example demonstrates, the reason is that even a serial m' does not necessarily have a reads-from relation that is compatible with a serial monoversion schedule.

Example 5.10

Let $m = w_0(x_0)w_0(y_0)c_0r_1(x_0)r_1(y_0)w_1(x_1)w_1(y_1)c_1r_2(x_0)r_2(y_1)c_2$.
If each $r_i(x_j)$ occurring in m is replaced by a $r_i(x)$, and correspondingly each $w_i(x_i)$ by a $w_i(x)$, the following monoversion schedule results:
$s = w_0(x)w_0(y)c_0r_1(x)r_1(y)w_1(x)w_1(y)c_1r_2(x)r_2(y)c_2$.
Both schedules are serial; however, in s transaction t_2 reads data item x from t_1, while in m it reads x from t_0. □

To address this problem, we need to be careful when defining view serializability for multiversion schedules. We actually need to compare multiversion schedules to monoversion schedules in terms of their reads-from relations. In fact, this insight

should not be a big surprise given that versioning should be completely transparent to all application programs.

The schedule m from the previous example has a reads-from relation that differs from the reads-from relation of the corresponding monoversion schedule (as derived according to Definition 3.13 given in Chapter 3 for conventional schedules), since t_2 reads x_0, but not the version x_1 which was written last. Moreover, we can show that m's reads-from relation differs from the reads-from relation of every possible serial monoversion schedule for the same transactions. This means that m cannot be view equivalent to a serial execution without versions, and should thus not be accepted as a correct multiversion schedule.

This example-driven discussion directly leads to the following correctness criterion:

multiversion view serializability

Definition 5.11 (Multiversion view serializability)

Let m be a multiversion history. Then m is called *multiversion view serializable* if there exists a serial monoversion history m' for the same set of transactions s.t. $m \approx_v m'$.

Let MVSR denote the class of all multiversion view serializable histories.

Thus, for a history $m \in \text{MVSR}$ there is always a serial monoversion history s for the same set of transactions s.t. $m \approx_v s$, where "$\approx_v$" is now well-defined. The following example illustrates the relationships just described.

Example 5.12

let $m = w_0(x_0)w_0(y_0)c_0w_1(x_1)c_1r_2(x_1)r_3(x_0)w_3(x_3)c_3w_2(y_2)c_2$ and
$m' = w_0(x_0)w_0(y_0)c_0r_3(x_0)w_3(x_3)c_3w_1(x_1)c_1r_2(x_1)w_2(y_2)c_2$.
Then we have $m \approx_v m'$. Moreover, m' is view equivalent to history
$s = w_0(x)w_0(y)c_0r_3(x)w_3(x)c_3w_1(x)c_1r_2(x)w_2(y)c_2$. □

casting conventional schedules into multiversion schedules

The previous definition can be generalized to arbitrary *schedules* in a straigtforward way: A multiversion schedule m is *multiversion view serializable* if there exists a serial monoversion history m' s.t. $\text{CP}(m) \approx_v m'$. Furthermore, we can now define multiversion view serializability even for conventional (input) schedules: a schedule is multiversion view serializable if there is a version function such that casting the schedule into a multiversion schedule with this version function yields a multiversion schedule whose committed projection is in MVSR.

5.3.2 Testing Membership in MVSR

A natural question now is what the complexity of recognizing members of MVSR is. To this end, it may not be expected that this problem is simpler than the corresponding one for VSR, since we have:

Theorem 5.13

VSR $\subset$ MVSR

Inclusion of VSR in MVSR holds since every view serializable history can be perceived as a monoversion history and hence as a specific multiversion history (in MVSR). The fact that the inclusion is strict can be seen from the following example; consider

$$m = w_0(x_0)w_0(y_0)c_0r_1(x_0)w_2(x_2)w_2(y_2)c_2r_1(y_0)c_1.$$

Interpreted as a monoversion schedule by omitting the version subscripts, this schedule is the canonical example of an inconsistent read and clearly does not fall into class VSR. By having t_1 read an older version y_0 the multiversion schedule is, however, acceptable under the MVSR criterion, and is indeed view equivalent to the serial monoversion schedule with the order $t_0 < t_1 < t_2$. This once again demonstrates the significant performance improvements achievable by transparent versioning.

The previous theorem implies that the membership test for MVSR can intuitively not be easier than the one for VSR, since MVSR is a *relaxation* of conditions over VSR; in other words, with additional restrictions (in VSR) over MVSR, we obtain an NP-complete decision problem. Indeed we can show the following:

Theorem 5.14

The problem of deciding whether a given multiversion history is in MVSR is NP-complete.

Proof. In order to show that the problem is in NP, we just "guess" a serial monoversion history m' for a given multiversion history m and test $m \approx_v m'$; based on Theorem 5.13, this can be done in polynomial time simply by comparing the two sets of operations.

In order to show that the problem is NP-complete we reduce the corresponding decision problem for VSR to it: Let s be a monoversion history. Transform s into a multiversion history m by replacing each operation $w_i(x)$ by $w_i(x_i)$, and each operation $r_j(x)$ by $r_j(x_i)$, provided t_j reads item x in s from t_i. By the definition of view serializability, it follows that $m \in$ MVSR iff there exists a serial monoversion schedule s' s.t. $m \approx_v s'$. Now s' exists iff $s \in$ VSR. Thus we have $m \in$ MVSR iff $s \in$ VSR, and our claim follows. □

Different from the situation for monoversion histories, for multiversion histories there is a graph-theoretic characterization of membership in MVSR with a graph that has transactions as nodes (as opposed to the step-graph constructions in Chapter 3 where a graph has the steps of a schedule as its nodes). This will turn out relevant later in this chapter when we discuss concurrency-control protocols, since they will all guarantee this graph property; thus, the characterization developed next will be used as a *sufficient* condition for membership in MVSR in the verification of protocols. We first remind the reader of the definition of a conflict graph given in Chapter 3, which can be simplified for multiversion schedules due to the fact that only conflicts of the form $(w_i(x_i), r_j(x_i))$, $i \neq j$, can still occur. Thus, the conflict graph $G(m)$ of a multiversion schedule m has an edge between (committed) transactions t_i and t_j (of the form $t_i \to t_j$, $i \neq j$) simply if $r_j(x_i)$ is in m. The following is then obvious:

conflict graph

Theorem 5.15

For any two multiversion schedules m and m', $m \approx_v m'$ implies $G(m) = G(m')$.

The converse of this theorem is not true. Consider the multiversion schedules $m = w_1(x1_1)r_2(x_0)w_1(y_1)r_2(y_1)$ and $m' = w_1(x1_1)r_2(x_1)w_1(y_1)r_2(y_0)$ that have the same conflict graph but differ significantly in their version functions and their reads-from relations. So conflicts alone, in the limited sense that we consider version writes and reads on exactly these versions, are insufficient to reason about multiversion serializability.

We now extend the conflict graph of a multiversion schedule m as follows:

version order

Definition 5.16 (Version order)

If x is a data item, a *version order* for x is any nonreflexive and total ordering of all versions of x which are written by operations in m. A *version order* $<<$ for m is the union of all version orders of data items written by operations in m.

Note that the version order is independent of the execution order in which the versions of a data item were created. So it is possible that for versions x_i and x_j such that $w_i(x_i)$ preceded $w_j(x_j)$ in the schedule the version order is $x_j << x_i$. The version order rather serves to help determining the serialization order of transactions, that is the order in which versions appear in an equivalent serial monoversion schedule.

Example 5.17

Let m be the following schedule:
$m = w_0(x_0)w_0(y_0)w_0(z_0)c_0r_1(x_0)r_2(x_0)r_2(z_0)r_3(z_0)w_1(y_1)w_2(x_2)w_3(y_3)w_3(z_3)c_1c_2$
$c_3r_4(x_2)r_4(y_3)r_4(z_3)c_4$
A version order for m could be
$x_0 << x_2$, $y_0 << y_1 << y_3$, $z_0 << z_3$ □

multiversion serialization graph (MVSG)

Definition 5.18 (Multiversion serialization graph (MVSG))

For a given schedule m and a version order $<<$, the *multiversion serialization graph* $\mathrm{MVSG}(m, <<)$ of m then is the conflict graph $G(m) = (V, E)$ with the following edges added for each $r_k(x_j)$ and $w_i(x_i)$ in $\mathrm{CP}(m)$, where k, i, and j are pairwise distinct: If $x_i << x_j$, then $(t_i, t_j) \in E$, otherwise $(t_k, t_i) \in E$.

So for each triple of operations $w_j(x_j)$, $r_k(x_j)$, and $w_i(x_i)$ on the same data item x, the MVSG contains two out of the three potential edges:

- An edge from t_j to t_k is always present.
- In addition, there is either an edge from t_i to t_j, namely, if the version order is $x_i << x_j$, meaning that in an equivalent monoversion schedule $w_i(x_i)$ would precede $w_j(x_j)$, or
- an edge from t_k to t_i, namely, if the version order is $x_j << x_i$ so that $w_i(x_i)$ would follow $w_j(x_j)$ in an equivalent monoversion schedule and consequently has to follow also $r_k(x_j)$ as t_k needs to read the most recent version in a monoversion schedule.

Example 5.19

Let m and $<<$ be as in the previous example. The graph $\mathrm{MVSG}(m, <<)$ is shown in Figure 5.1. In this example the only edges which do not occur in $G(m)$ are the following:
(t_1, t_2): $r_1(x_0), w_2(x_2) \in \mathrm{op}(m)$, and $x_0 << x_2$

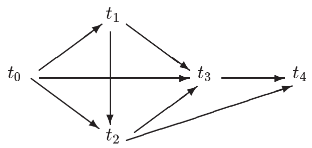


Figure 5.1: Multiversion serialization graph.

(t_1, t_3): $w_1(y_1), r_4(y_3) \in \mathrm{op}(m)$ and $y_1 << y_3$
(t_2, t_3): $r_2(z_0), w_3(z_3) \in \mathrm{op}(m)$, and $z_0 << z_3$ □

If a multiversion history happens to have an acyclic conflict graph, then we can apply topological sorting to obtain a total order or, equivalently, a serial multiversion history; however, this schedule obtained does not need to be view equivalent to a serial monoversion history, since reads-from relationships may change during the transition from a multiversion to a monoversion history. Version-order edges can be used to detect such situations: If $r_k(x_j)$ and $w_i(x_i)$ are in $\mathrm{CP}(m)$, a version order enforces that either $w_i(x_i)$ occurs before $w_j(x_j)$ or after $r_k(x_j)$; in both cases, the reads-from relation remains invariant from a transition to a monoversion history. A topological sorting of a multiversion serialization graph is possible only if the conflict graph is still acyclic after the addition of the version-order edges. With these considerations, the following can be verified:

Theorem 5.20

$m \in \mathrm{MVSR}$ iff there exists a version order $<<$ s.t. $\mathrm{MVSG}(m, <<)$ is acyclic.

Proof. (if) Since $\mathrm{MVSG}(m, <<)$ is acyclic, it can be sorted topologically into a serial history m' s.t. $\mathrm{RF}(m) = \mathrm{RF}(m')$, i.e., $m \approx_v m'$. It remains to be shown that m' can be perceived as a monoversion history. To this end, suppose that m' contains operations $r_k(x_j)$, $k \neq j$, as well as $w_i(x_i)$, $i \neq j$ and $i \neq k$. If $x_i << x_j$, the graph contains version-order edge $t_i \to t_j$, which enforces $t_i <_{m'} t_j$. Conversely, if $x_j << x_i$, then by definition the graph contains edge $t_k \to t_i$ enforcing $t_k <_{m'} t_i$ in m'. Thus, no transaction writes x in between t_j and t_k in m', so that version numbers can essentially be dropped from m' without changing the reads-from relation.

(only if) For a given history m and version order $<<$, let $\mathrm{MV}(m, <<)$ denote the graph that contains version-order edges *only*. Since version-order edges depend only on the operations in m and the version order, but not on the *sequencing* of the operations in m, we can conclude that if m and m' are multiversion histories s.t. trans(m) = trans(m'), then MV($m, <<$) = MV($m', <<$) for every version order $<<$.

Now let $m \in$ MVSR, and let m' be a serial monoversion history for the same transactions s.t. $m \approx_v m'$. By the remark just stated, MV($m, <<$) = MV($m', <<$) for every version order $<<$. By Theorem 5.15, $G(m) = G(m')$. Clearly, $G(m')$ is acyclic (since m' is serial), hence so is $G(m)$. Now define a specific version order

$$x_i <<_0 x_j \quad :\Longleftrightarrow \quad t_i <_{m'} t_j$$

(i.e., if $t_i \to t_j$ is in $G(m')$). Thus, if MV($m', <<_0$) is added to $G(m')$, the resulting graph MVSG($m', <<_0$) remains acyclic, and the same holds if MV($m', <<_0$) = MV($m, <<_0$) is added to $G(m)$ [$= G(m')$]. □

Notice that the decision problem for MVSR is not made easier by this result, since for a given schedule m it can not necessarily be tested in polynomial time whether a version order of the desired form exists. On the other hand, multiversion *schedulers* always produce such an order, so that their correctness can be verified on the basis of this theorem.

5.3.3 Multiversion Conflict Serializability

We next describe a subclass of MVSR that has a polynomial-time membership problem, which immediately results from a specific notion of "multiversion conflicts":

multiversion conflict

Definition 5.21 (Multiversion conflict)

A *multiversion conflict* in a multiversion schedule m is a pair of steps $r_i(x_j)$ and $w_k(x_k)$ such that $r_i(x_j) <_m w_k(x_k)$.

We emphasize that this notion of *multiversion* conflicts is fundamentally different from our earlier definition of conflicts where we had simply carried over the conventional notion of conflicts to a versioned setting in a syntactic manner so that only wr operations pairs on the same version counted as conflicts. In contrast, the above definition essentially says that in a multiversion schedule (including the special case of monoversion histories) the only relevant kind of conflicts are rw operation pairs on the same data item, not necessarily on the same version. It is easy to see that ww pairs on the same data item do no longer count as conflicts, as they create different versions and it is up to the read steps to choose the proper version. To realize that wr pairs are not really conflicts is a bit more involved. The essence of a conflict pair pq is that we are not allowed to commute p with q. For wr pairs,

however, commuting the pair so that the read would precede the write is admissible because it restricts the version selection choices for the read that can still render the schedule correct (i.e., MVSR). So if the resulting schedule, with the read before the write, is MVSR, the original schedule with the wr pair would definitely be MVSR as well. This is the rationale for considering wr pairs (on the same data item) as conflict-free. The third remaining type of operation pairs, rw pairs, has the opposite effect: by commuting an rw pair so that the read would follow the write, we end up with a schedule that has one more choice in selecting an appropriate version for the read step. So, if we succeed in verifying that the resulting schedule is MVSR, this would still not say anything about the correctness of the original schedule. Thus, rw pairs on the same data item constitute conflicts in a multiversion setting.

Because of the above asymmetry in the treatment of wr and rw pairs, the notion of a multiversion conflict cannot be easily used for defining an equivalence relation on multiversion schedules. Rather it leads to an asymmetric notion of transforming one schedule into another by means of repeatedly commuting conflict-free operations. This obviously resembles the notion of commutativity-based reducibility introduced in Chapter 3, with the decisive difference, however, that we do not have a symmetric commutativity relation in our current setting. If we can eventually produce a serial monoversion history by commuting operations such that we always (i.e., after each transformation step) respect the ordering of rw conflict pairs, then we know that a multiversion schedule is guaranteed to be MVSR. From this argument we can already suspect that this is only a sufficient but not necessary condition for membership in MVSR; this conjecture will soon be confirmed.

The above consideration leads to the following correctness criteria (which, for simplicity, is stated only for totally ordered multiversion histories, and would have to be extended, in a straightforward manner, by an ordering rule in the sense of Chapter 3, Section 3.8.3, to cover also partial orders):

multiversion reducibility

Definition 5.22 (Multiversion reducibility)

A (totally ordered) multiversion history m is *multiversion reducible* if it can be transformed into a serial monoversion history by a finite sequence of transformation steps each of which exchanges the order of two adjacent steps (i.e., steps p, q with $p < q$ such that $o < p$ or $q < o$ for all other steps o) but without reversing the ordering of a multiversion conflict (i.e., rw) pair.

multiversion conflict serializability (MCSR)

Definition 5.23 (Multiversion conflict serializability)

A multiversion history m is *multiversion conflict serializable* if there is a serial monoversion history for the same set of transactions in which all pairs of operations in multiversion conflict occur in the same order as in m. Let MCSR denote the class of all multiversion conflict serializable histories.

From what we have discussed above, the following theorems can be derived.

Theorem 5.24

A multiversion history is multiversion reducible iff it is multiversion conflict serializable.

Theorem 5.25

MCSR $\subset$ MVSR.

In order to see that this inclusion is strict, consider the following multiversion history which is effectively a monoversion history:

$$m = \quad w_0(x_0)w_0(y_0)w_0(z_0)c_0r_2(y_0)r_3(z_0)w_3(x_3)c_3r_1(x_3)w_1(y_1)c_1w_2(x_2)c_2 \\ r_\infty(x_2)r_\infty(y_1)r_\infty(z_0)c_\infty.$$

The pairs $r_1(x_3)$ and $w_2(x_2)$ as well as $r_2(y_0)$ and $w_1(y_1)$ of operations can apparently not be exchanged without m losing its monoversion property; hence $m \notin$ MCSR. On the other hand, $m \approx_v t_0t_3t_2t_1t_\infty$, which implies $m \in$ MVSR. □

It can be shown that membership of a history in class MCSR can be characterized in graph-theoretic terms using the following notion of a multiversion conflict graph.

multiversion conflict graph

Definition 5.26 (Multiversion conflict graph)

Let m be a multiversion schedule. The multiversion conflict graph of m is a graph that has the transactions of m as its nodes and an edge from t_i to t_k if there are steps $r_i(x_j)$ and $w_k(x_k)$ for the same data item x in m such that $r_i(x_j) <_m w_k(x_k)$.

Not surprisingly, we obtain:

Theorem 5.27

A multiversion history is MCSR iff its multiversion conflict graph is acyclic.

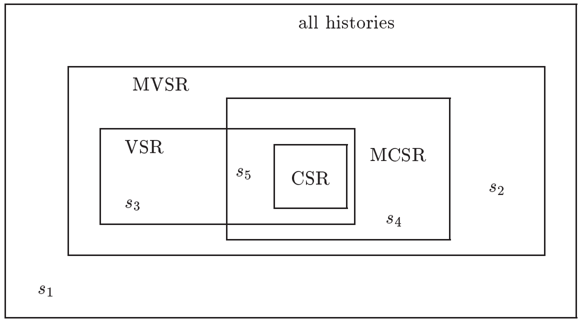
So MCSR has a polynomial membership test, and all practical multiversion concurrency control protocols fall into this class. Note, however, that the equivalent serial monoversion history for an acyclic multiversion conflict graph cannot simply be derived by topologically sorting the graph. Rather the above theorem only states that acyclicity of the graph is a sufficient and necessary condition for the existence of a sequence of transformation steps (in the sense of the definition of multiversion reducibility) that yields a serial monoversion history. For proving the correctness of protocols, the graph characterization of the larger class MVSR along with an appropriately chosen version order turns out to be an elegant and more practical tool, as we will see in the next section.

Figure 5.2 summarizes the relationships between the various classes of histories studied in this chapter so far. In the histories of this figure, transaction t_0 is not explicitly shown, and neither is t_∞. Moreover, all histories are shown as monoversion histories, but it should by now be clear that and how versioning comes into the picture. For example, s_2 is a representative for history

$$w_1(x_1)c_1r_2(x_1)r_3(y_0)w_3(x_3)w_2(y_2)c_2c_3.$$

5.4 Limiting the Number of Versions

In our discussion so far we have always assumed that enough storage space is available for holding *any* number versions. Thus, we have never put a limit on the number of versions of a particular data item, or on the number of versions of all data items taken together. While this is reasonable from a theoretical point of view, in that it has allowed us to develop a serializability theory for multiversion histories that indeed generalizes the monoversion case, it is rarely practical, since the number of versions will never be allowed to grow indefinitely. Even though disk space is relatively inexpensive, we cannot assume that a data server will be given unlimited space to manage its versions, nor can we assume that managing extremely large numbers of versions can be done efficiently. Therefore it appears reasonable to impose an upper limit on the number of versions that the system can distinguish, or keep around simultaneously. The important point we want to make in this section is that limiting the number of versions can make a difference, in the sense that schedules which are multiversion serializable if no upper bound on the number of versions exists may lose this property in the presence of a limit on the total number of versions.



$$
\begin{aligned}
s_1 &= r_1(x)r_2(x)w_1(x)w_2(x)c_1c_2 \\
s_2 &= w_1(x)c_1r_2(x)r_3(y)w_3(x)w_2(y)c_2c_3 \\
s_3 &= w_1(x)c_1r_2(x)r_3(y)w_3(x)w_2(y)c_2c_3w_4(x)c_4 \\
s_4 &= r_1(x)w_1(x)r_2(x)r_2(y)w_2(y)r_1(y)w_1(y)c_1c_2 \\
s_5 &= r_1(x)w_1(x)r_2(x)w_2(y)c_2w_1(y)w_3(y)c_1c_3
\end{aligned}
$$

Figure 5.2: Classes of multiversion serializable histories.

As an example, we consider the following monoversion history:

$$m = w_0(x_0)c_0r_1(x_0)w_3(x_3)c_3w_1(x_1)c_1r_2(x_1)w_2(x_2)c_2.$$

This history is obviously in MVSR, with the version order $x_0 << x_1 << x_2 << x_3$. However, it is not equivalent to any serial schedule that maintains at most two versions of x simultaneously. As is easily verified, there are six different serial monoversion histories for the three transactions t_1, t_2, and t_3 in m; they are as follows:

$$
\begin{aligned}
m_1 &= w_0(x_0)c_0r_1(x_0)w_1(x_1)c_1r_2(x_1)w_2(x_2)c_2w_3(x_3)c_3 \\
m_2 &= w_0(x_0)c_0r_1(x_0)w_1(x_1)c_1w_3(x_3)c_3r_2(x_3)w_2(x_2)c_2 \\
m_3 &= w_0(x_0)c_0r_2(x_0)w_2(x_2)c_2r_1(x_2)w_1(x_1)c_1w_3(x_3)c_3 \\
m_4 &= w_0(x_0)c_0r_2(x_0)w_2(x_2)c_2w_3(x_3)c_3r_1(x_3)w_1(x_1)c_1 \\
m_5 &= w_0(x_0)c_0w_3(x_3)c_3r_1(x_3)w_1(x_1)c_1r_2(x_1)w_2(x_2)c_2 \\
m_6 &= w_0(x_0)c_0w_3(x_3)c_3r_2(x_3)w_2(x_2)c_2r_1(x_2)w_1(x_1)c_1
\end{aligned}
$$

Since $m \approx_v m_1$ we first observe that $m \in$ MVSR. Now let us assume that the total number of versions that can exist at the same time is, for whatever reasons, limited to $k = 2$; since m and the serial histories for m's transactions only write x, this means only the last two versions of x that are written in a schedule will be stored.

For m_1 this means the following: First t_0 and t_1 produce x-versions x_0 and x_1, resp. Next t_2 will overwrite version x_0, which is older, and t_3 will overwrite x_1. So when m_1 terminates, the database will comprise versions x_2 and x_3. Notice that history m will produce versions x_1 and x_2 as those that are written last; thus m can no longer be considered equivalent to m_1.

It can be verified by similar arguments that m is also not equivalent to *any* of the other serial monoversion schedules shown above, since the following holds:

1. m_2 (like m_1) produces versions x_2 and x_3,
2. m_3 and m_4 both produce versions x_1 and x_3,
3. m_5 and m_6 produce x_1 and x_2 (as does m), but m_5 comprises operation $r_1(x_3)$, while m_6 comprises $r_2(x_3)$, both of which do not occur in m.

Since m is not equivalent to any of the monoversion histories possible in this example, we may conclude that, different from the general case, history m is *not* a member of class MVSR if only two versions are allowed.

The above observations can be formalized. In the presence of an upper bound for the total number of data-item versions, view equivalence of histories is again defined as in Chapter 3, namely by asking for identical reads-from relations *and* equal final write operations. To be even more precise, we can furthermore incorporate a given bound k for the number of versions into the definition and hence speak of *k-version view serializability*; the corresponding classes of histories will be denoted by kVSR, $k > 0$.

For the sample schedule m discussed above, it then follows immediately that

$$m \in \text{MVSR–2VSR}.$$

In addition, it is easily verified that

$$\text{VSR} = \text{1VSR}$$

and that

$$\text{MVSR} = \bigcup_{k>0} k\text{VSR}.$$

Since we have stated above that MVSR has an NP-complete decision problem, it moreover follows that the problem of deciding whether a given multiversion schedule is in class kVSR is NP-complete for every $k > 0$. Finally, it can be shown that the various classes of k-version view serializable histories form a strict hierarchy, i.e.,

a hierarchy of classes

$$\text{VSR} = \text{1VSR} \subset \text{2VSR} \subset \text{3VSR} \subset \ldots \subset \text{MVSR}$$

5.5 Multiversion Concurrency Control Protocols

We now develop concurrency-control protocols for multiversion databases, or enhancements of known protocols for a multiversion environment. For all of them, the important point is that they produce total version orders, so that their correctness follows from the characterization of membership in class MVSR stated in Theorem 5.20. In the discussion that follows we will assume (unless stated otherwise) that all versions written can be stored (so that limiting the number of available versions is not an issue). We will discuss enhancements of the TO, 2PL, and SGT protocols (in this order).

5.5.1 The MVTO Protocol

A *multiversion timestamp ordering* (MVTO) scheduler essentially processes operations in first-in-first-out (FIFO) fashion. In particular, it transforms data operations into operations on versions of data items, and processes them in such a way that the result appears as if it had been produced by a serial monoversion schedule with transactions in the order of timestamps that are assigned at the beginning of a transaction. Each version carries the timestamp $ts(t_i)$ of the transaction t_i by which it has been created. In detail, an MVTO scheduler acts as follows:

MVTO scheduler

1. A step $r_i(x)$ is transformed into a step $r_i(x_k)$, where x_k is the version of x that carries the largest timestamp $\leq ts(t_i)$ and was written by t_k, $k \neq i$;
2. a step $w_i(x)$ is processed as follows:
   - If a step of the form $r_j(x_k)$ s.t. $ts(t_k) < ts(t_i) < ts(t_j)$ has already been scheduled, then $w_i(x)$ is rejected and t_i is aborted;
   - otherwise, $w_i(x)$ is transformed into $w_i(x_i)$ and executed.
3. A commit c_i is delayed until the commit c_j of all transactions t_j that have written new versions of data items read by t_i have been processed. (This part of the protocol is optional and included in order to ensure correct transaction recovery, e.g., avoiding anomalies such as dirty reads. This will be considered in depth in Chapter 11).

Note that we do not consider the case where a transaction can read an item after having written it; if this was allowed, a transaction would have to be able to see a version it has itself produced (cf. Step 1).

The important observation for proving the MVTO protocol correct, i.e., for showing that

$$Gen(\text{MVTO}) \subseteq \text{MVSR}$$

correctness of MVTO

holds, is that a version order can be defined as follows:

$$x_i << x_j \iff ts(t_i) < ts(t_j)$$

Figure 5.3 shows a sample execution under the MVTO protocol. The interleaved execution of transactions t_1 and t_2 once more illustrates the great performance enhancement of multiversion concurrency control: as a monoversion schedule, the execution would have been a classical case of inconsistent reading; with versioning, however, the situation is perfectly acceptable as t_1 simply reads the old version y_0 thus ensuring consistency with its previous reading of x_0. With MVTO in particular, this is the result of t_1's timestamp being smaller than that of the concurrent transaction t_2. Transaction t_3 needs to wait (as indicated by the dashed line in the figure) right before its commit, because it has read the uncommitted version x_2 (and indeed had to read this version because its timestamp is larger than that of t_2); once t_2 is committed, t_3 can commit as well. Transaction t_4 is an example for a “late” writer: it creates the new version y_4; however, transaction t_5 has already read the version y_2. As the timestamp ordering prescribes the serialization order $t_2 < t_4 < t_5$, t_5 should actually have read y_4 (which, of course, was impossible as that version did not yet exist at the time of t_5's read). Thus, t_4's write occurs “too late”, and t_4 must be aborted according to the MVTO rules.

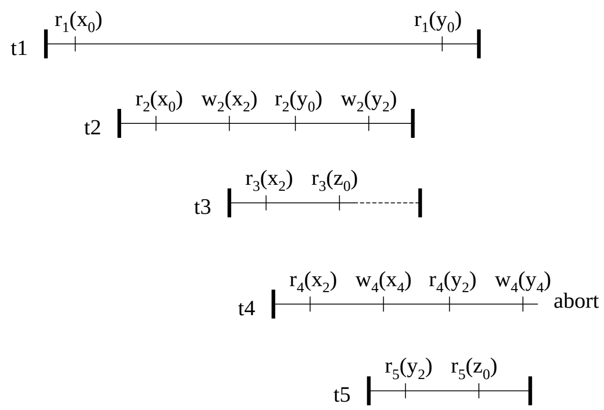


Figure 5.3: Sample execution under the MVTO protocol.

5.5.2 The MV2PL Protocol

We next describe a scheduler that uses locks and a strong two-phase discipline (i.e., the SS2PL rules as far as unlocking is concerned). We first assume as a simplification that *all* versions of data items ever written are kept around; this will be relaxed later by considering a special case in which at most two versions of a data item exist simultaneously. We generally distinguish

- *committed versions*, which have been written by transactions that are already committed;
- the *current version* of a data item, which is the committed version of that data item written by the transaction that was committed last;
- *uncommitted versions*, which are all remaining versions (created by transactions that are still active).

MV2PL scheduler

The scheduler makes sure that at each point in time there is *at most one* uncommitted version of any data item. Depending on whether read steps are allowed to read only the current version or also uncommitted versions, several variants of this protocol can be distinguished. Moreover, the scheduler treats the final step of a transaction different from the other steps, where "final" refers to the last data operation before the transaction's commit or to the commit itself (both interpretations are feasible under this protocol). An individual step is handled as follows:

1. If the step is not final within a transaction:
   (a) an $r(x)$ is executed right away, by assigning to it the current version of the requested data item, i.e., the most recently committed version (but not any other, previously committed one), or by assigning to it an uncommitted version of x,
   (b) a $w(x)$ is executed only when the transaction that has written x last is finished, so that there are no other uncommitted versions of x.
2. If the step is final within transaction t_i, it is delayed until the following types of transactions are committed:
   (a) all those transactions t_j which have read the current version of a data item written by t_i;
   (b) all those t_j from which t_i has read some version.

The following example indicates how such a scheduler works.

Example 5.28

Assume the MV2PL scheduler is supplied with the following sequence of steps:

$$s = r_1(x)w_1(x)r_2(x)w_2(y)r_1(y)w_2(x)c_2w_1(y)c_1$$

Then it proceeds as follows:

(1) $r_1(x)$ is assigned to x_0 and is executed: $r_1(x_0)$

(2) $w_1(x)$ is executed since no other transaction is still active: $w_1(x_1)$

(3) let $r_2(x)$ be assigned to x_1 and executed: $r_2(x_1)$

(4) $w_2(y)$ is executed: $w_2(y_2)$

(5) let $r_1(y)$ be assigned to y_0 and executed: $r_1(y_0)$

(6) if $w_2(x)$ were *not* the final step of t_2, it would be delayed since t_1 is still active and has written x_1. However, as it *is* the final step of t_2, the final-step rules need to be applied. It turns out that t_2 nonetheless has to wait for the following reason:

   (a) t_1 has read the current version of data item y (y_0), and t_2 overwrites this version;

   (b) t_2 has read x_1 from t_1.

(7) $w_1(y)$, the final step of t_1, is executed since

   (a) t_2 has *not* read a current version of a data item written by t_1 (current versions are x_0, y_0),

   (b) t_1 has not read a version written by t_2. $w_1(y_1)$

(8) finally, $w_2(x)$ can be executed: $w_2(x_2)$

□

In the following further elaboration of the protocol, we restrict ourselves to a special case of MV2PL that is particularly relevant in applications, namely, the *2V2PL protocol* (two-version 2PL), which keeps at most two versions of any data item at each point in time. These two versions are those which a data server may need to keep around (in some form) for recovery purposes (see Part III). Suppose that t_i writes data item x, but is not yet committed, the two versions of x are its *before image* as well as its *after image*. As soon as t_i commits, the before image can be dropped since the new version of x is now stable, and old versions are no longer needed nor maintained. In other words, a 2V2PL scheduler always maintains a current version for a data item as well as, in the presence of transactions which write, candidates for sucessor versions. It is a characteristic property of the 2V2PL variant that at most one such uncommitted candidate is allowed at every point of time. So it is mostly read operations that benefit from versioning, and in 2V2PL read operations are restricted to read current versions only (i.e., the last committed version).

2V2PL protocol

2V2PL uses three kinds of locks on data items (i.e., not on individual versions), all of which are kept until the termination of a transaction:

1. *rl* (read lock)

   A $rl(x)$ is set immediately prior to an operation $r(x)$ w.r.t. the current version of x.

2. *wl* (write lock)

   A $wl(x)$ is set (immediately) prior to an operation $w(x)$ for writing a new uncommitted version of x.

3. *cl* (certify (or commit) lock)

   A $cl(x)$ is set prior to the execution of the final step of a transaction (i.e., usually upon the transaction's commit request) on every data item x which this transaction has written.

The lock compatibilities for these three types of locks are as follows (as usual, + denotes compatibility and − incompatibility):

	$rl(x)$	$wl(x)$	$cl(x)$
$rl(x)$	+	+	−
$wl(x)$	+	−	−
$cl(x)$	−	−	−

Notice that unlock operations need to obey the 2PL rule.

The role of the write locks merely is to ensure that at most one uncommitted version can exist for each data item at each point in time. The key point to ensure that the resulting output schedules are multiversion serializable is the acquisition of the certify locks. The tests regarding the admissible orderings of reading current versions and creating new ones is encoded in the compatibility checking of read locks and certify locks. In this sense, certify locks play the role that write locks have in conventional, non-versioned locking. However, the fact that certify locks are acquired only at the end of the transaction and are thus usually held for a much shorter time is a great performance advantage over conventional, monoversion 2PL.

Example 5.29

Consider the input schedule

$$r_1(x)w_2(y)r_1(y)w_1(x)c_1r_3(y)r_3(z)w_3(z)w_2(x)c_2w_4(z)c_4c_3$$

and let x_0, y_0, z_0 denote the current versions that exist before this input schedule starts. The 2V2PL protocol produces the following output schedule, where ul denotes unlock operations that remove all locks of a transaction:

$$rl_1(x)r_1(x_0)wl_2(y)w_2(y_2)rl_1(y)r_1(y_0)wl_1(x)w_1(x_1)cl_1(x)ul_1c_1$$
$$rl_3(y)r_3(y_0)rl_3(z)r_3(z_0)wl_2(x)cl_2(x)wl_3(y)w_3(z_3)cl_3(z)ul_3c_3$$
$$cl_2(y)ul_2c_2wl_4(z)w_4(z_4)cl_4(z)ul_4c_4$$

The output schedule is also illustrated in Figure 5.4. Note that $cl_2(y)$ would be requested right before the begin of t_4, but cannot be granted until the commit of t_3 because of the conflicting read lock held by t_3. Further note that $wl_4(z)$ causes a lock wait, too, because of its incompatibility with the still held $wl_3(z)$; so t_4 is blocked until after the commit of t_3. □

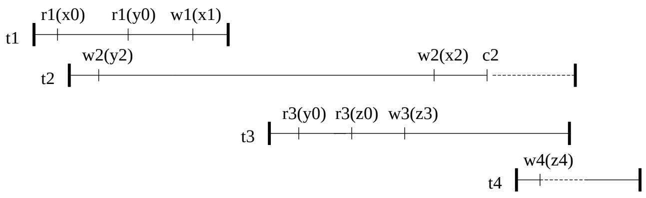


Figure 5.4: Sample execution under the 2V2PL protocol.

The correctness of the 2V2PL protocol follows from three facts. First, the order of the transactions' final steps (usually their commit requests) provide us with a total ordering from which we can derive a suitable version order. Second, because write locks on the same data item are incompatible at most one uncommitted version can exist at a given time, so that the following choice of the version order is unambiguous: $x_i << x_j \quad \Longleftrightarrow \quad f_i < f_j$ with f_i, f_j denoting the final steps of two transactions t_i and t_j that have both written (new versions of) x. Third, read operations are always directed to the current version, and certify locks of concurrent writers, which are incompatible with read locks, serve to determine situations when a new committed version is produced while a reader is still on progress. In the latter case the commit of the writer is blocked until the reader is terminated. Putting these three observations together allows us to construct the multiversion serialization graph for the specific version order, and to show that this graph is guaranteed to be acyclic.

correctness of 2V2PL

The following example shows that, just like 2PL, 2V2PL (and hence MV2PL) is not deadlock-free. Consider the following scheduler input:

$$r_1(x) r_2(y) w_1(y) w_2(x) c_1 c_2$$

From this input, the scheduler produces

$$rl_1(x) r_1(x_0) rl_2(y) r_2(y_0) wl_1(y) w_1(y_1) wl_2(x) w_2(x_2)$$

where x_0 and y_0 denote the current versions as of the begin of the example schedule. Next t_1 needs a certify lock $cl_1(y)$, which is incompatible with $rl_2(y)$, and t_2 needs another certify lock $cl_2(x)$, which is incompatible with $rl_1(x)$. Thus the scheduler is stuck in a deadlock. There are various techniques to cope with such a situation, as we already discussed in Chapter 4 for the ordinary 2PL protocol.

5.5.3 The MVSGT Protocol

A final multiversion scheduler we look at is another SGT protocol, now enhanced for managing multiple versions. We note that it is no longer possible, like in the monoversion case, to provide a protocol capable of generating *exactly* MCSR. Instead there now exists an unlimited number of schedulers, each of which generates a distinct subset of MCSR s.t. the union of all of them equals MCSR. We sketch a general framework for such schedulers here, which comprises certain nondeterminstic steps; if these steps are made more precise, the various schedulers result.

MVSGT scheduler

The general scheduler maintains a multiversion conflict graph G whose nodes are the various transactions including t_0. In the beginning, G has edges from t_0 to all other nodes; in the end, when the input has been processed completely, G will contain all edges from the conflict graph plus a few addtional ones, depending on the result of the nondeterministic steps.

Let us first consider the assignment of a new version to some read step: Assume that $r_i(x)$ arrives at the scheduler; *candidates* from which t_i can read are t_0 as well as all $w_j(x)$ which have already been scheduled. However, the following have to be excluded:

(i) those t_j on a path that originates from t_i, since these are supposed to follow t_i in an equivalent serial schedule; let us call them *late*;

(ii) those t_j for which a path exists from t_j to another candidate t_k that writes x *and* from t_k to t_i, since in an equivalent serial schedule t_k writes x after t_j; let us call them *early*.

Now a first nondeterministic step consists of choosing a version for $r_i(x)$ among those written by transactions that are neither late nor early.

Example 5.30

Let us consider the situation shown in Figure 5.5, where t_0 as well as the edges from t_0 to any other transaction $t_i, i \in \{1, \ldots, 7\}$, are not shown.

Suppose that $r_1(x)$ arrives at the scheduler, and that t_2 through t_7 have already written x; hence they are "candidates" in the sense just discussed. Late transactions are t_2

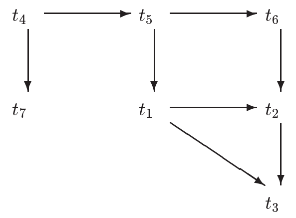


Figure 5.5: A conflict graph.

and t_3, while early ones are t_0 and t_4. Thus there is a choice between candidates t_5, t_6, and t_7. □

Now if the conflict graph G is acyclic, then for each read step there is at least one candidate which is neither late nor early. The simple reason is that not all transactions in a schedule are late; among those which are not late choose one without path-wise connection to other non-late candidates. The latter one is not early. The multiversion conflict graph G is now maintained as follows:

1. If $w_i(x)$ is executed, an edge of the form (t_j, t_i) is added to the graph for each t_j for which an $r_j(x)$ has already been scheduled; in this way, G remains a super-graph of the MVSG. If G becomes cyclic, then $w_i(x)$ is *not* executed; otherwise the version newly written is kept.

2. If $r_i(x)$ is scheduled and receives the version written by $w_j(x)$, an edge of the form (t_j, t_i) is added to G; since t_j is not late, this does not close a cycle. In addition, an edge of the form either (t_k, t_j) or (t_i, t_k) is added to G for each step $w_k(x)$ already scheduled, where t_k is neither late nor early.

Consider the previous example again: If t_7 is chosen from the set of candidates, the new edges (t_7, t_1), (t_5, t_7) (edge (t_1, t_5) would close a cycle), and (t_1, t_6) (alternatively (t_6, t_7)) result. More precisely, a choice is made s.t. (t_k, t_i) if there exists a path from t_k to t_i in G, and (t_i, t_k) otherwise. It can be shown that in this way G remains acyclic; in addition, each schedule produced is in MCSR.

5.5.4 A Multiversion Protocol for Read-Only Transactions

Many modern data-intensive applications exhibit workloads with a dominant fraction of read-only transactions that read and analyze large amounts of data. In

the presence of (at least a nontrivial fraction of) concurrent update transactions, concurrency control is crucial to ensure that the read-only transactions "see" consistent data. In fact, such situations pose a stress test for the concurrency control component, because of the long duration of the read-only transactions. Under a conventional monoversion scheduler, say a 2PL protocol, such long transactions would often lead to a high probability of blocking the update transactions and eventually to disastrous performance (e.g., unacceptably high response times).

Multiversion concurrency control protocols, on the other hand, can avoid many of these blocking situations by assigning "old" versions to readers. However, as we have seen in the previous subsections, such protocols can become fairly complex, are all but easy to implement, and are likely to incur noticeable run-time overhead. In this subsection we consider a simple, hybrid protocol that aims to reconcile the simplicity of conventional (S)2PL or simple timestamping with the performance benefits of versioning. It does so by exploiting versioning only for read-only transactions, thus "picking the low-hanging fruit" in the sense that say 10 percent of the complexity of multiversion protocols is sufficient to achieve say 90 percent of the benefit. To this end, read-only transactions must be explicitly marked as such upon their beginning. In practice, this is often feasible; if in doubt, a transaction would simply have to be marked as a (potential) updater. This dichotomy of transactions is the basis for the following combined protocol:

1. *Update transactions* are subject to the conventional S2PL protocol. They acquire conventional locks for both read and write steps, which are released according to the two-phase rule with write locks held until commit (and possibly even read locks held until commit if the strong variant SS2PL is used). In contrast to the conventional monoversion setting, however, each write step creates a new version rather than overwriting the data item; each version is timestamped with the timestamp of its transaction which corresponds to the commit time of the transaction. So update transactions do not benefit from versioning at all, but provide the potential to benefit the read-only transactions while at the same time keeping the overhead of the protocol as low as possible.

read-only multiversion protocol (ROMV)

2. *Read-only transactions* are handled by a multiversion timestamp protocol, similar to the MVTO rules but with a twist on committed versus uncommitted versions. Each such transaction is assigned a timestamp which, unlike the update transactions, corresponds to the begin of the transaction (rather than its commit). Now the key point of the protocol is that a read operation by a read-only transaction is assigned to the most recent version that has been committed at the time of the reader's begin; in other words: the read-only

transactions always accesses the version with the largest timestamp that is smaller than the transaction's timestamp (assuming that timestamps of different transactions are never equal).

Figure 5.6 shows a self-explanatory scenario of how the protocol works, with update transactions $t2$, $t3$, and read-only transactions $t1$, $t4$, and $t5$. The lock wait of $t3$ because of the (conventional) wr lock conflict on x is indicated by a dashed line.

A correctness proof for this protocol, let us call it *read-only multiversion protocol (ROMV)*, is relatively straightforward: the version order for a data item is given by the commit order (and thus timestamp order) of the update transactions, and read-only transactions are placed in the serialization order according to their timestamps (i.e., in between the update transactions). This argument could be further formalized by explicitly showing that the multiversion serialization graph is acyclic for the given version order. In fact, the ROMV protocol generates only schedules in the MCSR subclass, and another simple proof of its correctness would thus amount to showing the following: a serial monoversion history can be derived from a given ROMV-generated history by commuting conflict-free operations of update transactions and commuting the steps of read-only transactions with preceding writes such that each read-only transaction eventually follows right after the transaction with the most recent commit as of the transaction's begin.

correctness of ROMV

Obviously, although the ROMV protocol is very simple, it does raise a number of nontrivial implementation issues. In particular, as versions are transient and merely serve concurrency-control purposes, it is desirable to have some form of garbage collection for old versions that are definitely no longer beneficial for any read-only transaction. A simple sufficient condition for a version being eligible to garbage collection would be that it is (i) not the most recent committed version (which, of course, must not be garbage-collected) and (ii) its timestamp is older than the timestamp of the oldest active read-only transaction. This approach assumes that there is no explicit limit for the number of simultaneously kept versions of a data item. If there is such a limit, an additional complication is that some read-only transactions may have to be aborted because the version that they would have to read according to the timestamping rule is no longer available. Another nontrivial implementation issue is how to manage the timestamps for the versions themselves; this may be somewhat tricky because timestamps are assigned only upon the commit of a transaction. A straightforward solution could be to place timestamps in the versions themselves (e.g., in page headers, assuming that data items are pages), but this would require re-visiting all accessed pages at commit time; alternative approaches include additional data structures that temporarily maintain write sets and timestamps of committed update transactions.

garbage collection of old versions

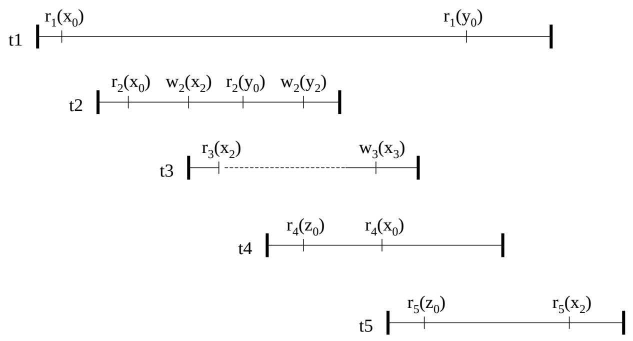


Figure 5.6: Sample execution under the ROMV protocol.

5.6 Lessons Learned

Keeping multiple versions of data items around is an attractive idea, and at the same time it is a practical one, as we will see when we discuss recovery. From a theoretical point of view, an adaptation of serializability theory to a multiversion setting is straightforward, as soon as the appropriate versioning restrictions have been identified. A tricky situation, considerably different from the general case, arises for a limited number of versions, as an entire hierarchy of classes of serializable histories enters the picture.

From a more practical point of view, adapting the various scheduling protocols to a multiversion setting is not too difficult, once the appropriate locks (or alternative synchronization mechanisms) have been found. Intuitively, multiversion protocols allow various degrees of freedom for assigning, or choosing, versions to process within a schedule. The special case of providing consistent views of the data to read-only transactions, discussed in Subsection 5.5.4, is of very high practical relevance and greatly benefits from versioning; protocols like the one described there are used in several commercial database systems.

Exercises

1. For the following three histories, test whether they are monoversion histories or members of MVSR or MCSR; in case a schedule is a member of MVSR, additionally find out for which values of $k > 0$ the history is in class kVSR:
   $m_1 = w_0(x_0)w_0(y_0)w_0(z_0)c_0r_3(x_0)w_3(x_3)c_3w_1(x_1)c_1r_2(x_1)w_2(y_2)w_2(z_2)c_2$
   $m_2 = w_0(x_0)w_0(y_0)c_0w_1(x_1)c_1r_3(x_1)w_3(x_3)r_2(x_1)c_3w_2(y_2)c_2$
   $m_3 = w_0(x_0)w_0(y_0)c_0w_1(x_1)c_1r_2(x_1)w_2(y_2)c_2r_3(y_0)w_3(x_3)c_3$
   For schedules in MVSR, give also an appropriate version function for a final transaction t_∞.

2. For the multiversion schedule
   $m = w_0(x_0)w_0(y_0)c_0r_1(x_0)w_1(x_1)r_2(x_1)w_2(y_2)w_1(y_1)w_3(x_3)$
   test whether there exists a version order $<<$ s.t. MVSG$(m, <<)$ is acyclic. If there is an acyclic graph, find an appropriate version function for a final transaction t_∞ such that the graph remains acyclic.

3. Prove: In the "no-blind-writes" model where each data item written by a transaction must have been read before in the same transaction, MCSR = MVSR.

4. Prove: In the "action" model where each step is a combination of a read operation immediately followed by a write operation on the same data item, MVSR = VSR.

5. Consider the following schedule, given in "conventional" form without a specific version function:
   $$r_1(x)r_2(x)r_3(y)w_2(x)w_1(y)c_1w_2(z)w_3(z)r_3(x)c_3r_2(y)c_2$$
   Show that this schedule is multiversion serializable, i.e., could be allowed by a multiversion concurrency control. Give a feasible version function, and also a feasible version order. How do the resulting executions (i.e., output schedules) under the MVTO and the 2V2PL protocols look like?

6. Consider the input schedule:
   $$w_1(x)c_1r_2(x)r_3(x)c_2r_4(x)w_3(x)c_4c_3$$
   Give the resulting output schedule under the MVTO protocol.

7. Consider the input schedule of the MV2PL example 5.28:
   $$s = r_1(x)w_1(x)r_2(x)w_2(y)r_1(y)w_2(x)c_2w_1(y)c_1.$$
   Apply the specialized 2V2PL protocol to this input and give the resulting output.

8. Complete the proof sketches for the correctness of the MVTO, 2V2PL, and ROMV protocols.

9. Reconsider the ROMV protocol that has been specifically geared for read-only transactions. What happens if the protocol is relaxed in such a way that update transactions use the timestamp-based version selection for their read steps? That is, update transactions would still use conventional exclusive locks for writes, but would exploit versioning for reads by selecting the most recent version that was committed at the time of the update transaction's begin. Is this protocol still correct in that it guarantees MVSR schedules?

Bibliographical and Other Remarks

The idea of using several versions of data items in concurrency control goes back to Reed (1978, 1983) and Bayer et al. (1980), which are also the origins of the MVTO and 2V2PL protocols, respectively. Further seminal papers on this topic have been written by Stearns and Rosenkrantz (1981) as well as Bernstein and Goodman (1983).

Our discussion in this chapter is along the lines of Bernstein and Goodman (1983) as well as Bernstein et al. (1987), and Claybrook (1992). The notion of a version function as used in the beginning of the chapter (cf. Definition 5.2) is essentially from Papadimitriou (1986); serial monoversion schedules are termed *1-serial* multiversion schedules in Bernstein et al. (1987), and members of MVSR are called *one-copy serializable*. The NP-completeness results mentioned are due to Bernstein and Goodman (1983), Lausen (1983), and Papadimitriou and Kanellakis (1984). The latter source also proves the hierarchy result mentioned in Section 5.2.4. A systematic investigation of multiversion serializability in the presence of a limited number of versions was launched by Papadimitriou and Kanellakis (1984) as well as Morzy (1993). Hadzilacos and Papadimitriou (1986) investigate MCSR in more depth. Vidyasankar (1991) proves an alternative characterization of membership in MVSR.

Raz (1993) extends the notion of commit-order preservation (COCSR) that we have introduced in Chapter 3 to a multiversion setting. Commit-order-preserving MVSR, or COMVSR for short, requires view equivalence to a serial monoversion history in which the transaction ordering corresponds to the commit order of the given multiversion history. To this end, Raz restricts the version order and introduces additional constraints between reads and writes.

The scheduling protocols we have discussed in this chapter are also described in Bernstein et al. (1987) or Papadimitriou (1986); for more on protocols, see also Cellary et al. (1988) or Claybrook (1992). Agrawal and Sengupta (1993) describe a possibility to take the aspects of scheduling and version management apart when implementing a protocol. Other approaches are reported by Ahuja and Browne (1987), Bober and Carey (1992a, b), Buckley and Silberschatz (1983), Hadzilacos (1988a), Ibaraki et al. (1990), or Yu and Rosenkrantz (1988).

The practically most relevant, hybrid protocol for read-only transactions has been first proposed by DuBourdieu (1982) and Chan et al. (1982, 1985). Mohan et al. (1992) is a good source for implementation issues of these protocols. A highly enjoyable, semi-technical paper on the general idea of versioning is Schueler (1977).

We have called the concepts discussed in this chapter *transparent* versioning since it is always transparent from a user's point of view. There is another form of versioning, *non-transparent* versioning, which is relevant in application domains such as computer-aided design (CAD) or computer-aided software engineering (CASE), where a user needs to be able to handle versions explicitly. More on this form of versioning can be found, for example, in Cellary and Jomier (1990), Katz (1990), Bernstein (1998), or Härder et al. (2000).