4

NON-LOCKING SCHEDULERS

4.1 INTRODUCTION

In this chapter we will examine two scheduling techniques that do not use locks, timestamp ordering (*TO*) and serialization graph testing (*SGT*). As with 2PL, we'll see aggressive and conservative as well as centralized and distributed versions of both techniques.

We will also look at a very aggressive variety of schedulers, called certifiers. A certifier never delays operations submitted by TMs. It always outputs them right away. When a transaction is ready to commit, the certifier runs a "certification test" to determine whether the transaction was involved in a non-SR execution. If the transaction fails this test, then the certifier aborts the transaction. Otherwise, it allows the transaction to commit. We will describe certifiers based on all three techniques: 2PL, TO, and SGT.

In the final section, we will show how to combine scheduling techniques into composite schedulers. For example, a composite scheduler could use 2PL for part of its synchronization activity and TO for another part. Using the composition rules, you can use the basic techniques we have discussed to construct hundreds of different types of schedulers, all of which produce SR executions.

Unlike 2PL, the techniques described in this chapter are not currently used in many commercial products. Moreover, their performance relative to 2PL is not well understood. Therefore, the material in this chapter is presented

mostly at a conceptual level, with fewer performance comparisons and practical details than in Chapter 3.

4.2 TIMESTAMP ORDERING (TO)

Introduction

In timestamp ordering, the TM assigns a unique timestamp, $ts(T_i)$, to each transaction, T_i. It generates timestamps using any of the techniques described in Section 3.11, in the context of timestamp-based deadlock prevention. The TM attaches a transaction's timestamp to each operation issued by the transaction. It will therefore be convenient to speak of the timestamp of an *operation* $o_i[x]$, which is simply the timestamp of the transaction that issued the operation. A TO scheduler orders conflicting operations according to their timestamps. More precisely, it enforces the following rule, called the TO rule.

TO Rule: If $p_i[x]$ and $q_j[x]$ are conflicting operations, then the DM processes $p_i[x]$ before $q_j[x]$ iff $ts(T_i) < ts(T_j)$.

The next theorem shows that the TO rule produces SR executions.

Theorem 4.1: If H is a history representing an execution produced by a TO scheduler, then H is SR.

Proof: Consider $SG(H)$. If $T_i \rightarrow T_j$ is an edge of $SG(H)$, then there must exist conflicting operations $p_i[x]$, $q_j[x]$ in H such that $p_i[x] < q_j[x]$. Hence by the TO rule, $ts(T_i) < ts(T_j)$. If a cycle $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$ existed in $SG(H)$, then by induction $ts(T_1) < ts(T_1)$, a contradiction. So $SG(H)$ is acyclic, and by the Serializability Theorem, H is SR. □

By enforcing the TO rule, we are ensuring that every pair of conflicting operations is executed in timestamp order. Thus, a TO execution has the same effect as a serial execution in which the transactions appear in timestamp order. In the rest of this section we will present ways of enforcing the TO rule.

Basic TO

Basic TO is a simple and aggressive implementation of the TO rule. It accepts operations from the TM and immediately outputs them to the DM in first-come-first-served order. To ensure that this order does not violate the TO rule, the scheduler rejects operations that it receives too late. An operation $p_i[x]$ is *too late* if it arrives after the scheduler has already output some conflicting operation $q_j[x]$ with $ts(T_j) > ts(T_i)$. If $p_i[x]$ is too late, then it cannot be sched-

uled without violating the TO rule. Since the scheduler has already output $q_j[x]$, it can only solve the problem by rejecting $p_i[x]$.

If $p_i[x]$ is rejected, then T_i must abort. When T_i is resubmitted, it must be assigned a larger timestamp — large enough that its operations are less likely to be rejected during its second execution. Notice the difference with timestamp-based deadlock prevention, where an aborted transaction is resubmitted with the *same* timestamp to avoid cyclic restart. Here it is resubmitted with a *new* and *larger* timestamp to avoid certain rejection.

To determine if an operation has arrived too late, the Basic TO scheduler maintains for every data item x the maximum timestamps of Reads and Writes on x that it has sent to the DM, denoted max-r-scheduled[x] and max-w-scheduled[x] (respectively). When the scheduler receives $p_i[x]$, it compares ts(T_i) to max-q-scheduled[x] for all operation types q that conflict with p. If ts(T_i) < max-q-scheduled[x], then the scheduler rejects $p_i[x]$, since it has already scheduled a conflicting operation with a larger timestamp. Otherwise, it schedules $p_i[x]$ and, if ts(T_i) > max-p-scheduled[x], it updates max-p-scheduled[x] to ts(T_i).

The scheduler must handshake with the DM to guarantee that operations are processed by the DM in the order that the scheduler sent them. Even if the scheduler decides that $p_i[x]$ can be scheduled, it must not send it to the DM until every conflicting $q_j[x]$ that it previously sent has been acknowledged by the DM. Notice that 2PL automatically takes care of this problem. 2PL does not schedule an operation until all conflicting operations previously scheduled have released their locks, which does not happen until after the DM acknowledges those operations.

To enforce this handshake, the Basic TO scheduler also maintains, for each data item x, the number of Reads and Writes that have been sent to, but not yet acknowledged by, the DM. These are denoted r-in-transit[x] and w-in-transit[x] (respectively). For each data item x the scheduler also maintains a queue, queue[x], of operations that *can* be scheduled insofar as the TO rule is concerned, but are waiting for acknowledgments from the DM to previously sent conflicting operations. Conflicting operations are in the queue in timestamp order.

Let us consider a simple scenario to see how the scheduler uses these data structures to enforce the TO Rule. For simplicity, assume that the timestamp of each transaction (or operation) is equal to its subscript (i.e., ts(T_i) = i). We use ack($o_i[x]$) to denote the acknowledgment that the DM sends to the scheduler indicating that $o_i[x]$ has been processed. Suppose initially max-r-scheduled[x] = 0, r-in-transit[x] = 0, and queue[x] is empty.

1. $r_1[x]$ arrives and the scheduler dispatches it to the DM. It sets max-r-scheduled[x] to 1 and r-in-transit[x] to 1.
2. $w_2[x]$ arrives. Although the TO rule says $w_2[x]$ can be scheduled, since r-in-transit[x] = 1 the scheduler must wait until it receives ack($r_1[x]$). It therefore appends $w_2[x]$ to queue[x]. (w-in-transit[x] is unaffected.)

3. $r_4[x]$ arrives and although the TO rule says $r_4[x]$ can be scheduled, the scheduler must wait until it receives ack($w_2[x]$). It therefore appends $r_4[x]$ to queue[x] (after $w_2[x]$). (r-in-transit[x] is unaffected.)
4. $r_3[x]$ arrives. Just like $r_4[x]$, it must wait for $w_2[x]$. So, the scheduler appends it to queue[x] (after $r_4[x]$).
5. ack($r_1[x]$) arrives from the DM. The scheduler decrements r-in-transit[x] to 0. It can now dispatch $w_2[x]$, so it removes $w_2[x]$ from queue[x], sends it to the DM, and sets max-w-scheduled to 2 and w-in-transit[x] to 1. It cannot yet dispatch $r_4[x]$ and $r_3[x]$ because w-in-transit[x] > 0, indicating that the DM has not yet acknowledged some conflicting Write.
6. ack($w_2[x]$) arrives from the DM. The scheduler decrements w-in-transit[x] to 0. Now it can send both $r_4[x]$ and $r_3[x]$ to the DM simultaneously. So, it sets max-r-scheduled to 4 and r-in-transit[x] to 2, and queue[x] becomes empty again.

The principles of operation of a Basic TO scheduler should now be clear. When it receives an operation $p_i[x]$, it accepts it for scheduling if ts(T_i) $\geq$ max-q-scheduled[x] for all operation types q that conflict with p. Otherwise, it rejects $p_i[x]$ and T_i must be aborted. Once $p_i[x]$ is accepted for scheduling, the scheduler dispatches it to the DM immediately, if for all operation types q that conflict with p, q-in-transit[x] = 0 and there are no q operations in queue[x]. Otherwise a conflicting operation $q_j[x]$ is in transit between the scheduler and the DM, or is waiting in queue[x], and so $p_i[x]$ must be delayed; it is therefore inserted in queue[x]. Finally, when it receives ack($p_i[x]$), the scheduler updates p-in-transit[x] accordingly, and removes all the operations in (the head of) queue[x] that can now be dispatched and sends them to the DM.

Strict TO

Although the TO rule enforces serializability, it does not necessarily ensure recoverability. For example, suppose that ts(T_1) = 1 and ts(T_2) = 2, and consider the following history: $w_1[x]$ $r_2[x]$ $w_2[y]$ c_2. Conflicting operations appear in timestamp order. Thus this history could be produced by Basic TO. Yet it is not recoverable: T_2 reads x from T_1, T_2 is committed, but T_1 is not.

As we discussed in Chapters 1 and 2, we usually want the scheduler to enforce an even stronger condition than recoverability, namely, strictness. Here is how Basic TO can be modified to that end.

Recall that w-in-transit[x] denotes the number of $w[x]$ operations that the scheduler has sent to the DM but that the DM has not yet acknowledged. Since two conflicting operations cannot be "in transit" at any time and Writes on the same data item conflict, w-in-transit[x] at any time is either 0 or 1.

The Strict TO scheduler works like Basic TO in every respect, except that it does *not* set w-in-transit[x] to 0 when it receives the DM's acknowledgment

of a $w_i[x]$. Instead it waits until it has received acknowledgment of a_i or c_i. It then sets w-in-transit[x] to zero for every x for which it had sent $w_i[x]$ to the DM. This delays all $r_j[x]$ and $w_j[x]$ operations with $ts(T_j) > ts(T_i)$ until after T_i has committed or aborted. This means that the execution output by the scheduler to the DM is strict. Notice that since T_j waits for T_i only if $ts(T_j) > ts(T_i)$, these waiting situations cannot lead to deadlock.

Note that w-in-transit[x] acts like a lock. It excludes access to x by other transactions until the transaction that owns this "lock" — the transaction that issued the Write that is "in transit" — commits or aborts. This may lead one to believe that we, in effect, turned our TO scheduler into a 2PL scheduler. The following history shows that this is not so:

$$H_1 = r_2[x]\ w_3[x]\ c_3\ w_1[y]\ c_1\ r_2[y]\ w_2[z]\ c_2.$$

H_1 is equivalent to the serial history $T_1\ T_2\ T_3$, and thus is SR. Moreover, it is strict. The only potentially troublesome conflict is $w_1[y] < r_2[y]$, but $c_1 < r_2[y]$, as required for strictness. If $ts(T_1) < ts(T_2) < ts(T_3)$, this history could be produced by the Strict TO scheduler we described. However, H_1 could not possibly have been produced by 2PL. In 2PL, T_2 must release its read lock on x before $w_3[x]$ but may not set its read lock on y until after $w_1[y]$. Since $w_3[x] < w_1[y]$, T_2 would release a lock before obtaining another lock, violating the two phase rule.

It is possible to modify Basic TO to enforce only the weaker conditions of recoverability or cascadelessness (see Exercise 4.2).

Timestamp Management

Suppose we store timestamps in a table, where each entry is of the form [x, max-r-scheduled[x], max-w-scheduled[x]]. This table could consume a lot of space. Indeed, if data items are small, this timestamp information (and o-in-transit) could occupy as much space as the database itself. This is a potentially serious problem.

We can solve the problem by exploiting the following observation. Suppose TMs use relatively accurate real time clocks to generate timestamps, and suppose transactions execute for relatively short periods of time. Then at any given time t, the scheduler can be pretty sure it won't receive any more operations with timestamps smaller than $t - \delta$, where δ is large compared to transaction execution time. The only reason the scheduler needs the timestamps in max-r-scheduled[x] and max-w-scheduled[x], say ts_r and ts_w, is to reject Reads and Writes with even smaller timestamps than ts_r and ts_w. So, once ts_r and ts_w are smaller than $t - \delta$, ts_r and ts_w are of little value to the scheduler, because it is unlikely to receive any operation with a timestamp smaller than ts_r or ts_w.

Using this observation, we can periodically purge from the timestamp table entries that have uselessly small timestamps. Each Purge operation

uses a timestamp ts_{min}, which is the $t - \delta$ value in the previous paragraph. Purge(ts_{min}) removes every entry [x, max-r-scheduled[x], max-w-scheduled[x]] from the timestamp table where max-r-scheduled[x] $< ts_{min}$ and max-w-scheduled[x] $< ts_{min}$. In addition, it tags the table with ts_{min}, indicating that a Purge with that timestamp value has taken place.

Once the timestamp table has been purged, the scheduler must use a modified test to determine whether an operation $o_i[x]$ is too late. First, it looks for an entry for x in the timestamp table. If it finds one, it compares ts(T_i) to max-r-scheduled[x] and/or max-w-scheduled[x] in the usual manner. However, if there is no entry for x, then it must compare ts(T_i) with the ts_{min} that tags the table. If ts(T_i) $\geq ts_{min}$, then if an entry for x was purged from the table, it was irrelevant, and thus $o_i[x]$ is not too late. But if ts(T_i) $< ts_{min}$, then the last Purge might have deleted an entry for x, where either ts(T_i) < max-r-scheduled[x] or ts(T_i) < max-w-scheduled[x]. If that entry still existed, it might tell the scheduler to reject $o_i[x]$. To be safe, the scheduler must therefore reject $o_i[x]$.

If ts_{min} is sufficiently small, it will be rare to reject an $o_i[x]$ with ts(T_i) < ts_{min}. However, the smaller the value of ts_{min}, the smaller the number of entries that Purge will delete, and hence the larger the size of the timestamp table. Therefore, selecting ts_{min} entails a tradeoff between decreasing the number of rejections and minimizing the size of the timestamp table.

Distributed TO schedulers

TO schedulers are especially easy to distribute. Each site can have its own TO scheduler which schedules operations that access the data stored at that site. The decision to schedule, delay, or reject an operation $o_i[x]$ depends only on other operations accessing x. Each scheduler can maintain all the information about the operations accessing the data items it manages. It can therefore go about its decisions independently of the other schedulers. Unlike distributed 2PL, where coordination among distributed schedulers is usually needed to handle distributed deadlocks, distributed TO requires no inter-scheduler communication whatsoever.

Conservative TO

If a Basic TO scheduler receives operations in an order widely different from their timestamp order, then it may reject too many operations, thereby causing too many transactions to abort. This is due to its aggressive nature. We can remedy this problem by designing more conservative schedulers based on the TO rule.

One approach is to require the scheduler to artificially delay each operation it receives for some period of time. To see why this helps avoid rejections, consider some operation $o_i[x]$. The danger in scheduling $o_i[x]$ right away is that the scheduler may later receive a conflicting operation with a smaller

timestamp, which it will therefore have to reject. However, if it holds $o_i[x]$ for a while before scheduling it, then there is a better chance that any conflicting operations with smaller timestamps will arrive in time to be scheduled. The longer the scheduler holds each operation before scheduling it, the fewer rejections it will be forced to make. Like other conservative schedulers, conservative TO delays operations to avoid rejections.

Of course, delaying operations for too long also has its problems, since the delays slow down the processing of transactions. When designing a conservative TO scheduler, one has to strike a balance by adding enough delay to avoid too many rejections without slowing down transactions too much.

An "ultimate conservative" TO scheduler *never* rejects operations and thus never causes transactions to abort. Such a scheduler can be built if we make certain assumptions about the system. As with Conservative 2PL, one such assumption is that transactions predeclare their readset and writeset and the TM conveys this information to the scheduler. We leave the construction of a conservative TO scheduler based on this assumption as an exercise (Exercise 4.11).

In this section we'll concentrate on an ultimate conservative TO scheduler based on a different assumption, namely, that each TM submits its operations to each DM in timestamp order. One way to satisfy this assumption is to adopt the following architecture. At any given time, each TM supervises *exactly* one transaction (e.g., there is one TM associated with each terminal from which users can initiate transactions). Each TM's timestamp generator returns increasing timestamps every time it's called. Thus, each TM runs transactions serially, and each transaction gets a larger timestamp than previous ones supervised by that TM. Of course, since many TMs may be submitting operations to the scheduler in parallel, the scheduler does not necessarily receive operations serially.

Under these assumptions we can build an ultimate conservative TO scheduler as follows. The scheduler maintains a queue, called unsched-queue, containing operations it has received from the TMs but has not yet scheduled. The operations in unsched-queue are kept in timestamp order, the operations with the smallest timestamp being at the head of the queue. Operations with the same timestamp are placed according to the order received, the earlier ones being closer to the head.

When the scheduler receives $p_i[x]$ from a TM, it inserts $p_i[x]$ at the appropriate place in unsched-queue to maintain the order properties just given. The scheduler then checks if the operation at the head of unsched-queue is ready to be dispatched to the DM. The head of unsched-queue, say $q_j[x]$, is said to be *ready* if

1. unsched-queue contains at least one operation from *every* TM, and
2. all operations conflicting with $q_j[x]$ previously sent to the DM have been acknowledged by the DM.

If the head of unsched-queue is in fact ready, the scheduler removes it from unsched-queue and sends it to the DM. The scheduler repeats this activity until the head of unsched-queue is no longer ready.

Ready rule (1) requires that we know if there are operations from all TMs in unsched-queue. One way of doing this efficiently is to maintain, for each TM_v, the count of operations in unsched-queue received from TM_v, denoted op-count[v]. To enable the scheduler to decrement the appropriate op-count when it removes an operation from unsched-queue, unsched-queue should actually store pairs of the form (v, $o_i[x]$), meaning that operation $o_i[x]$ was submitted by TM_v. Each scheduler needs this information anyway, so it knows which TM should receive the acknowledgment for each operation.

Ready rule (2) is a handshake between the scheduler and the DM. This can be implemented as in Basic TO, by keeping for each x a count of the Reads and Writes that are in transit between the scheduler and the DM.

It is easy to see that the ultimate conservative TO scheduler described previously enforces the TO rule. This follows from the fact that the operations in unsched-queue are maintained in timestamp order, and it is always the head of the queue that is sent to the DM. Thus, not only conflicting operations but *all* operations are scheduled in timestamp order. Moreover, the handshake mechanism guarantees that the DM will process conflicting operations in timestamp order.

One problem with this scheduler is that it may get stuck if a TM stops sending operations for a while. To send *any* operation to the DM, the scheduler must have at least one unscheduled operation from *all* TMs. If some TM has no operations to send, then the scheduler is blocked. To avoid this problem, if a TM has no operations to send to the scheduler, it sends a *Null* operation. A Null must carry a timestamp consistent with the requirement that each TM submits operations to the scheduler in timestamp order. When a TM sends a Null to the scheduler, it is promising that every operation it sends in the future will have a timestamp greater than ts(Null). The scheduler treats Nulls just like other operations, except that when a Null becomes the head of the queue, the scheduler simply removes it, and decrements the appropriate op-count, but does *not* send it to the DM.

Particular care must be exercised if a TM fails and is therefore unable to send any operations, whether Null or not, to the scheduler. In this case, the scheduler must somehow be informed of the failure, so that it does not expect operations from that TM. After the TM is repaired and before it starts submitting operations to the scheduler, the latter must be made aware of that fact. Indeed, the TM should not be allowed to submit operations to the scheduler before it is explicitly directed that it may do so.

A second, and maybe more serious, problem with conservative TO schedulers is that they are, true to their name, extremely restrictive. The executions they produce are serial! There are several methods for enhancing the degree of concurrency afforded by conservative TO schedulers.

One way to improve conservative TO is to avoid the serialization of nonconflicting operations by using transaction classes. A *transaction class* is defined by a readset and a writeset. A transaction is *a member of a transaction class* if its readset and writeset are subsets of the class's readset and writeset (respectively). Transaction classes need not be mutually exclusive. That is, they may have overlapping readsets and writesets, so a transaction can be a member of many classes.

We associate each TM with exactly one class and require that each transaction be a member of the class of the TM that supervises it. Conservative TO exploits the association of TMs with transaction classes by weakening ready rule (1). Instead of requiring that operations are received from *all* TMs, the scheduler only needs to have received operations from the TMs associated with transaction classes that contain x in their writeset, if the head of unsched-queue is $r_i[x]$, or x in either their readset or writeset, if the head of the queue is $w_i[x]$. By relaxing the condition under which a queued operation may be sent to the DM, we potentially reduce the time for which operations will be delayed.

Each transaction must predeclare its readset and writeset, so the system can direct it to an appropriate TM. Alternatively, a preprocessor or compiler could determine these sets and thereby the class(es) to which a transaction belongs. The scheduler must know which TMs are associated with which classes. Class definitions and their associations with TMs must remain static during normal operation of the DBS. Changing this information can be done, but usually must be done off-line so that various system components can be simultaneously informed of such changes and modified appropriately.

A more careful analysis of transaction classes can lead to conditions for sending operations to the DM that are even weaker than the one just described. Such an analysis can be conveniently carried out in terms of a graph structure called a *conflict graph*. However, we shall not discuss this technique in this book. Relevant references appear in the Bibliographic Notes.

4.3 SERIALIZATION GRAPH TESTING (SGT)

Introduction

So far, we have seen schedulers that use locks or timestamps. In this section we shall discuss a third type of scheduler, called *serialization graph testing (SGT)* schedulers. An SGT scheduler maintains the SG of the history that represents the execution it controls. As the scheduler sends new operations to the DM, the execution changes, and so does the SG maintained by the scheduler. An SGT scheduler attains SR executions by ensuring the SG it maintains always remains acyclic.

According to the definition in Chapter 2, an SG contains nodes for all committed transactions and for no others. Such an SG differs from the one that

is usually maintained by an SGT scheduler, in two ways. First, the SGT scheduler's SG may not include nodes corresponding to all committed transactions, especially those that committed long ago. Second, it usually includes nodes for all active transactions, which by definition are not yet committed. Due to these differences, we use a different term, *Stored SG* (*SSG*), to denote the SG maintained by an SGT scheduler.

Basic SGT

When an SGT scheduler receives an operation $p_i[x]$ from the TM, it first adds a node for T_i in its SSG, if one doesn't already exist. Then it adds an edge from T_j to T_i for every previously scheduled operation $q_j[x]$ that conflicts with $p_i[x]$. We have two cases:

1. The resulting SSG contains a cycle. This means that if $p_i[x]$ were to be scheduled now (or at any point in the future), the resulting execution would be non-SR. Thus the scheduler rejects $p_i[x]$. It sends a_i to the DM and, when a_i is acknowledged, it deletes from the SSG T_i and all edges incident with T_i. Deleting T_i makes the SSG acyclic again, since all cycles that existed involved T_i. Since the SSG is acyclic, the execution produced by the scheduler now — with T_i aborted — is SR.
2. The resulting SSG is still acyclic. In this case, the scheduler can accept $p_i[x]$. It can schedule $p_i[x]$ immediately, if all conflicting operations previously scheduled have been acknowledged by the DM; otherwise, it must delay $p_i[x]$ until the DM acknowledges all conflicting operations. This handshake can be implemented as in Basic TO. Namely, for each data item x the scheduler maintains queue[x] where delayed operations are inserted in first-in-first-out order, and two counts, r-in-transit[x] and w-in-transit[x], for keeping track of unacknowledged Reads and Writes for each x sent to the DM.

To determine if an operation conflicts with a previously scheduled one, the scheduler can maintain, for each transaction T_i that has a node in SSG, the sets of data items for which Reads and Writes have been scheduled. These sets will be denoted r-scheduled[T_i] and w-scheduled[T_i], respectively. Then, $p_i[x]$ conflicts with a previously scheduled operation of transaction T_j iff $x \in q$-scheduled[T_j], for q conflicting with p.

A significant practical consideration is when the scheduler may discard the information it has collected about a transaction. To detect conflicts, we have to maintain the readset and writeset of every transaction, which could consume a lot of space. It is therefore important to discard this information as soon as possible.

One may naively assume that the scheduler can delete information about a transaction as soon as it commits. Unfortunately, this is not so. For example, consider the (partial) history

$$H_2 = r_{k+1}[x]\ w_1[x]\ w_1[y_1]\ c_1\ w_2[x]\ w_2[y_2]\ c_2\ \ldots\ w_k[x]\ w_k[y_k]c_k.$$

Since SSG(H_2) is acyclic, the execution represented by H_2 could have been produced by an SGT scheduler. Now, suppose that the scheduler receives $w_{k+1}[z]$. According to the SGT policy, the operation can be scheduled iff $z \notin \{x, y_1, \ldots y_k\}$. But for the scheduler to be able to test that, it must remember that x, y_1, ..., y_k were the data items accessed by transactions T_1, T_2, ..., T_k *even though these transactions have committed.*

The scheduler can delete information about a terminated transaction T_i iff T_i could not, at any time in the future, be involved in a cycle of the SSG. For a node to participate in a cycle it must have at least one incoming and one outgoing edge. As in H_2, new edges *out* of a transaction T_i may arise in the SSG even after T_i terminates. However, once T_i terminates no new edges directed *to* it may subsequently arise. Therefore, once a terminated transaction has no incoming edges in the SSG, it cannot possibly become involved in a cycle in the future. So a safe rule for deleting nodes is that *information about a transaction may be discarded as soon as that transaction has terminated and is a source* (i.e., a node with no incoming edges) *in the SSG.* If it is not a source at the time it terminates, then it must wait until all transactions that precede it have terminated and therefore have been deleted from the SSG.

By explicitly checking whether the SSG is acyclic, an SGT scheduler allows any interleaving of Reads and Writes that is SR. In this sense, it is more lenient than TO (which only allows timestamp ordered executions of Reads and Writes) and 2PL (which doesn't allow certain interleavings of Reads and Writes, such as history H_1 in Section 4.2). However, it attains this flexibility at the expense of extra overhead in maintaining the SG and checking for cycles. Moreover, it is currently unknown under what conditions the extra leniency of SGT leads to improved throughput or response time.

Conservative SGT

A conservative SGT scheduler never rejects operations but may delay them. As with 2PL and TO, we can achieve this if each transaction T_i predeclares its readset and writeset, denoted r-set[T_i] and w-set[T_i], by attaching them to its Start operation.

When the scheduler receives T_i's Start, it saves r-set[T_i] and w-set[T_i]. It then creates a node for T_i in the SSG and adds edges $T_j \rightarrow T_i$ for every T_j in the SSG such that p-set[T_i] $\cap$ q-set[T_j] $\neq \{\}$[1] for all pairs of conflicting operation types p and q.

For each data item x the scheduler maintains the usual queue[x] of delayed operations that access x. Conflicting operations in queue[x], say $p_i[x]$ and

[1]We use "{}" to denote the empty set.

$q_j[x]$, are kept in an order consistent with SSG edges. That is, if $T_j \rightarrow T_i$ is in the SSG, then $q_j[x]$ is closer to the head of queue[x] than $p_i[x]$; thus, $q_j[x]$ will be dequeued before $p_i[x]$. The order of nonconflicting operations in queue[x] (i.e., Reads) is immaterial; for specificity, let's say they are kept in order of arrival. When the scheduler receives operation $o_i[x]$ from the TM, it inserts $o_i[x]$ in queue[x] in accordance with the ordering just specified.

The scheduler may send the operation at the head of some queue to the DM iff the operation is "ready." An operation $p_i[x]$ is *ready* if

1. all operations that conflict with $p_i[x]$ and were previously sent to the DM have been acknowledged; and
2. for every T_j that directly precedes T_i in the SSG (i.e., $T_j \rightarrow T_i$ is in the SSG) and for every operation type q that conflicts with p, either $x \notin q$-set[T_j] or $q_j[x]$ has already been received by the scheduler (i.e., $x \in q$-scheduled[T_j]).

Condition (1) amounts to the usual handshake that makes sure the DM processes conflicting operations in the order they are scheduled. Condition (2) is what makes this scheduler avoid aborts. The rationale for it is this. Suppose T_j precedes T_i in the SSG. If the SSG is acyclic, then the execution is equivalent to a serial one in which T_j executes before T_i. Thus if $p_i[x]$ and $q_j[x]$ conflict, $q_j[x]$ must be scheduled before $p_i[x]$. So if $p_i[x]$ is received before $q_j[x]$, it must be delayed. Otherwise, when $q_j[x]$ is eventually received it will have to be rejected, as its acceptance would create a cycle involving T_i and T_j in the SSG. Note that to evaluate condition (2), the Conservative SGT scheduler must, in addition to o-set[T_j], maintain the sets o-scheduled[T_j], as discussed in Basic SGT.

One final remark about condition (2). You may wonder why we have limited it only to transactions that *directly* precede T_i. The reason is that the condition is necessarily satisfied by transactions T_j that indirectly precede T_i; that is, the shortest path from T_j to T_i has more than one edge. Then T_i and T_j do not issue conflicting operations. In particular, $x \in p$-set[T_i] implies $x \notin q$-set[T_j] for all conflicting operation types p, q.

Every time it receives $p_i[x]$ from the TM or an acknowledgment of some $q_j[x]$ from the DM, the scheduler checks if the head of queue[x] is ready. If so, it dequeues the operation and sends it to the DM. The scheduler then repeats the same process with the new head of queue[x] until the queue is empty or its head is not ready. The policy for discarding information about terminated transactions is the same as for Basic SGT.

Recoverability Considerations

Basic and Conservative SGT produce SR histories, but not necessarily recoverable — much less cascadeless or strict — ones.

Both types of SGT schedulers can be modified to produce only strict (and SR) histories by using the same technique as Strict TO. The scheduler sets w-in-transit[x] to 1 when it sends $w_i[x]$ to the DM. But rather than decrementing it back to zero when the DM acknowledges $w_i[x]$, the scheduler does so when it receives an acknowledgment that the DM processed a_i or c_i. Recall that w-in-transit[x] is used to delay sending $r_j[x]$ and $w_j[x]$ operations until a previously sent $w_i[x]$ is processed. By postponing the setting of w-in-transit[x] to zero, the scheduler delays $r_j[x]$ and $w_j[x]$ until the transaction that last wrote into x has terminated, thereby ensuring the execution is strict.

It's also easy to modify Basic or Conservative SGT to enforce only the weaker condition of avoiding cascading abort. For this, it is only necessary to make sure that before scheduling $r_i[x]$, the transaction from which T_i will read x has committed. To do this, every time the scheduler receives an acknowledgment of a Commit operation, c_j, it marks node T_j in the SSG as "committed." Now suppose the scheduler receives $r_i[x]$ from the TM and accepts it. Let T_j be a transaction that satisfies:

1. $x \in$ w-scheduled[T_j]; and
2. for any $T_k \neq T_j$ such that $x \in$ w-scheduled[T_k], $T_k \rightarrow T_j$ is in the SSG.

At most one T_j can satisfy both of these conditions. (It is possible that no transaction does, for instance, if all transactions that have ever written into x have been deleted from the SSG by now.) The scheduler can send $r_i[x]$ to the DM only if T_j is marked "committed" or no such T_j exists.

The same idea can be used to enforce only the weaker condition of recoverability. The difference is that instead of delaying individual Reads, the scheduler now delays T_i's Commit until all transactions T_j from which T_i has read either are marked "committed" or have been deleted from the SSG. Also, since in this case cascading aborts are possible, when the scheduler either receives T_i's Abort from the TM or causes T_i to abort to break a cycle in the SSG, it also aborts any transaction T_j that read from T_i. An SGT scheduler can detect if T_j has read from T_i by checking if

1. $T_i \rightarrow T_j$ is in the SSG, and
2. there is some $x \in$ r-scheduled[T_j] $\cap$ w-scheduled[T_i] such that for every T_k where $T_i \rightarrow T_k$ and $T_k \rightarrow T_j$ are in the SSG, $x \notin$ w-scheduled[T_k].

Distributed SGT Schedulers

SGT schedulers present problems in distribution of control since their decisions are based on the SSG, an inherently global structure. The problems are reminiscent of distributed deadlocks. If each scheduler maintains a local SSG reflecting only the conflicts on the data items that it manages, then it is possible to construct executions in which all such local SSGs are acyclic, yet the global SSG contains a cycle.

For example, consider

$$H_3 = w_1[x_1]\ r_2[x_1]\ w_2[x_2]\ r_3[x_2]\ w_3[x_3]\ r_4[x_3]\ w_4[x_4]\ \ldots\ r_k[x_{k-1}]\ w_k[x_k].$$

Suppose that there are k sites, and x_i is stored at site i, for $1 \leq i \leq k$. At each site $i < k$, the local SSG contains the edge $T_i \rightarrow T_{i+1}$. Now, if T_1 issues $w_1[x_k]$, the local SSG at site k contains the edge $T_k \rightarrow T_1$. Thus, globally we have a cycle $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_k \rightarrow T_1$, yet all local SSGs are acyclic (each consists of a single edge).

This is essentially the same problem we had with global deadlock detection in 2PL (Section 3.11). There is an important difference, however, that makes our new problem more severe. In global deadlock detection any transactions involved in a cycle in the WFG are just waiting for each other and thus none can proceed; in particular, none can commit. So we may take our time in checking for global cycles, merely at the risk of delaying the discovery of a deadlock. On the other hand, transactions that lie along an SSG cycle do *not* wait for each other. Since a transaction should not commit until the scheduler has checked that the transaction is not in an SSG cycle, global cycle detection must take place *at least* at the same rate as transactions are processed. In typical applications, the cost of this is prohibitive.

*Space-Efficient SGT Schedulers

The implementations of SGT just described have the unpleasant property of potentially requiring an unbounded amount of space. History

$$H_2 = r_{k+1}[x]\ w_1[x]\ w_1[y_1]\ c_1\ w_2[x]\ w_2[y_2]\ c_2\ \ldots\ w_k[x]\ w_k[y_k]\ c_k$$

is a model for histories with an arbitrary number of committed transactions about which the scheduler must keep readset and writeset information. Recall that in the implementation we discussed, each transaction T_i that appears in the SSG requires space for representing T_i and, more substantially, for storing r-scheduled$[T_i]$ and w-scheduled$[T_i]$. If t is the total number of transactions appearing in an execution such as H_2, and D is the set of data items, then the scheduler may require space proportional to t^2 for storing the SSG and space proportional to $t \cdot |D|$ for other information maintained by the scheduler. Since there is no bound on the number of transactions in the execution, the scheduler's space requirements can grow indefinitely.

In all of the other schedulers we have studied, the space required is proportional to $a \cdot |D|$, where a is the number of *active* transactions (i.e., those that have not yet committed or aborted). We can make SGT's space requirements comparable to those schedulers by using a different implementation, which requires space proportional to the maximum of a^2 (for the SSG) and $a \cdot |D|$ (for the conflict information). Since at any given time a is generally much smaller than the total number of transactions that have appeared in that execution and is a number under system control, this is a significant improvement over Basic SGT.

In outline, here is the space-efficient implementation for Basic SGT. The scheduler maintains an SSG which is transitively closed and only contains nodes for active transactions. For each node T_i of SSG, the scheduler also maintains four sets: o-scheduled[T_i] and o-conflict[T_i], for $o \in \{r, w\}$. As before, r-scheduled[T_i] (or w-scheduled[T_i]) is the set of data items for which Reads (or Writes) have been scheduled. r-conflict[T_i] (or w-conflict[T_i]) is the set of data items read (or written) by terminated transactions that must follow T_i in any execution equivalent to the present one. More precisely, at the end of an execution represented by history H, o-conflict[T_i] = $\{x \mid o_j[x] \in H$, T_j is committed in H, and $T_i \rightarrow T_j \in SG^+(H)\}$, where $SG^+(H)$ is the transitive closure of $SG(H)$.[2]

When a transaction T_i begins, the scheduler adds a new node to the SSG, and initializes o-scheduled[T_i] and o-conflict[T_i] to the empty set, for $o \in \{r, w\}$. When the scheduler receives an operation $p_i[x]$, it tests whether x is in q-conflict[T_i] for some q conflicting with p. If so, it rejects $p_i[x]$. Otherwise, it adds edges $T_j \rightarrow T_i$ to the SSG for every transaction T_j with $x \in q$-scheduled[T_j] $\cup$ q-conflict[T_j] for all operation types q conflicting with p. If the resulting SSG contains a cycle, the scheduler rejects $p_i[x]$. Otherwise, it schedules $p_i[x]$ after all conflicting operations previously sent to the DM have been acknowledged.

When the scheduler receives the acknowledgment of c_i from the DM, it computes SSG^+. For each T_j with $T_j \rightarrow T_i$ in SSG^+, it sets o-conflict[T_j] to o-conflict[T_j] $\cup$ o-scheduled[T_i] $\cup$ o-conflict[T_i], $o \in \{r, w\}$. Note that at this point, r-scheduled[T_i] (or w-scheduled[T_i]) is precisely the readset (or writeset) of T_i. Then the scheduler deletes T_i from SSG^+ along with all its incoming and outgoing edges. The resulting graph becomes the new SSG.

To see why this method is correct, first consider the scheduler's response to receiving $p_i[x]$. Let history H represent the execution at the time the scheduler receives $p_i[x]$. If x is in q-conflict[T_i] for some q conflicting with p, then for some committed transaction T_j, some operation $q_j[x]$ has been scheduled and $T_i \rightarrow T_j \in SG^+(H)$. If $p_i[x]$ were to be scheduled, we would also have that $T_j \rightarrow T_i$ (because of $q_j[x]$ and $p_i[x]$) and thus a cyclic SG would result. To avoid this, $p_i[x]$ is rejected.

Next, consider the scheduler's response to c_i, and let history H represent the execution at this time. Consider any active transaction T_j such that $T_j \rightarrow T_i$ is in $SG^+(H)$ (which implies $T_j \rightarrow T_i$ is in SSG^+). Before discarding T_i we must record the fact that the scheduler must not subsequently schedule any of T_j's operations that conflict with T_i's operations or with operations of committed transactions that follow T_i in SG^+. Otherwise, a cyclic SG will subsequently arise. This is why we must, at this point, update r-conflict[T_j] (or w-conflict[T_j]) by adding r-scheduled[T_i] $\cup$ r-conflict[T_i] (or w-scheduled[T_i] $\cup$ w-conflict[T_i]) to it.

---

[2]See Section A.3 of the Appendix for the definition of transitive closure of a directed graph.

We compute SSG^+ before deleting the node corresponding to T_i, the transaction that committed, to avoid losing indirect precedence information (represented as paths in the SSG). For example, suppose that there is a path from T_j to T_k in the SSG and that *all* such paths pass through node T_i. Then, deleting T_i will produce a graph that doesn't represent the fact that T_j must precede T_k. SSG^+ contains an edge $T_j \rightarrow T_k$ iff the SSG has a *path* from T_j to T_k. Thus SSG^+ represents exactly the same precedence information as SSG. Moreover, the *only* precedence information lost by deleting T_i from the SSG^+ pertains to T_i itself (in which we are no longer interested since it has terminated) and to no other active transactions.

The scheduler as described produces SR executions. By using the techniques of the previous subsection we can further restrict its output to be recoverable, cascadeless, or strict (see Exercise 4.19).

4.4 CERTIFIERS

Introduction

So far we have been assuming that *every* time it receives an operation, a scheduler must decide whether to accept, reject, or delay it. A different approach is to have the scheduler immediately schedule each operation it receives. From time to time, it checks to see what it has done. If it thinks all is well, it continues scheduling. On the other hand, if it discovers that in its hurry to process operations it has inappropriately scheduled conflicting operations, then it must abort certain transactions.

When it's about to schedule a transaction T_i's Commit, the scheduler checks whether the execution that includes c_i is SR. If not, it rejects the Commit, thereby forcing T_i to abort. (It cannot check less often than on every Commit, as it would otherwise risk committing a transaction involved in a non-SR execution.) Such schedulers are called *certifiers*. The process of checking whether a transaction's Commit can be safely scheduled or must be rejected is called *certification*. Certifiers are sometimes called *optimistic schedulers*, because they aggressively schedule operations, hoping nothing bad, such as a non-SR execution, will happen.

There are certifiers based on all three types of schedulers — 2PL, TO, and SGT — with either centralized or distributed control. We will explore all of these possibilities in this section.

2PL Certification

When a 2PL certifier receives an operation from the TM, it notes the data item accessed by the operation and immediately submits it to the DM. When it receives a Commit, c_i, the certifier checks if there is any operation $p_i[x]$ of T_i that conflicts with some operation $q_j[x]$ of some other active transaction, T_j. If

so, the certifier rejects c_i and aborts T_i.[3] Otherwise it *certifies* T_i by passing c_i to the DM, thereby allowing T_i to terminate successfully.

The 2PL certifier uses several data structures: a set containing the names of active transactions, and two sets, r-scheduled[T_i] and w-scheduled[T_i], for each active transaction T_i, which contain the data items read and written, respectively, by T_i so far.

When the 2PL certifier receives $r_i[x]$ (or $w_i[x]$), it adds x to r-scheduled[T_i] (or w-scheduled[T_i]). When the scheduler receives c_i, T_i has finished executing, so r-scheduled[T_i] and w-scheduled[T_i] contain T_i's readset and writeset, respectively. Thus, testing for conflicts can be done by looking at intersections of the r-scheduled and w-scheduled sets. To process c_i, the certifier checks every other active transaction, T_j, to determine if any one of r-scheduled[T_i] $\cap$ w-scheduled[T_j], w-scheduled[T_i] $\cap$ r-scheduled[T_j], or w-scheduled[T_i] $\cap$ w-scheduled[T_j] is nonempty. If so, it rejects c_i. Otherwise, it certifies T_i and removes T_i from the set of active transactions.

To prove that the 2PL certifier only produces SR executions, we will follow the usual procedure of proving that every history it allows must have an acyclic SG.

Theorem 4.2: The 2PL certifier produces SR histories.

Proof: Let H be a history representing an execution produced by the 2PL certifier. Suppose $T_j \rightarrow T_i$ is an edge of SG(H). Then T_i and T_j are committed in H and there are conflicting operations $q_j[x]$ and $p_i[x]$ such that $q_j[x] < p_i[x]$. We claim that the certification of T_j preceded the certification of T_i. For suppose T_i was certified first. At the time T_i is certified, $p_i[x]$ must have been processed by the scheduler; hence x is in p-scheduled[T_i]. Moreover, since $q_j[x] < p_i[x]$, x is in q-scheduled[T_j]. Thus p-scheduled[T_i] $\cap$ q-scheduled[T_j] $\neq \{\}$. T_j, not having been certified yet, is active. But then the scheduler would not certify T_i, contrary to our assumption that T_i is committed in H. We have therefore shown that if $T_j \rightarrow T_i$ is in SG(H), T_j must be certified before T_i in H. By induction, if a cycle existed in SG(H), every transaction on that cycle would have to be certified before itself, an absurdity. Therefore SG(H) is acyclic. By the Serializability Theorem, H is SR. □

We called this certifier a 2PL certifier, yet there was no mention of locks anywhere! The name is justified if one thinks of an imaginary read (or write) lock being obtained by T_i on x when x is added to r-scheduled[T_i] (or w-scheduled[T_i]). If there is ever a lock conflict between two active transactions,

[3] A transaction is committed *only* when the scheduler acknowledges to the TM the processing of Commit, *not* when the TM sends the Commit to the scheduler. So, it is perfectly legitimate for the scheduler to reject c_i and abort T_i at this point.

the first of them to attempt certification will be aborted. This is very similar to a 2PL scheduler that never allows a conflicting operation to wait, but rather always rejects it. In fact, the committed projection of every history produced by a 2PL certifier could also have been produced by a 2PL scheduler.

To enforce recoverability, when a 2PL certifier aborts a transaction T_i, it must also abort any other active transaction T_j such that w-scheduled[T_i] $\cap$ r-scheduled[T_j] $\neq \{\}$. Note that this may cause T_j to be aborted unnecessarily if, for example, there are data items in w-scheduled[T_i] $\cap$ r-scheduled[T_j] but T_j actually read them *before* T_i wrote them. However, the 2PL certifier does not keep track of the order in which conflicting operations were processed; it can't distinguish, at certification time, the case just described from the case in which T_j read some of the items in w-scheduled[T_i] $\cap$ r-scheduled[T_j] *after* T_i wrote them. For safety, then, it must abort T_j.

One can modify the 2PL certifier to enforce the stronger conditions of cascadelessness or strictness, although this involves delaying operations and therefore runs counter to the optimistic philosophy of certifiers (see Exercise 4.25).

To understand the performance of 2PL certification, let's compare it to its on-line counterpart, Basic 2PL. Both types of scheduler check for conflicts between transactions. Thus, the overhead for checking conflicts in the two methods is about the same. If transactions rarely conflict, then Basic 2PL doesn't delay many transactions and neither Basic 2PL nor 2PL certification aborts many. Thus, in this case throughput for the two methods is about the same.

At higher conflict rates, 2PL certification performs more poorly. To see why, suppose T_i issues an operation that conflicts with that of some other active transaction T_j. In 2PL certification, T_i and T_j would execute to completion, even though at least one of them is doomed to be aborted. The execution effort in completing that doomed transaction is wasted. By contrast, in 2PL T_i would be delayed. This ensures that at most one of T_i and T_j will be aborted due to the conflict. Even if delaying T_i causes a deadlock, the victim is aborted before it executes completely, so less of its execution effort is wasted than in 2PL certification.

Quantitative studies are consistent with this intuition. In simulation and analytic modelling of the methods, 2PL certification has lower throughput than 2PL for most application parameters. The difference in throughput increases with increasing conflict rate.

SGT Certification

SGT lends itself very naturally to a certifier. The certifier dynamically maintains an SSG of the execution it has produced so far, exactly as in Basic SGT. Every time it receives an operation $p_i[x]$, it adds the edge $T_j \rightarrow T_i$ to the SSG for every transaction T_j such that the certifier has already sent to the DM an

operation $q_j[x]$ conflicting with $p_i[x]$. After this is done, it immediately dispatches $p_i[x]$ to the DM (without worrying whether the SSG is acyclic). Of course, handshaking is still necessary to ensure that the DM processes conflicting operations in the order scheduled.

When the scheduler receives c_i, it checks whether T_i lies on a cycle of the SSG. If so, it rejects c_i and T_i is aborted. Otherwise it certifies T_i and T_i commits normally.

The implementation issues are essentially those that we discussed for Basic SGT. The same data structures can be used to maintain the SSG and to enforce handshaking between the certifier and DM. The problem of space inefficiency is of concern here also, and the same remedies apply.

Finally, the SGT certifier is modified to enforce recoverability in essentially the same way as SGT schedulers. The rule is that if T_i aborts, the certifier also aborts any active transaction T_j such that, for some data item x, $x \in$ w-scheduled$[T_i] \cap$ r-scheduled$[T_j]$, $T_i \rightarrow T_j$ is in the SSG, and for every T_k such that $T_i \rightarrow T_k$ and $T_k \rightarrow T_j$ are in the SSG, $x \notin$ w-scheduled$[T_k]$.

TO Certification

A TO certifier schedules Reads and Writes without delay, except for reasons related to handshaking between the certifier and the DM. When the certifier receives c_i, it certifies T_i if all conflicts involving operations of T_i are in timestamp order. Otherwise, it rejects c_i and T_i is aborted. Thus, T_i is certified iff the execution so far satisfies the TO rule. That is, in the execution produced thus far, if some operation $p_i[x]$ precedes some conflicting operation $q_j[x]$ of transaction T_j, then $\text{ts}(T_i) < \text{ts}(T_j)$. This is the very same condition that Basic TO checks when it receives each operation. However, when Basic TO finds a violation of the TO rule, it immediately rejects the operation, whereas the TO certifier delays this rejection until it receives the transaction's Commit. Since allowing such a transaction to complete involves extra work, with no hope that it will ultimately commit, Basic TO is preferable to a TO certifier.

Distributed Certifiers

A distributed certifier consists of a collection of certifier processes, one for each site. As with distributed schedulers, we assume that the certifier at a site is responsible for regulating access to exactly those data items stored at that site.

Although each certifier sends operations to its respective DM independently of other certifiers, the activity of transaction certification must be carried out in a coordinated manner. To certify a transaction, a decision must be reached involving all of the certifiers that received an operation of that transaction. In SGT certification, the certifiers must exchange their local SSGs to ensure that the global SSG does not have a cycle involving the transaction

being certified. If no such cycle exists, then the transaction is certified (by all certifiers involved); otherwise it is aborted.

In 2PL or TO certification, each certifier can make a local decision whether or not to certify a transaction, based on conflict information for the data items it manages. A global decision must then be reached by consensus. If the local decision of *all* certifiers involved in the certification of a transaction is to certify the transaction, then the global decision is to certify. If even one certifier's local decision is to abort the transaction, then the global decision is to abort. The fate of a transaction is decided only after this global decision has been reached. A certifier *cannot* proceed to certify a transaction on the basis of its local decision only.

This kind of consensus can be reached by using the following communication protocol between the TM that is supervising T_i and the certifiers that processed T_i's operations. The TM distributes T_i's Commit to all certifiers that participated in the execution of T_i. When a certifier receives c_i, it makes a local decision, called its *vote*, on whether to certify T_i or not, and sends its vote to the TM. After receiving the votes from all the certifiers that participate in T_i's certification, the TM makes the global decision accordingly. It then sends the global decision to all participating certifiers, which carry it out as soon as they receive it.[4]

Using this method for reaching a unanimous decision, a certifier may vote to certify a transaction, yet end up having to abort it because some other certifier voted not to certify. Thus, a certifier that votes to certify experiences a period of uncertainty on the fate of the transaction, namely, the period between the moment it sends its vote and the moment it receives the global decision from the TM. Of course, a certifier that votes to abort is not uncertain about the transaction. It knows it will eventually be aborted by all certifiers.

4.5 INTEGRATED SCHEDULERS

Introduction

The schedulers we have seen so far synchronize conflicting operations by one of three mechanisms: 2PL, TO, or SGT. There are other schedulers that use combinations of these techniques to ensure that transactions are processed in an SR manner. Such schedulers are best understood by decomposing the problem of concurrency control into certain subproblems. Each subproblem is solved by one of our familiar three techniques. Then we have to make sure that these solutions fit together consistently to yield a correct solution to the entire problem of scheduling.

---

[4]We'll study this type of protocol in much greater detail in Chapter 7.

We decompose the problem by separating the issue of scheduling Reads against conflicting Writes (and, symmetrically, of Writes against conflicting Reads) from that of scheduling Writes against conflicting Writes. We'll call the first subproblem *rw synchronization* and the second, *ww synchronization*. We will call an algorithm used for rw synchronization an *rw synchronizer*, and one for ww synchronization a *ww synchronizer*. A complete scheduler consists of an rw and a ww synchronizer. In a correct scheduler, the two synchronizers must resolve read-write and write-write conflicts in a consistent way. Schedulers obtained in this manner are called *integrated schedulers*. Integrated schedulers that use (possibly different versions of) the same mechanism (2PL, TO, or SGT) for both the rw and the ww synchronizer are called *pure schedulers*. All of the schedulers we have seen so far are pure schedulers. Schedulers combining different mechanisms for rw and ww synchronization are called *mixed schedulers*.

To use any of the variations of the three concurrency control techniques that we have seen to solve each of these two subproblems, all we need to do is change the definition of "conflicting operations" to reflect the type of synchronization we are trying to achieve. Specifically, in rw synchronization, two operations accessing the same data item conflict if one of them is a Read and the other is a Write. Two Writes accessing the same data item are *not* considered to conflict in this case. Similarly, in ww synchronization two operations on the same data item conflict if *both* are Writes.

For example, if a scheduler uses 2PL for rw synchronization only, $w_i[x]$ is delayed (on account of the 2PL rw synchronizer) *only* if some other transaction T_j is holding a *read* lock on x. That is, several transactions *may* be sharing a write lock on x, as long as no transaction is concurrently holding a read lock on x. This may sound wrong, but remember that there is another part to the scheduler, the ww synchronizer, which will somehow ensure that write-write conflicts are resolved consistently with the way the 2PL rw synchronizer resolves read-write conflicts. Similarly, in a scheduler using 2PL for ww synchronization only, Reads are never delayed by the 2PL ww synchronizer, although they may be delayed by the rw synchronizer.

A TO rw synchronizer only guarantees that Reads and Writes accessing the same data item are processed in timestamp order. It will *not* force two Writes $w_i[x]$ and $w_j[x]$ to be processed in timestamp order, unless there is some operation $r_k[x]$ such that $\text{ts}(T_i) < \text{ts}(T_k) < \text{ts}(T_j)$ or $\text{ts}(T_j) < \text{ts}(T_k) < \text{ts}(T_i)$. Similarly, a TO ww synchronizer ensures that $w_i[x]$ is processed before $w_j[x]$ only if $\text{ts}(T_i) < \text{ts}(T_j)$, but imposes no such order on $r_i[x]$ and $w_j[x]$; of course, the rw synchronizer will impose such an order.

In SGT, the SG maintained by the scheduler contains only those edges that reflect the kind of conflicts being resolved. An SGT rw (or ww) synchronizer adds edges corresponding to read-write (or write-write) conflicts, every time it receives an operation.

The *rw serialization graph of history H*, denoted $SG_{rw}(H)$, consists of nodes corresponding to the committed transactions appearing in H and edges $T_i \rightarrow T_j$ iff $r_i[x] < w_j[x]$ or $w_i[x] < r_j[x]$ for some x. Similarly, the *ww serialization graph of H*, denoted $SG_{ww}(H)$, consists of nodes corresponding to the committed transactions appearing in H and edges $T_i \rightarrow T_j$ iff $w_i[x] < w_j[x]$ for some x.

It is easy to show that if a history H represents some execution produced by a scheduler using an rw (or ww) synchronizer based on 2PL, TO, or SGT, then $SG_{rw}(H)$ (or $SG_{ww}(H)$) is acyclic. The arguments are similar to those used in earlier sections to prove that $SG(H)$ is acyclic if H represents an execution produced by a 2PL, TO, or SGT scheduler. Therefore for a scheduler using any combination of 2PL, TO, and SGT for rw or ww synchronization, we know that if H is produced by the scheduler, then $SG_{rw}(H)$ and $SG_{ww}(H)$ are both acyclic.

To ensure that H is SR, we need the complete graph $SG(H)$ to be acyclic. It is not enough for each of $SG_{rw}(H)$ and $SG_{ww}(H)$ to be acyclic. In addition, we need the two graphs to represent *compatible* partial orders. That is, the *union* of the two graphs must be acyclic. Ensuring the compatibility of these two graphs is the hardest part of designing correct integrated schedulers, as we'll see later in this section.

Thomas' Write Rule (TWR)

Suppose a TO scheduler receives $w_i[x]$ after it has already sent $w_j[x]$ to the DM, but $ts(T_i) < ts(T_j)$. The TO rule requires that $w_i[x]$ be rejected. But if the scheduler is only concerned with ww synchronization, then this rejection is unnecessary. For if the scheduler had processed $w_i[x]$ when it was "supposed to," namely, before it had processed $w_j[x]$, then x would have the same value as it has now, when it is faced with $w_i[x]$'s having arrived too late. Said differently, processing a sequence of Writes in timestamp order produces the same result as processing the single Write with maximum timestamp; thus, late operations can be ignored.

This observation leads to a ww synchronization rule, called *Thomas' Write Rule (TWR)*, that never delays or rejects any operation. When a TWR ww synchronizer receives a Write that has arrived too late insofar as the TO rule is concerned, it simply *ignores* the Write (i.e., doesn't send it to the DM) but reports its successful completion to the TM.

More precisely, Thomas' Write Rule states: Let T_j be the transaction with maximum timestamp that wrote into x before the scheduler receives $w_i[x]$. If $ts(T_i) > ts(T_j)$, process $w_i[x]$ as usual (submit it to the DM, wait for the DM's $ack(w_i[x])$, and then acknowledge it to the TM). Otherwise, process $w_i[x]$ by simply acknowledging it to the TM.

But what about Reads? Surely Reads care about the order in which Writes are processed. For example, suppose we have four transactions T_1, T_2, T_3, and

T_4 where $ts(T_1) < ts(T_2) < ts(T_3) < ts(T_4)$; T_1, T_2, and T_4 just write x; and T_3 reads x. Now, suppose the scheduler schedules $w_1[x]$, $r_3[x]$, $w_4[x]$ in that order, and then receives $w_2[x]$. TWR says that it's safe for the ww synchronizer to accept $w_2[x]$ but not process it. But this seems wrong, since T_2 should have written x before $r_3[x]$ read the value written by $w_1[x]$. This is true. But the problem is one of synchronizing Reads against Writes and therefore none of the ww synchronizer's business. The rw synchronizer must somehow prevent this situation.

This example drives home the division of labor between rw and ww synchronizers. And it emphasizes once more that care must be taken in integrating rw and ww synchronizers to obtain a correct scheduler.

We will examine two integrated schedulers, one using Basic TO for rw synchronization and TWR for ww synchronization, and another using 2PL for rw synchronization and TWR for ww synchronization. The first is a pure integrated scheduler because both rw and ww synchronization are achieved by the same mechanism, TO. The second is a mixed integrated scheduler because it combines a 2PL rw synchronizer with a TWR ww synchronizer.

A Pure Integrated Scheduler

Our first integrated scheduler can be viewed as a simple optimization over Basic TO, in the sense that it sometimes avoids rejecting Writes that Basic TO would reject.

Recall that a Basic TO scheduler schedules an operation if all conflicting operations that it has previously scheduled have smaller timestamps. Otherwise it rejects the operation. Our new scheduler uses this principle for rw synchronization only. For ww synchronization it uses TWR instead. That is, it behaves as follows:

1. It schedules $r_i[x]$ provided that for all $w_j[x]$ that have already been scheduled, $ts(T_i) > ts(T_j)$. Otherwise, it rejects $r_i[x]$.
2. It rejects $w_i[x]$ if it has already scheduled some $r_j[x]$ with $ts(T_j) > ts(T_i)$. Otherwise, if it has scheduled some $w_j[x]$ with $ts(T_j) > ts(T_i)$, it ignores $w_i[x]$ (i.e., acknowledges the processing of $w_i[x]$ to the TM but does not send the operation to the DM). Otherwise, it processes $w_i[x]$ normally.

Thus, the difference from Basic TO is that a late Write is only rejected if a conflicting *Read* with a greater timestamp has already been processed. If the only conflicting operations with greater timestamps that have been processed are Writes, the late Write is simply ignored. The implementation details and recoverability considerations of this scheduler are similar to those of Basic TO (see Exercise 4.33).

A Mixed Integrated Scheduler

Now let's construct an integrated scheduler that uses Strict 2PL for rw synchronization and TWR for ww synchronization. Suppose that H is a history repre-

senting some execution of such a scheduler. We know that $SG_{rw}(H)$ and $SG_{ww}(H)$ are each acyclic. To ensure that $SG(H) = SG_{rw}(H) \cup SG_{ww}(H)$ is acyclic, we will also require that

1. if $T_i \rightarrow T_j$ is an edge of $SG_{rw}(H)$, then $ts(T_i) < ts(T_j)$.

We know that TO synchronizes all conflicts in timestamp order, so if $T_i \rightarrow T_j$ is in $SG_{ww}(H)$ then $ts(T_i) < ts(T_j)$. By (1), the same holds in $SG_{rw}(H)$. Since every edge $T_i \rightarrow T_j$ in $SG_{rw}(H)$ and $SG_{ww}(H)$ has $ts(T_i) < ts(T_j)$, $SG(H)$ must be acyclic. The reasoning is identical to that of Theorem 4.1, which proved that TO is correct. In the remainder of this section we describe a technique for making condition (1) hold.

If $T_i \rightarrow T_j$ is in $SG_{rw}(H)$, then T_j cannot terminate until T_i releases some lock that T_j needs. In Strict 2PL, a transaction holds its locks until it commits. Therefore, if $T_i \rightarrow T_j$, then T_i commits before T_j terminates, which implies that T_i commits before T_j commits. Thus, we can obtain (1) by ensuring that

2. if T_i commits before T_j, then $ts(T_i) < ts(T_j)$.

The scheduler can't ensure (2) unless it delays assigning a timestamp to a transaction T_i until it receives T_i's Commit. But the scheduler cannot process any of T_i's Writes using TWR until it knows T_i's timestamp. These last two observations imply that the scheduler must delay processing all of the Writes it receives from each transaction T_i until it receives T_i's Commit.

This delaying of Writes creates a problem if, for some x, T_i issues $r_i[x]$ after having issued $w_i[x]$. Since $w_i[x]$ is still delayed in the scheduler when the scheduler receives $r_i[x]$, the scheduler can't send $r_i[x]$ to the DM; if it did, then $r_i[x]$ would not (as it should) read the value produced by $w_i[x]$.

The scheduler can deal with this problem by checking a transaction's queue of delayed Writes every time it receives a Read to process. If the transaction previously wrote the data item, the scheduler can service the Read internally, without going to the DM. Since the scheduler has to get a read lock before processing the Read, it will see the transaction's write lock at that time, and will therefore know to process the Read internally. If there is no write lock owned by the same transaction, then the scheduler processes the Read normally.

In a centralized DBS, the scheduler can enforce (2) simply by assigning each transaction a larger timestamp than the previous one that committed. In a distributed DBS, this is difficult to do, because different schedulers (and TMs) are involved in committing different transactions. Therefore, for distributed DBSs we need a way to assign timestamps to different transactions independently at different sites. The following is one method for accomplishing this.

Each scheduler maintains the usual information used by 2PL and TO schedulers. Each transaction has a timestamp, which its TM assigns according to a rule that we will describe later on. For each data item x (that it manages), a scheduler maintains read locks and write locks (used for 2PL rw synchroniza-

tion), and a variable max-w-scheduled[x] containing the largest timestamp of all transactions that have written into x (used for the TWR ww synchronization). (Since TO is not used for rw synchronization, max-r-scheduled[x] is unnecessary.) To coordinate handshaking with the DM, the scheduler also keeps the usual queue of delayed operations on x, queue[x], and two counts, r-in-transit[x] and w-in-transit[x], of the Reads and Writes sent to, but not yet acknowledged by, the DM. The scheduler also maintains a variable to help in generating transaction timestamps; for each data item x, max-ts[x] contains the largest timestamp of all transactions that have ever obtained a lock on x (be it a read or a write lock). And for each active transaction T_i, the TM keeps a variable max-lock-set[T_i], which it initializes to 0 when T_i begins executing.

Each scheduler uses 2PL for rw synchronization. To process $r_i[x]$, the scheduler obtains $rl_i[x]$. To process $w_i[x]$, it sets $wl_i[x]$, after which it inserts $w_i[x]$ into w-queue[T_i], a buffer that contains T_i's Writes. After processing $o_i[x]$ ($o \in \{r, w\}$), the scheduler sends the TM an acknowledgment that includes max-ts[x]. The TM processes the acknowledgment by setting max-lock-set[T_i] := max(max-lock-set[T_i], max-ts[x]).

When the TM receives c_i, it generates a unique timestamp for T_i greater than max-lock-set[T_i]. It then sends c_i and ts(T_i) to each scheduler that processed operations on behalf of T_i. To process c_i, a scheduler sets max-ts[x] := max(ts(T_i), max-ts[x]) for each x that T_i has locked (at that scheduler). Then, the scheduler processes T_i's Writes. For every $w_i[x]$ buffered in w-queue[T_i], if ts(T_i) > max-w-scheduled[x], then the scheduler sends $w_i[x]$ to the DM and sets max-w-scheduled[x] to ts(T_i); otherwise it discards $w_i[x]$ without processing it. Notice that it's crucial that no $w_i[x]$ was sent to the DM before the scheduler receives c_i and ts(T_i), because the latter is needed for processing $w_i[x]$ according to TWR. After a DM acknowledges all of the Writes that were sent to it, its scheduler sends c_i to the DM. After it receives ack(c_i), the scheduler can release T_i's locks (that were set by that scheduler) and acknowledge T_i's commitment (by that scheduler) to the TM.

We want to show that the implementation sketched here achieves condition (1), which is sufficient for proving that Strict 2PL rw synchronization and TWR ww synchronization are integrated correctly. Let H be a history representing an execution of the DBS just outlined and suppose that $T_i \rightarrow T_j$ is in $SG_{rw}(H)$. This means that $p_i[x] < q_j[x]$ for some x where p is r or w and q is w or r, respectively. Because we are using Strict 2PL for rw synchronization, T_i committed before T_j did. Consider the time at which T_j is ready to commit. At that time max-lock-set[T_j] $\geq$ max-ts[x], since T_j is holding a q lock on x, and after it obtained that lock, the TM set max-lock-set[T_j] to at least max-ts[x]. But before T_i released its p lock on x, the scheduler set max-ts[x] to at least ts(T_i). Since max-ts[x] never decreases with time, max-lock-set[T_j] $\geq$ ts(T_i). By the rule for generating timestamps, ts(T_j) > max-lock-set[T_j] and therefore ts(T_j) > ts(T_i). Thus, timestamps assigned by the TM are consistent with the order of Commits, which is the condition we needed to show that Strict

2PL rw synchronization and TWR ww synchronization are correctly integrated.

BIBLIOGRAPHIC NOTES

Early TO based algorithms appear in [Shapiro, Millstein 77a], [Shapiro, Millstein 77b], and [Thomas 79]. The latter paper, first published in 1976 as a technical report, also introduced certification and TWR, and was the first to apply voting to replicated data (see Chapter 8). An elaborate TO algorithm using TWR, classes, and conflict graphs was built in the SDD-1 distributed DBS [Bernstein et al. 78], [Bernstein, Shipman 80], [Bernstein, Shipman, Rothnie 80], and [McLean 81]. Other TO algorithms include [Cheng, Belford 80] and [Kaneko et al. 79]. Multigranularity locking ideas are applied to TO in [Carey 83].

Serialization graph testing has been studied in [Badal 79], [Casanova 81], [Hadzilacos, Yannakakis 86], and [Schlageter 78]. [Casanova 81] contains the space-efficient SGT scheduler in Section 4.3.

The term "optimistic" scheduler was coined in [Kung, Robinson 81], who developed the concept of certifier independently of [Thomas 79]. Other work on certifiers includes [Haerder 84], [Kersten, Tebra 84], and [Robinson 82]. [Lai, Wilkinson 84] studies the atomicity of the certification activity. The performance of certifiers is analyzed in [Menasce, Nakanishi 82a], [Morris, Wong 84], [Morris, Wong 85], and [Robinson 82].

The rw and ww synchronization paradigm of Section 4.5 is from [Bernstein, Goodman 81] and [Bernstein, Goodman 81]. The 2PL and TWR mixed integrated scheduler is from [Bernstein, Goodman, Lai 83].

EXERCISES

4.1 In Basic TO, suppose the scheduler adjusts max-q-scheduled[x] when it sends $p_i[x]$ to the DM, instead of when it adds $p_i[x]$ to queue[x]. What effect does this have on the rate at which the scheduler rejects operations? What are the benefits of this modification to Basic TO?

4.2 Modify Basic TO to avoid cascading aborts. Modify it to enforce the weaker condition of recoverability. Explain why your modified schedulers satisfy the required conditions.

4.3 In Basic TO, under what conditions (if any) is it necessary to insert an operation $p_i[x]$ to queue[x] other than at the end?

4.4 Generalize the Basic TO scheduler to handle arbitrary operations (e.g., Increment and Decrement).

4.5 Modify the Basic TO scheduler of the previous problem to avoid cascading aborts. Does the compatibility matrix contain enough information to make this modification? If not, explain what additional informa-

tion is needed. Prove that the resulting scheduler produces histories that are cascadeless.

4.6 Compare the behavior of distributed 2PL with Wait-Die deadlock prevention to that of distributed Basic TO.

4.7 Prove that the Strict TO scheduler of Section 4.2 produces strict histories.

4.8 Design a conservative TO scheduler that uses knowledge of process speeds and message delays to avoid rejecting operations.

4.9 Prove that the ultimate conservative TO scheduler in Section 4.2 produces SR histories.

4.10 Modify the ultimate conservative TO scheduler in Section 4.2 so that each TM can manage more than one transaction concurrently.

4.11 Design an ultimate conservative TO scheduler that avoids rejections by exploiting predeclaration. (Do not use the TM architecture of Section 4.2, where each TM submits operations to the DM in timestamp order.) Prove that your scheduler produces SR executions.

4.12 Design a way of changing class definitions on-line in conservative TO.

4.13 Design a TO scheduler that guarantees the following property: For any history H produced by the scheduler, there is an equivalent serial history H_s such that if T_i is committed before T_j in H, then T_i precedes T_j in H_s. (T_i and T_j may not have operations that conflict with each other.) Prove that it has the property.

4.14 A *conflict graph* for a set of classes is an undirected graph whose nodes include R_I and W_I, for each class I, and whose edges include

- (R_I, W_I) for all I,
- (R_I, W_J) if the readset of class I intersects the writeset of class J, and
- (W_I, W_J) if the writeset of class I intersects the writeset of class J $(I \neq J)$.

Suppose each class is managed by one TM, and that each TM executes transactions serially. A transaction can only be executed by a TM if it is a member of the TM's class.

a. Suppose the conflict graph has no cycles. What additional constraints, if any, must be imposed by the scheduler to ensure SR executions? Prove that the scheduler produces SR executions.

b. Suppose the scheduler uses TWR for ww synchronization, and the conflict graph has no cycles containing an (R_I, W_J) edge (i.e., all cycles only contain (W_I, W_J) edges). What additional constraints, if any, must be imposed by the scheduler to ensure SR executions? Prove that the scheduler produces SR executions.

4.15 If the size of the timestamp table is too small, then too many recent timestamps will have to be deleted in order to make room for even more

recent ones. This will cause a TO scheduler to reject some older operations that access data items whose timestamps were deleted from the table. An interesting project is to study this effect quantitatively, either by simulation or by mathematical analysis.

4.16 Prove that the conservative SGT scheduler described in Section 4.3 produces SR executions.

4.17 Show that for any history produced by an SGT scheduler, there exists an assignment of timestamps to transactions such that the same history could be produced by a TO scheduler.

4.18 Design an SGT scheduler that guarantees the following property: For any history H produced by the scheduler, there is an equivalent serial history H_s such that if T_i is committed before T_j in H, then T_i precedes T_j in H_s. Prove that it has the property.

4.19 Modify the space-efficient SGT scheduler described in Section 4.3 to produce recoverable, cascadeless, and strict executions. Explain why each of your modified schedulers satisfies the required condition.

4.20 Give a serializability theoretic correctness proof of the space-efficient SGT scheduler described in Section 4.3.

4.21 Although the space requirements of both 2PL and space-efficient SGT are proportional to $a \cdot |D|$, where a is the number of active transactions and D is the size of the database, space-efficient SGT will usually require somewhat more space than 2PL. Explain why.

4.22 Since a certifier does not control the order in which Reads and Writes execute, a transaction may read arbitrarily inconsistent data. A correct certifier will eventually abort any transaction that reads inconsistent data, but this may not be enough to avoid bad results. In particular, a program may not check data that it reads from the database adequately enough to avoid malfunctioning in the event that it reads inconsistent data; for example, it may go into an infinite loop. Give a realistic example of a transaction that malfunctions in this way using 2PL certification, but never malfunctions using Basic 2PL.

4.23 Prove that the committed projection of every history produced by a 2PL certifier could have been produced by a 2PL scheduler.

4.24 Give an example of a complete history that could be produced by a 2PL certifier but not by a 2PL scheduler. (In view of the previous exercise, the history must include aborted transactions.)

4.25 Modify the 2PL certifier so that it avoids cascading aborts. What additional modifications are needed to ensure strictness? Explain why each modified certifier satisfies the required condition.

4.26 If a 2PL certifier is permitted to certify two (or more) transactions concurrently, is there a possibility that it will produce a non-SR execution? Suppose the 2PL certifier enforces recoverability. If it is permitted to certify

two transactions concurrently, is there a possibility that it will reject more transactions than it would if it certified transactions one at a time?

4.27 A transaction is called *rw phased* if it waits until all of its Reads have been processed before it submits any of its Writes. Its *read* (or *write*) phase consists of the time the scheduler receives its first Read (or Write) through the time the scheduler acknowledges the processing of its last Read (or Write). Consider the following certifier for rw phased transactions in a centralized DBS. The scheduler assigns a timestamp to a transaction when it receives the transaction's first Write. Timestamps increase monotonically, so that $ts(T_i) < ts(T_j)$ iff T_i begins its write phase before T_j begins its write phase. To certify transaction T_j, the certifier checks that for all T_i with $ts(T_i) < ts(T_j)$, either (1) T_i completed its write phase before T_j started its read phase, or (2) T_i completed its write phase before T_j started its write phase and w-scheduled(T_i) $\cap$ r-scheduled(T_j) is empty. If neither condition is satisfied for some such T_i, then T_j is aborted; otherwise, it is certified.

a. Prove that this certifier only produces serializable executions.

b. Are there histories produced by this certifier that could not be produced by the 2PL certifier?

c. Are there histories involving only rw phased transactions that could be produced by the 2PL certifier but not by this certifier?

d. Design data structures and an algorithm whereby a certifier can efficiently check (1) and (2).

4.28 In the previous exercise, suppose that for each T_i with $ts(T_i) < ts(T_j)$, the certifier checks that (1) or (2) or the following condition (3) holds: T_i completed its read phase before T_j completed its read phase and w-scheduled(T_i) $\cap$ (r-scheduled(T_j) $\cup$ w-scheduled(T_j)) is empty. Answer (a) - (d) in the previous exercise for this new type of certifier.

4.29 Suppose the scheduler uses a *workspace model* of transaction execution, as in the mixed integrated scheduler of Section 4.5. That is, it delays processing the Writes of each transaction T_i until after it receives T_i's Commit. Suppose a scheduler uses the 2PL certification rule, but uses the workspace model for transaction execution. Does the scheduler still produce SR executions? If so, prove it. If not, modify the certification rule so that it does produce SR executions. Are the executions recoverable? Cascadeless? Strict?

4.30 Does the SGT certifier still produce SR executions using a workspace model of transaction execution (see Exercise 4.29)? If so, prove it. If not, modify the certification rule so that it does produce SR executions.

4.31 In the workspace model of transaction execution (see Exercise 4.29), the scheduler buffers a transaction T_i's Writes in w-queue[T_i]. Since the scheduler must scan w-queue[T_i] for every Read issued by T_i, the efficiency of that scan is quite important. Design a data structure and search

algorithm for w-queue[T_i] and analyze its efficiency.

4.32 Describe an SGT certifier based on the space-efficient SGT scheduler of Section 4.3, and prove that it produces SR executions.

4.33 Modify the pure integrated scheduler of Section 4.5 (Basic TO for rw synchronization and TWR for ww synchronization) to produce cascadeless executions. Describe the algorithm using the actual data structures maintained by the scheduler, such as max-r-scheduled, max-w-scheduled, and w-in-transit.

4.34 Describe a pure integrated scheduler that uses ultimate conservative TO for rw synchronization and TWR for ww synchronization. Prove that it produces SR executions.

4.35 Describe a mixed integrated scheduler that uses conservative TO for rw synchronization and 2PL for ww synchronization. Prove that it produces SR executions.