Chapter 4

Concurrency Control Algorithms

"The optimist believes we live in the best of all possible worlds. The pessimist fears this is true." (Robert Oppenheimer)

4.1 Goal and Overview of the Chapter

scheduling algorithms (schedulers)

In this chapter we will present a number of algorithms — called *scheduling algorithms* or *schedulers* for short — a data server could use to produce serializable schedules for multiple concurrent transactions. Following our discussion from the previous chapter, we will concentrate on schedulers that produce *conflict*-serializable schedules. We will also follow the previous chapter in that we do not yet consider transaction aborts or system failures; as before, we assume that system components other than the scheduler are capable of handling them properly. Later, in Part III of the book, we will expand the notion of schedules to make the implementation of aborts explicit (this will lead to notions such as *prefix reducibility* in Chapter 11).

We will place some emphasis in this chapter on the question of how to design scheduling protocols in general, and on how to verify a given protocol and prove it correct. With respect to the latter, we will be interested in two criteria: First, each protocol that we present has to be *safe* in the sense that all histories it outputs have to be members of CSR, the class of conflict serializable schedules. Second, we are interested in the *scheduling power* of a protocol, or in its ability to produce some class of serializable histories (e.g., class CSR) *in its entirety or partially only.* For example, if scheduler S_1 can produce *every* history in CSR, and scheduler S_2 can produce a strict subset of CSR only, then S_1 intuitively has more options than S_2 to

choose a CSR schedule for a given set of transactions, and hence has more scheduling power. In other words, the scheduling power of a scheduler is a kind of measure for the amount of "parallelism" it can utilize.

locking schedulers

After some general considerations on the design of schedulers in Section 4.2, we will first look, in Section 4.3, at the most prominent type of schedulers, which are schedulers based on *locking.* As will be seen, this amounts to a rich class of scheduling protocols that essentially exploits a variety of options in the use of locks. Locking protocols are also the most important ones in practice, since they mostly (but not always) outperform all other scheduling protocols, are easy to implement, incur little run-time overhead, and can be generalized to various transaction-related settings. In particular, we will see that locking protocols can be successfully employed for both major computational models of this book, the page model considered here and the object model considered later. A second major class of protocols, discussed in Section 4.4, does not use locking at all, but replaces this mechanism by other constructs, such as *timestamps* in the timestamp-ordering protocol or *validation* in so-called *optimistic protocols.* Finally, Section 4.5 briefly presents *hybrid protocols* that combine elements from both locking and non-locking methods.

non-locking schedulers

hybrid schedulers

4.2 General Scheduler Design

transaction manager (TM)

Before we will look at actual protocols, we briefly provide a conceptual view of the "system environment" in which transaction processing takes place in a data server. To this end, we recall Figure 1.4 from Chapter 1, which gave some kind of overview of the layered components involved in a data server such as a database system. We now consider an extension and, at the same time, a convenient abstraction of such a layered architecture by putting the scheduler component in its appropriate place. Conceptually, we insert a *transaction manager (TM)* as an additional component between two layers, typically (but not necessarily) between the query execution layer and the access layer or between the access layer and the storage layer. This TM intercepts all calls from one layer to the next lower layer, takes the necessary steps for concurrency control (and also recovery), and then passes the calls further on to the lower layer to execute the invoked operation. With this abstract architecture, the further processing of the operations by the lower layer and the other layers underneath is not really relevant for the TM, nor is it observable by the TM anyway. Therefore, we can conceptually collapse all layers beneath the TM into a single "virtual" system component that we refer to as the *data manager (DM).* This architectural model and the dynamic behavior of a scheduler are illustrated in Figure 4.1. The scheduler dynamically interleaves data operations that result from the requests of multiple clients.

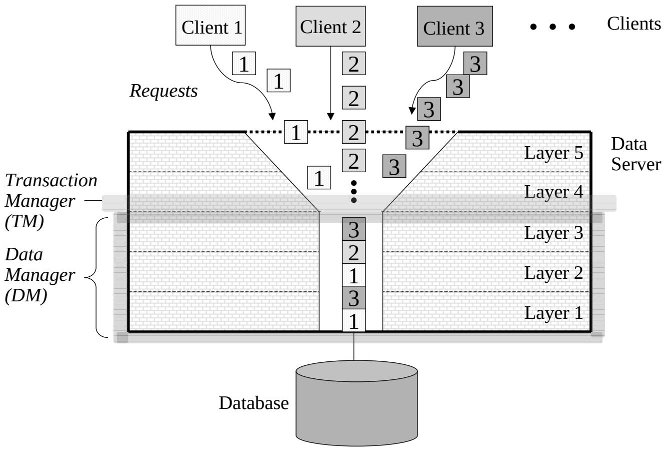

Figure 4.1: Illustration of a transaction scheduler.

The TM's main task is bookkeeping and in particular managing the lists *trans, commit, abort*, and *active* we introduced for our model of schedules and histories. In addition it keeps a list of ready-to-execute steps for all the active transactions. From this list it selects individual steps and sends them to the scheduler. In that way, the scheduler receives an arbitrary *input schedule* from the TM, and its task is to transform that into a serializable *output schedule*.

Since we have assumed in Chapter 3 that termination steps of transactions only become relevant in the context of a schedule or a history, we can assume now that the TM can recognize the beginning and the end of a transaction. Thus a transaction is delimited by its "begin transaction" and "commit transaction" (or, infrequently, "rollback transaction") calls. We furthermore assume that the TM:

- outputs a step c_i to the scheduler when it sees the "commit transaction" request of t_i and no failures have occurred (from the TM's point of view) during the execution of t_i (normal termination),
- outputs a step a_i as soon as it recognizes an error situation for t_i (including the case that the transaction program explicitly requests a rollback).

Recall from Chapter 1 that this is in line with the way commercial data servers such as database systems operate when accessed through an application program; there are explicit commands such as "commit" and "rollback" for controlling transaction termination.

In addition, a scheduler can autonomously decide to abort a transaction, e.g., if it discovers a situation that is no longer serializable. We remark at this point that a transaction can be in various *states* during its lifetime, as shown in Figure 4.2:

Upon its "begin transaction" call a transaction becomes *active*. An active transaction can execute a data operation or await the execution of the next such operation. In the former case it is in state *running*, in the latter it is *blocked*. The end of an active transaction can be either *aborted* or *committed*, where the latter state is the normal outcome.

Now the *scheduler* receives steps of the form r, w, a, and c in an input schedule and has to transform that into a serializable output schedule. To this end, it can perform one of the following actions:

1. *output*: The step is output right away, i.e., attached to the end of the present output schedule (which is initially empty) to form a new output schedule; in this case, the step can be an r, w, a, or c action.

scheduler actions

2. *reject*: The step is not output (for example, since it is already clear that its execution would destroy the serializability of the output). In this case, it can

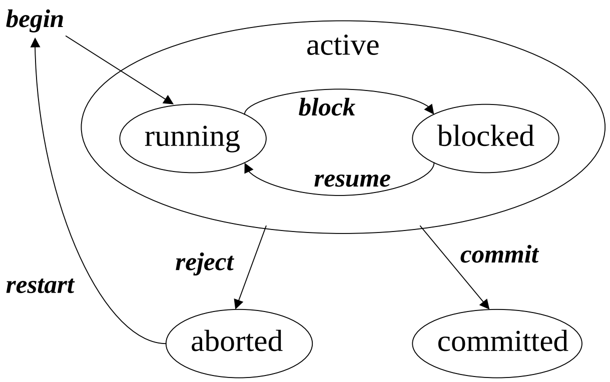

Figure 4.2: States of a transaction.

only be an r or a w; the transaction in question is then aborted, i.e., terminated by an a.

3. *block*: The step is neither output nor rejected, but considered as "currently not executable" and thus postponed. The execution of this step will then be retried later; again, the step in question can only be an r or a w (under special circumstances, the step could also be a c, for example if the scheduler is desired for COCSR).

The data manager (DM) executes the steps of a (serializable) output schedule in the order that has been determined by the scheduler; in other words,

- for an r it reads a data item,
- for a w it writes a data item,
- for a c it initiates steps to make the results of the transaction persistent,
- for an a it initiates steps to undo the transaction.

In this chapter, our main interest is in the *scheduler*, or in protocols on which such a device can be based. Regarding the design of a scheduling protocol, we already know that its output should at least fall into CSR. Once such a design goal has been fixed, a scheduler can conceptually be seen as a "filter" which lets exactly the members of that class pass unmodified. Thus, if the input schedule is from that class, it coincides with the output schedule; the serializable input schedules then form the scheduler's *fixpoint set*. Now a "generic" scheduler can be described roughly as follows:

```
scheduler ( ):
var newstep:    step;
{ state := initial_state;
   repeat
     on arrival(newstep) do
        { update(state);
           if test(state, newstep)
              then output(newstep)
              else block(newstep) or reject(newstep) }
   forever };
```

generic scheduler

For a concrete scheduling algorithm we will now have to make precise:

1. what information characterizes a "*state*" and how the scheduler updates that information depending on a current step,

2. what makes up the test that decides upon the scheduler's next action (execute, block, or reject).

scheduler classification

Depending on how the test result looks most of the time, schedulers can be classified as "optimistic" or "pessimistic": A scheduler is

optimistic schedulers

pessimistic schedulers

- *optimistic*, sometimes also referred to as "aggressive", if it mostly lets steps pass and rarely blocks; clearly, this bears the dangers of "getting stuck" eventually when the serializability of the output can no longer be guaranteed;
- *pessimistic*, sometimes also referred to as "conservative", if it mostly blocks (upon recognizing conflicts); in the extreme, albeit unlikely case, the output could become a serial schedule or history, namely if all transactions but one were blocked.

Locking schedulers, which we will consider in the next section, belong to the class of pessimistic schedulers. *Non-locking* schedulers, on the other hand, to be considered in later sections, can be either optimistic or pessimistic.

CSR-safety

Regarding the safety as well as the scheduling power of a protocol, we will use the following notions and notations:

Definition 4.1 (CSR-safety)

A scheduler is called *CSR-safe* if every output it produces is (conflict-) serializable. If S denotes a scheduler, then $Gen(S)$ denotes the set of all schedules that S can generate as an output. Thus, a scheduler S is CSR-safe if $Gen(S) \subseteq$ CSR.

4.3 Locking Schedulers

We now embark on a discussion of the first major class of scheduling protocols, the so-called *locking schedulers*. We do this here from a purely conceptual point of view, in order to clarify the methodology as well as the options and implications behind it. For the time being, we entirely neglect implementation techniques such as *lock tables*, where each lock is represented as one entry with hash-based access on data-item and transaction identifier, or *multiple granularity*, where transactions can choose an appropriate level of granularity depending on how many data items they intend to access and the contention for these items. Here we consider a uniform level of granularity like pages, which suffices for the discussion of this chapter. We will discuss such implementation issues in Chapter 10.

4.3.1 Introduction

In a nutshell, the idea of a locking scheduler is to synchronize access to shared data by using *locks*, which can be set on and removed from data items on behalf of transactions. The intuitive meaning is that if a transaction holds a lock on a data item, the item is not available to other, concurrent transactions (i.e., those transactions are "locked out"). When a transaction *requests* a lock (or, actually, the scheduler requests the lock on behalf of a transaction), the scheduler checks whether the lock is already held by another transaction (in a conflicting mode, as we will explain shortly). If so, the lock request is considered to be a *lock conflict* and the requesting transaction is suspended, i.e., suffers a *lock wait* and becomes *blocked.* Otherwise, the requested lock can be granted, and we say that the transaction acquires the lock. Eventually, each transaction *releases* its locks that it holds at appropriate points in time. At this time the scheduler checks whether any waiting transaction can be resumed; if so, the scheduler will now grant the transaction's lock request that has formerly led to the lock wait and the transaction can go ahead accessing the data item of interest.

lock request
lock conflict
lock wait
lock release

lock modes:
(shared) read lock
(exclusive) write lock

Since in our model transactions can read or write data items, we will associate *two* types of locks, or *lock modes*, with every data item x: a *read lock* $rl(x)$, also known as *shared lock*, and a *write lock* $wl(x)$, also known as *exclusive lock.* As we do for other steps in a schedule, we use subscripts to distinguish the transactions that issue lock operations. We thus use rl and wl both to denote the lock status and mode of a data item and the operation that sets the lock. Even further, we neglect the distinction between requesting a lock on a data item and acquiring a (previously requested) lock; all of this should be subsumed by rl and wl.

The following is now near at hand: Two locks $pl_i(x)$ and $ql_j(y)$ are in *conflict* if $x = y$, $i \neq j$, and the data operations p and q are in conflict (i.e., $p = r \wedge q = w$ or $p = w \wedge q = r$ or $p = w \wedge q = w$). The immediate consequence of this is that our notion of conflict carries over to locks; we will encounter similar regulations in other transaction models later. It is common to describe the *compatibility* of locks (or lock modes) in tabular form:

compatibility of locks

		lock requested	
		$rl_i(x)$	$wl_i(x)$
lock	$rl_j(x)$	+	–
held	$wl_j(x)$	–	–

The table is to be read as follows: If a transaction t_i has already set a lock $pl_i(x)$ and a transaction t_j, $j \neq i$, requests a lock $ql_j(x)$, this latter lock can be granted (it is *compatible* or *not in conflict* with the former) if the corresponding table entry is a "+"; otherwise, it cannot be granted.

For unlocking data items, i.e., for releasing locks, we will use similar notations: $ru(x)$ will stand for "read unlock," $wu(x)$ for "write unlock" w.r.t. a data item x.

When using locks, a scheduler has to manage their usage by controlling in what order transactions set and release locks. In light of the generic scheduling algorithm shown earlier this proceeds as follows: Let a step $o_i(x)$ arrive as next input ($o \in \{r, w\}$). If $ol_i(x)$ is not set, the scheduler has to test whether some $pl_j(x)$, $j \neq i$, $p \in \{r, w\}$, has been set:

1. If this is not the case, $ol_i(x)$ will be set;
2. if it is the case, the scheduler has to test whether $ol_i(x)$ is in conflict with locks on x already set on behalf of other transactions:
   (2.1) if there is no conflict, then $ol_i(x)$ is set (actually, a lock on x is not set again, but the scheduler records in a *lock table* that now t_i also holds a lock on x);
   (2.2) if there exists a conflicting lock, transaction t_i is blocked.

Formally, a locking scheduler augments every transaction it processes by new actions of type rl, wl, ru, and wu, resulting in a "locked transaction". It now makes sense to restrict the use of locks by some near-at-hand rules; specifically, we will from now on require that for each transaction t_i which appears *completely* in a schedule s produced by the locking scheduler in question, the following *locking well-formedness rules* hold:

rules for well-formed locking

LR1: If t_i contains a step of the form $r_i(x)$ $[w_i(x)]$ then schedule s also contains a step of the form $rl_i(x)$ $[wl_i(x)]$ *before* the data operation, i.e., a lock on x is held at the point of the operation on x. Moreover, s contains a step of the form $ru_i(x)$ $[wu_i(x)]$ somewhere *after* the operation.

LR2: For each x accessed by t_i, schedule s has at most one $rl_i(x)$ and at most one $wl_i(x)$ step; in other words, locks of the same type are set at most once per transaction and per data item.

LR3: No step of the form $ru_i(.)$ or $wu_i(.)$ is redundant, i.e., executed per transaction more than once.

These rules apply to schedules, i.e., incomplete histories, in an analogous fashion. For example, for an arbitrary schedule s rule LR1 reads as follows:

If t_i contains a step of the form $r_i(x)$ $[w_i(x)]$ then s also contains a step of the form $rl_i(x)$ $[wl_i(x)]$ before the data operation. Moreover, s will eventually be augmented to also contain a step of the form $ru_i(x)$ $[wu_i(x)]$ somewhere after the operation.

In slight abuse of the terminology just introduced, we will also say that the rules above apply to a *transaction*, which then means that they apply to the transaction in the context of a schedule in which it occurs.

Example 4.2

Consider transactions $t_1 = r_1(x)w_1(y)$ and $t_2 = w_2(x)w_2(y)$ as well as the following history:

$s_1 = rl_1(x)r_1(x)ru_1(x)wl_2(x)w_2(x)wl_2(y)w_2(y)wu_2(x)wu_2(y)c_2$
$wl_1(y)w_1(y)wu_1(y)c_1$

Clearly, s_1 satisfies rules LR1–LR3; the same applies to the following history:

$s_2 = rl_1(x)r_1(x)wl_1(y)w_1(y)ru_1(x)wu_1(y)c_1wl_2(x)w_2(x)wl_2(y)$
$w_2(y)wu_2(x)wu_2(y)c_2$.

Notice that s_2 is even serial. □

The schedules (and eventually the histories) produced by a locking scheduler contain data operations, termination operations, and lock and unlock steps. Since it will sometimes be useful to distinguish schedules without lock and unlock operations from schedules in which these occur, we introduce the following notation: For some schedule s, let $\mathrm{DT}(s)$ denote the projection of s onto the steps of type r, w, a, c, i.e., onto Data and Termination operations.

Example 4.3

Let s_1 and s_2 be as in the previous example; then we have:

$\mathrm{DT}(s_1) = r_1(x)w_2(x)w_2(y)c_2w_1(y)c_1$
$\mathrm{DT}(s_2) = r_1(x)w_1(y)c_1w_2(x)w_2(y)c_2 \quad (= t_1t_2)$ □

Note that history s_1 of the above example cannot be an acceptable execution, since $\mathrm{DT}(s_1) \notin \mathrm{CSR}$. The reason is that in between $ru_1(x)$ and $wl_1(y)$ both data items are written by t_2, so that a conflict equivalent serial history would have to satisfy "$t_1 < t_2$" w.r.t. x, but "$t_2 < t_1$" w.r.t. y, which contradict each other. Clearly, a CSR-safe scheduler has to avoid such situations.

Whenever there is no ambiguity, we will use the notation s rather than the above notation $\mathrm{DT}(s)$ in order to refer to a schedule or history that is free of lock and unlock operations; however, the reader is urged to keep in mind that there is a formal difference between the output produced by a locking protocol and that of the output needed to determine correctness.

We mention that our locking rules LR1–LR3 apply to *every* transaction in a schedule produced by a locking scheduler; in particular, they apply not only to active, but also to aborted transactions, where we assume for aborted transactions that the execution of the abort operation also takes care of releasing all locks set on behalf of that transaction.

Transaction t_i ***holds a lock*** on data item x at all points in between t_i's lock acquisition (i.e., the $rl_i(x)$ or $wl_i(x)$ operation) and the corresponding lock release (i.e., the $ru_i(x)$ or $wu_i(x)$ operation). In general, we say that a scheduler operates according to a *locking protocol*, if the following holds for each output s:

LR1–LR3: s satisfies the locking rules LR1–LR3,

LR4: if x is held locked by both t_i and t_j for $t_i, t_j \in \text{trans}(s)$, $i \neq j$, then these locks are not in conflict (i.e., they are *compatible*).

4.3.2 The Two-Phase-Locking Protocol

We now present the first concrete scheduler based on locking, the *two-phase locking protocol (2PL)*; this protocol is the one used most in commercial database systems.

Within the family of locking protocols, two-phase locking is characterized by the following rule regarding the timepoints of lock releases:

two-phase locking (2PL)

Definition 4.4 (Two-phase locking (2PL))

A locking protocol is *two-phase* if for every output s and every transaction $t_i \in \text{trans}(s)$ it is true that no ql_i step follows the first ou_i step, $o, q \in \{r, w\}$. A two-phase locking protocol is abbreviated *2PL* or *2PL scheduler.*

In words, a locking protocol is two-phase if for every transaction a phase during which locks are set is distinguished from and strictly followed by a phase during which locks are released. (Later we will see deviations from this two-phase rule in which the two phases of a transaction are allowed to overlap; for the time being, this is not the case.)

Example 4.5

Consider history s_1 from the previous example again: It obviously violates the 2-phase rule since $ru_1(x) <_s wl_1(y)$. A possible correct execution order is history s_2 from the same example. □

The next example, besides showing how a 2PL scheduler might process a given input, also shows how a transaction that first reads a data item and later writes it may start out by getting a read lock, and later upgrade this read lock to a write lock. Such an upgrade is also called *lock conversion.*

lock c
(lock

Example 4.6

Consider the following input schedule:

$$s = w_1(x)r_2(x)w_1(y)w_1(z)r_3(z)c_1w_2(y)w_3(y)c_2w_3(z)c_3$$

A 2PL scheduler could transform s into the following output history (which, of course, is not uniquely determined):

$$wl_1(x)w_1(x)wl_1(y)w_1(y)wl_1(z)w_1(z)wu_1(x)rl_2(x)r_2(x)wu_1(y)wu_1(z)c_1rl_3(z)$$
$$r_3(z)wl_2(y)w_2(y)wu_2(y)ru_2(x)c_2wl_3(y)w_3(y)wl_3(z)w_3(z)wu_3(z)wu_3(y)c_3$$

Here t_3 upgrades its read lock on z to a write lock, so that only that latter needs to be released at the end of t_3. Also note that locks need not be released in the same order in which they were set. The resulting execution is also illustrated in Figure 4.3, where time proceeds from left to right and dashed lines correspond to lock waits. □

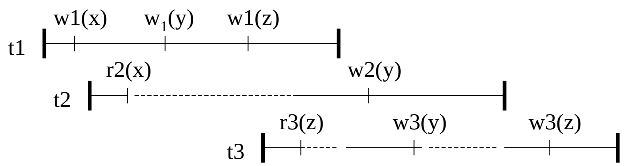

Figure 4.3: Example execution under the 2PL protocol.

We now discuss the (CSR-)safety of the 2PL protocol. To this end, we need to verify that $\text{DT}(s) \in \text{CSR}$ holds for each output s created by a 2PL scheduler. The argument, which will deliver a positive result, goes along the following lines:

Lemma 4.7

Let s be the output of a 2PL scheduler. Then for each transaction $t_i \in \text{commit}(\text{DT}(s))$ the following holds:

(i) If $o_i(x)$ ($o \in \{r, w\}$) occurs in CP(DT(s)), then so do $ol_i(x)$ and $ou_i(x)$ with the sequencing $ol_i(x) < o_i(x) < ou_i(x)$. (Here CP(.) again denotes the committed projection of a schedule introduced in Chapter 3.)

(ii) If $t_j \in$ commit(DT(s)), $i \neq j$, is another transaction s.t. some steps $p_i(x)$ and $q_j(x)$ from CP(DT(s)) are in conflict, then either $pu_i(x) < ql_j(x)$ or $qu_j(x) < pl_i(x)$ holds (if two steps are in conflict, then so are their lock operations; locks in conflict are not set simultaneously).

(iii) If $p_i(x)$ and $q_i(y)$ are in CP(DT(s)), then $pl_i(x) < qu_i(y)$, i.e., every lock operation occurs before every unlock operation of the *same* transaction.

Lemma 4.8

Let s be the output of a 2PL scheduler, and let $G := G(\mathrm{CP}(\mathrm{DT}(s)))$ be the conflict graph of CP(DT(s)). Then the following holds:

(i) If (t_i, t_j) is an edge in G, then $pu_i(x) < ql_j(x)$ for some data item x and two operations $p_i(x), q_j(x)$ in conflict.

(ii) If $(t_1, t_2, \ldots, t_n)$ is a path in $G, n \geq 1$, then $pu_1(x) < ql_n(y)$ for two data items x and y as well as operations $p_1(x)$ und $q_n(y)$.

(iii) G is acyclic.

Proof. (i) If (t_i, t_j) is an edge in G, then CP(DT(s)) comprises two steps $p_i(x)$ and $q_j(x)$ in conflict s.t. $p_i(x) < q_j(x)$. According to (i) in Lemma 4.7, this implies $pl_i(x) < p_i(x) < pu_i(x)$ and $ql_j(x) < q_j(x) < qu_j(x)$. According to (ii) in Lemma 4.7, we moreover find (a) $pu_i(x) < ql_j(x)$ or (b) $qu_j(x) < pl_i(x)$. Case (b) means $ql_j(x) < q_j(x) < qu_j(x) < pl_i(x) < p_i(x) < pu_i(x)$ and hence $q_j(x) < p_i(x)$, a contradiction to $p_i(x) < q_j(x)$. Thus, $pu_i(x) < ql_j(x)$ (which is case (a)), which had to be shown.

(ii) The proof goes by induction on n: The induction base $n = 2$ follows directly from part (i): If (t_1, t_2) is an edge in G, there is a conflict between t_1 and t_2. Thus, $pu_1(x) < ql_2(x)$, i.e., t_1 unlocks x before t_2 locks x. In other words, when t_2 sets a lock, t_1 has already released one.

Now assume our claim holds for n transactions on a path through G, and consider a path of length $n + 1$. The inductive assumption now tells us that there are data items x and z s.t. $pu_1(x) < ol_n(z)$ in s. Since (t_n, t_{n+1}) is an edge in G, it follows from (i) above that for operations $v_n(y)$ and $q_{n+1}(y)$ in conflict we have $vu_n(y) < ql_{n+1}(y)$. According to (iii) of Lemma 4.7, this implies $ol_n(z) < vu_n(y)$ and hence $pu_1(x) < ql_{n+1}(y)$.

(iii) Assume that G is cyclic. Then there exists a cycle, say, of the form $(t_1, t_2, \ldots, t_n, t_1)$, $n \geq 1$. By (ii), $pu_1(x) < ql_1(y)$ for operations $p_1(x), q_1(y)$, a contradiction to the 2-phase rule (or to (iii) of Lemma 4.7). □

Since the conflict graph of an output produced by a 2PL scheduler is acyclic, and since by the serializability theorem (Theorem 3.41) we have proved in the previous chapter acyclicity of a conflict graph characterizes membership of the respective history in class CSR, we have thus shown:

Theorem 4.9

A 2-phase locking scheduler is CSR-safe, i.e., $Gen(\mathrm{2PL}) \subseteq \mathrm{CSR}$.

Formally, the $Gen(S)$ of a locking scheduler S contains DT(s) only for every output schedule s.

We note that the inclusion just proved is strict, i.e., 2PL is unable to generate *all* of CSR:

Example 4.10

Let $s = w_1(x)r_2(x)c_2r_3(y)c_3w_1(y)c_1$. As in a previous example we have $s \in$ CSR, since $s \approx_c t_3t_1t_2$. On the other hand, it is easily verified that s cannot be produced by a 2PL scheduler. The simple reason is that 2PL would start out by setting a write lock for t_1 on x; that would immediately block t_2. The next lock operation possible would then be to set a read lock for t_3 on y; once t_3 finishes, t_1 can set a lock on y and finish. Only thereafter t_2 would run, so if s was the input to 2PL, the following would be output: $s' = w_1(x)r_3(y)c_3w_1(y)c_1r_2(x)c_2$.

A fully formal proof of the infeasibility of s under 2PL proceeds as follows. First, by Lemma 4.8 (ii), we know that $wu_1(x) < rl_2(x)$ and $ru_3(y) < wl_1(y)$. Second, the well-formedness rules for locked schedules require $rl_2(x) < r_2(x)$ and $r_3(y) < ru_3(y)$. Third, the schedule itself includes the order $r_2(x) < r_3(y)$. Putting all these ordering constraints together and using transitivity, we infer that $wu_1(x) < wl_1(y)$, which contradicts the fundamental rule of the 2PL protocol. Therefore, the schedule is not feasible under 2PL, regardless of how lock and unlock operations may be inserted. □

The example shows that the fact that a history has been created by 2PL is sufficient for membership in CSR, but not necessary. The latter can even be refined; to this end, recall the definition of OCSR from Definition 3.45: A history s is *order-preserving conflict serializable*, if it is conflict serializable, i.e., there exists a serial history s' s.t. $\mathrm{op}(s) = \mathrm{op}(s')$ and $s \approx_c s'$, and if the following holds for s' and all

OCSR

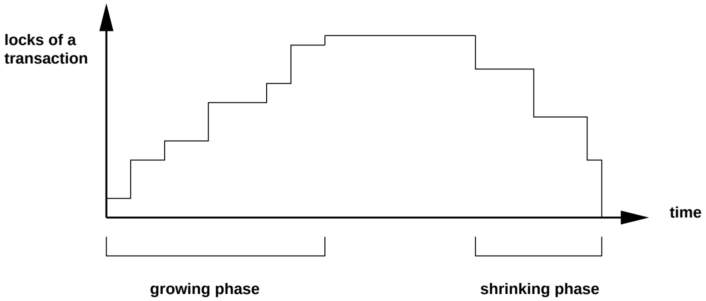

Figure 4.4: Growing and shrinking phase of a transaction under 2PL.

$t, t' \in \text{trans}(s)$: If t occurs completely before t' in s, then the same holds in s'. OCSR denotes the class of all order-preserving conflict serializable histories. Now we have:

Theorem 4.11

$Gen(\text{2PL}) \subset \text{OCSR}$

To see that the inclusion is proper, consider the schedule

$$w_1(x)r_2(x)r_3(y)r_2(z)w_1(y)c_3c_1c_2.$$

This schedule, which is a slight variation of the sample schedule for showing that 2PL cannot generate all of CSR, is CSR and trivially falls into OCSR (as there is no pair of strictly sequential transactions), but is still not feasible under 2PL. In Section 4.3.5, we will discuss an alternative to 2PL which is capable of generating *every* schedule in OCSR.

In the output of a 2PL scheduler, for each transaction a *growing phase* (in which locks are set only) can be distinguished from a *shrinking phase* (in which locks are released only). This is illustrated in Figure 4.4. We will exploit this observation later for designing variants of 2PL.

4.3.3 Deadlock Handling

As locking protocols require transactions to wait when requested locks cannot be granted immediately, a set of transactions, each holding some locks and requesting

an additional one, may end up being mutually blocked. For example, in the simplest case with only two transactions t_1 and t_2, t_1 may have requested a lock that t_2 is holding in a conflicting mode so that t_1 is waiting for t_2 to release this lock, while at the same time t_2 has requested a lock held by t_1 in a conflicting mode. Such cyclic wait situations are commonly known as *deadlocks*.

deadlocks

For an implementation of the 2PL protocol it is important to realize that it is not deadlock-free. To see this, consider the above scenario, refined into the following input schedule:

$$r_1(x)w_2(y)w_2(x)c_2w_1(y)c_1$$

Let this schedule be submitted to a 2PL scheduler. The scheduler processes the input from left to right and hence creates

$$rl_1(x)r_1(x)wl_2(y)w_2(y)$$

At this point, it has to stop, since the next action would be $wl_2(x)$ which is incompatible with $rl_1(x)$, so that $w_2(x)$ has to be blocked. The only step left would then be $w_1(y)$, requiring lock operation $wl_1(y)$, which is incompatible with $wl_2(y)$. Notice that the scheduler cannot even execute $ru_1(x)$, since thereafter t_1 could no longer acquire new locks due to the 2-phase rule. Hence the scheduler has encountered a deadlock. This situation is pictorially illustrated in Figure 4.5 with time proceeding from left to right and lock waits indicated by dashed lines.

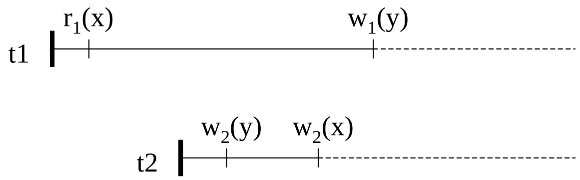

Figure 4.5: Illustration of a deadlock.

There is another situation that can cause deadlocks, namely the already mentioned *lock conversion*: Suppose a transaction t first reads and later writes a data item, and initially sets $rl(x)$; then it needs to upgrade this to $wl(x)$ at some later point. If two transactions do this at the same time, the following can happen: Let $t_1 = r_1(x)w_1(x)$ and $t_2 = r_2(x)w_2(x)c_2$, and let the scheduler execute

lock conversion

$$rl_1(x)r_1(x)rl_2(x)r_2(x)$$

Again, this prefix cannot be extended following the 2-phase rule, since an upgrade of $rl_1(x)$ to $wl_1(x)$ would be in conflict with $rl_2(x)$, and analogously $wl_2(x)$ with $rl_1(x)$.

If a protocol allows deadlocks, such as 2PL does, it needs appropriate measures to detect and resolve them. There are various categories of approaches to handle deadlocks, which are discussed next. The first category, designed to allow deadlocks as in situations like the ones just described, comprises approaches for *deadlock detection*. Strategies for deadlock detection are generally based on the notion of a *waits-for graph* (WFG), which is a graph $G = (V, E)$ whose nodes are the active transactions, and in which an edge of the form (t_i, t_j) indicates that t_i waits for t_j to release a lock that it needs. Clearly, a *cycle* in WFG exhibits a deadlock.

deadlock detection

For deadlock detection, a WFG needs to be built and maintained explicitly. In this approach the question arises how often should a WFG be tested for cycles. There are essentially two options for this issue:

- Under *continuous detection*, the WFG is always kept cycle-free by checking for cycles (and eventually breaking them) whenever a lock request is not granted immediately, i.e., every time a transaction blocks;
- under *periodic detection* the WFG is tested only periodically for cycles, e.g., once per second. Clearly, selecting an appropriate time interval for periodic detection is not an easy task, and may even impact performance to some extent.

When the scheduler detects a cycle (or several cycles), it has to resolve the corresponding deadlock(s) by aborting at least one transaction from every cycle. Again, there are several *victim selection* criteria, including the following:

1. *Last blocked*: Pick the transaction that blocked most recently, i.e., the one that just blocked in the case of continuous detection.
2. *Random*: from those involved in a deadlock cycle, pick one transaction randomly.
3. *Youngest*: pick the transaction with the most recent startup time, i.e., the one that began running most recently.

deadlock resolution

4. *Minimum locks*: pick the transaction that holds the fewest locks.
5. *Minimum work*: pick the transaction that has consumed the least amount of resources (e.g., CPU time) so far.
6. *Most cycles*: pick the transaction that breaks the largest number of cycles simultaneously.
7. *Most edges*: pick the transaction which eliminates as many edges as possible.

The idea of the youngest, minimum-locks, and minimum-work strategies is aim at minimizing the amount of wasted work, for the victim's work so far is lost, the cost of the rollback itself is proportional to that work, and in most cases the transaction will be restarted some time after the rollback is completed. The three strategies are heuristics that approximate this goal to a certain extent, with the subsidiary goal of keeping the overhead extremely low. A problem with all victim selection strategies is that they are susceptible to so-called "livelocks", situations where a transaction (or strictly speaking different incarnations of the same transaction) is repeatedly chosen as deadlock victim, rolled back and restarted, and thus appears to make progress from a microscopic perspective, but does not really come any closer to its commit from a global viewpoint. In essence, such a transaction would "starve", hence the other name "starvation" for this phenomenon. Heuristics for fixing this problem so that starvation-freedom can be guaranteed are the subject of Exercise 3.

starvation (livelock)

For the most-cycles strategy, consider the example shown in Figure 4.6, where an insertion of the dashed edge in either graph closes several cycles simultaneously. For the most-edges option, Figure 4.7 shows a WFG with a cycle and candidate victims t_1 and t_2. Depending on which of the two transactions is aborted, either 2 or 4 edges remain in the graph, and clearly the former is more desirable.

The second category of approaches to handle deadlocks is *deadlock prevention*, in which a WFG is not explicitly maintained. Since prevention amounts to never allowing blocked states that can lead to circular waiting, it is clear that conservative protocols are now needed. Possible restrictions include the following:

deadlock prevention

1. *Wait-die*: If a lock request from transaction t_i leads to a conflict with t_j, resolve as follows: If t_i started before t_j, it is blocked (t_i "waits for" t_j); otherwise, it is restarted (t_i "dies"). In other words, a transaction can only be blocked by a younger transaction.
2. *Wound-wait*: If a lock request from transaction t_i leads to a conflict with another transaction t_j, resolve as follows: If t_i started before t_j, then t_j is restarted (t_i "wounds" t_j); otherwise, t_i is blocked (t_i "waits for" t_j). Thus, a transaction can only be blocked by an older transaction, and a transaction can kill any younger one it conflicts with.

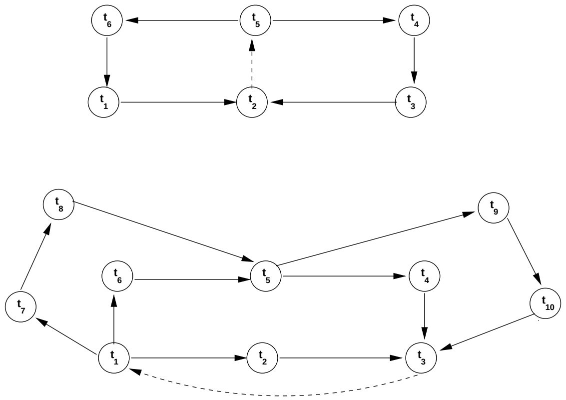

Figure 4.6: Waits-for graphs.

3. *Immediate restart*: If a lock request from t_i leads to a conflict with another transaction t_j, simply restart t_i. Thus, no transaction is ever blocked.

4. *Running priority*: If a lock request from t_i results in a conflict with t_j, resolve as follows: If t_j is currently waiting due to another conflict, then restart t_j and grant t_i's lock request; otherwise, block t_i. Thus, blocked transactions are not allowed to impede the progress of active ones.

Notice that in these approaches to deadlock prevention, a transaction that is restarted has not necessarily been involved in a deadlock, which indicates the conservativeness of the approaches.

Finally, the third category for deadling with deadlocks comprises just the *timeout* strategy: For each transaction t the scheduler maintains a timer that is activated as soon as a step of t is blocked. If the timer times out, t is aborted under the *assumption* that it has been involved in a deadlock. Clearly, this decision may be wrong, so the choice of the time period for which the timer runs is crucial. Again observe that an aborted transaction has not necessarily been involved in a deadlock.

timeouts

4.3.4 Variants of 2PL

From Figure 4.4 above we can immediately derive the following two variants of 2PL:

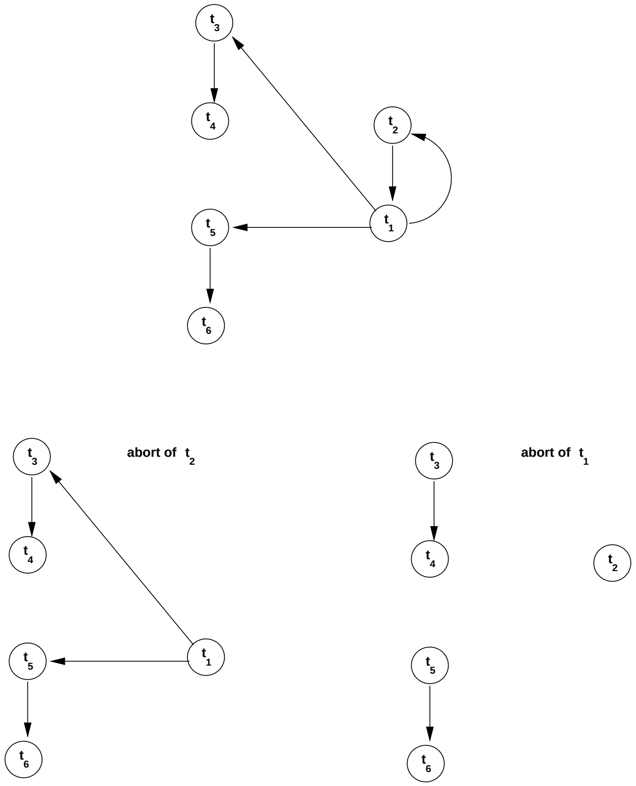

Figure 4.7: On choosing a deadlock victim.

Definition 4.12 (Conservative 2PL)

Under *static* or *conservative* 2PL (C2PL) each transaction sets all locks that it needs in the beginning, i.e., before it executes its first r or w step. This is also known as *preclaiming* all necessary locks up front.

The conservative 2PL variant is illustrated in Figure 4.8.

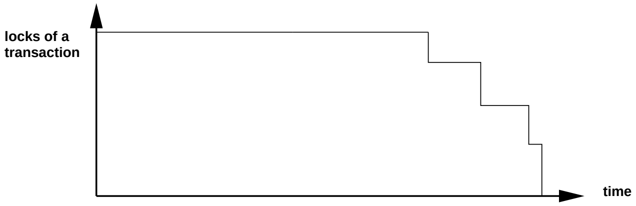

Figure 4.8: Conservative 2PL (preclaiming).

An advantage of C2PL over 2PL is that deadlocks are avoided, since a transaction can only aquire all its locks prior to execution if these do not conflict with any others; thus, the scheduler never needs to abort transactions due to their involvement in a deadlock. On the other hand, each transaction has to declare its entire *read set* as well as its *write set* to the scheduler in advance, which is possible in restricted application scenarios only. When read and write sets of a transaction are known to the scheduler, it can try to obtain all necessary locks on behalf of that transaction; if that is impossible, the transaction has to wait. It follows that, under C2PL, a blocked transaction never holds a lock.

conservative 2PL (preclaiming)

Definition 4.13 (Strict 2PL)

Under *strict* 2PL (S2PL) all (exclusive) write locks a transaction has acquired are held until the transaction terminates.

The strict 2PL variant is illustrated in Figure 4.9, assuming that all locks are exclusive. Strict 2PL is the protocol mostly used in real-world implementations. The strict variant of 2PL is motivated by the observation that a scheduler — as long as the schedule it produces is incomplete — cannot be sure that a transaction t

strict 2PL

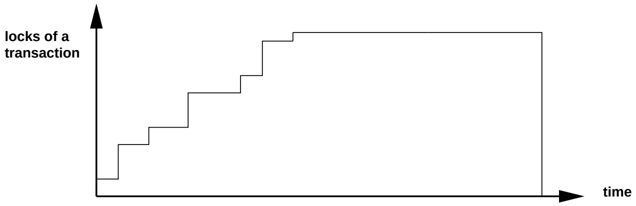

Figure 4.9: Strict 2PL (locking until end of transaction).

will not need any further locks and hence cannot release a lock held by t until the end of t. An S2PL scheduler has the interesting additional property of producing "strict" histories as output, which we will return to later in the context of transaction recovery.

It can be shown that it can make a difference whether *all* locks of a transaction are held until its commmit, or this is true for write locks only. This leads to the following third variant of 2PL:

Definition 4.14 (Strong 2PL)

Under *strong* 2PL (SS2PL), all locks, i.e., both (exclusive) write and (shared) read locks, a transaction has acquired are held until the transaction terminates.

The difference between strict and strong 2PL, in terms of their "scheduling power", becomes clear in the following theorem. Note that all inclusions in the theorem are proper.

Theorem 4.15

$$Gen(\text{SS2PL}) \subset Gen(\text{S2PL}) \subset Gen(\text{2PL})$$

Moreover, the following relationship holds, to be exploited in the context of distributed systems:

Theorem 4.16

$$Gen(\text{SS2PL}) \subset \text{COCSR}$$

We will return to this issue in Chapter 11 when we discuss correctness notions for transaction recovery.

4.3.5 Ordered Sharing of Locks

In this subsection we introduce a generalization of 2PL that is less restrictive than 2PL, and which allows one to generate a larger subclass of CSR schedules. Recall the example from the end of Subsection 4.3.2 that showed that 2PL cannot generate all schedules in the class OCSR of order-preserving conflict serializable schedules. This example, history

$$s = w_1(x)r_2(x)r_3(y)c_3r_2(z)c_2w_1(y)c_1,$$

is not in Gen(2PL), since an initial write lock of t_1 on x would block t_2. On the other hand, $s \in$ CSR and even $s \in$ OCSR. The relaxation we are about to introduce will allow situations such as the one in s for the first two operations, and will allow the two conflicting locks to be held *simultaneously*.

The two lock modes we have used so far are *exclusive* (or *non-shared*) as well as *non-exclusive* (or *shared*), and one of the locking rules said that only two non-exclusive locks on the same data item may be set on behalf of distinct transactions at the same time, in order to share that data item. We now define a third lock mode as follows, thereby relaxing the compatibility of locks used so far:

> Two locks on the same data item, whether due to conflicting operations or not, can be held simultaneously by distinct transactions as long as the lock operations and the corresponding data operations are executed in the same order.
> Note that the corresponding scheduler thus needs to be more sophisticated than a "pure" lock manager, as it needs some bookkeeping about the ordering of operations in addition to the lock table.

ordered sharing

We call this mode *ordered sharing* and denote the corresponding relationship between lock operations by $pl_i(x) \to ql_j(x)$, $i \neq j$. Thus, given a schedule s, $pl_i(x) \to ql_j(x)$ implies $pl_i(x) <_s ql_j(x)$ and $p_i(x) <_s q_j(x)$. Ordered sharing from transaction t_i to transaction t_j hence indicates that t_j can acquire a lock on data item x even though t_i holds a (possibly conflicting) lock on x, as long as the ordered-sharing property will be guaranteed.

Ordered sharing is of course meant to replace the former exclusive mode, and in terms of the lock compatibility table shown earlier, we can replace each "−" in this table by "→", individually or in combination. We thus obtain eight compatibility or lock tables as shown in Figure 4.10. As before, transaction t_i holds a lock in these

LT_1	$rl_i(x)$	$wl_i(x)$
$rl_j(x)$	+	−
$wl_j(x)$	−	−

LT_2	$rl_i(x)$	$wl_i(x)$
$rl_j(x)$	+	→
$wl_j(x)$	−	−

LT_3	$rl_i(x)$	$wl_i(x)$
$rl_j(x)$	+	−
$wl_j(x)$	→	−

LT_4	$rl_i(x)$	$wl_i(x)$
$rl_j(x)$	+	−
$wl_j(x)$	−	→

LT_5	$rl_i(x)$	$wl_i(x)$
$rl_j(x)$	+	→
$wl_j(x)$	→	−

LT_6	$rl_i(x)$	$wl_i(x)$
$rl_j(x)$	+	−
$wl_j(x)$	→	→

LT_7	$rl_i(x)$	$wl_i(x)$
$rl_j(x)$	+	→
$wl_j(x)$	−	→

LT_8	$rl_i(x)$	$wl_i(x)$
$rl_j(x)$	+	→
$wl_j(x)$	→	→

Figure 4.10: Selective use of ordered sharing in lock tables.

tables, whereas transaction t_j is requesting one, and each table shows which types of locks can be shared, ordered shared, or not shared. Notice that lock table LT_1 is the one previously used in 2PL, in which *no* exclusive lock is replaced by ordered sharing.

Example 4.17

Consider the following history:

$$s_1 = w_1(x)r_2(x)r_3(y)c_3w_1(y)c_1w_2(z)c_2$$

Suppose locks are set as requested by the operations as given, and imagine we replace the old lock compatibility table, i.e., table LT_1 in Figure 4.10, by lock table LT_2. A corresponding scheduler could then produce the following output:

$$wl_1(x)w_1(x)rl_2(x)r_2(x)rl_3(y)r_3(y)ru_3(y)c_3wl_1(y)$$
$$w_1(y)wu_1(x)wu_1(y)c_1wl_2(z)w_2(z)ru_2(x)wu_2(z)c_2$$

In particular, the write lock of t_1 on x and the read lock of t_2 on the same data item are now permitted simultaneously. Notice that lock tables LT_5, LT_7, and LT_8 would allow the same history.

Next consider the following history:

$$s_2 = r_1(x)w_2(x)r_3(y)c_3w_1(y)c_1w_2(z)c_2$$

By a similar argument, s_2 would be in the fixpoint set of a scheduler based on LT_3 (i.e., admitted by the scheduler without any blocks or reorderings of steps), and the same holds for schedulers based on LT_5, LT_6, or LT_8. Finally,

$$s_3 = w_1(x)w_2(x)r_3(y)c_3w_1(y)c_1w_2(z)c_2$$

would be allowed by a scheduler based on table LT_4, LT_6, LT_7, or LT_8. □

Although it should be intuitively clear how a locking protocol based on one of the tables from Figure 4.10 works, we now discuss this in more detail. Specifically, we need to extend our previous definitions by the following *lock acquisition rule*:

lock acquisition rule for ordered sharing

OS1: In a schedule s, for any two operations $p_i(x)$ and $q_i(x)$, $i \neq j$, s.t. $pl_i(x) \rightarrow ql_j(x)$ is permitted, if t_i acquires $pl_i(x)$ before t_j acquires $ql_j(x)$, then the execution of $p_i(x)$ must occur before the execution of $q_j(x)$.

The lock acquisition rule OS1 alone is not sufficient for guaranteeing conflict serializability. Consider the following counterexample:

$$wl_1(x)w_1(x)wl_2(x)w_2(x)wl_2(y)w_2(y)wu_2(x)wu_2(y)c_2wl_1(y)w_1(y)wu_1(x)wu_1(y)c_1.$$

This schedule would be allowed by the locking rules LR1–LR4 and OS1. The order of lock acquisitions on x entails a serialization order with t_1 before t_2. Once t_2 holds its write lock on y and has actually written y, t_1 would no longer be granted a write lock on y as this would lead to the contradicting order with t_2 before t_1. In fact, rule OS1 disallows this lock acquisition. However, when t_2 commits and thus releases all its locks, there is no "marker" left that would prevent t_1 from acquiring the lock on y at some later point. This situation leads to a conflict cycle in the above example. The solution to this problem is to enforce also an ordering of unlock steps. If t_2 would be prevented from unlocking y until after t_1 starts its unlocking phase and thus would no longer be able to acquire new locks under a two-phase regime, then the second half of the above incorrect schedule would be disallowed but would still allow situations like the ordered sharing of the two write operations on x.

So for a correct protocol we need a rule for unlocking. If we have a relationship $pl_i(x) \rightarrow ql_j(x)$ of ordered sharing between two transactions t_i and t_j and t_i has not yet released any lock, t_j is called *order-dependent* on t_i. If there exists any such t_i, transaction t_j is *on hold*. Now we have:

lock release rule for ordered sharing

OS2: While a transaction is on hold, it cannot release any of its locks.

In summary, we now obtain a family of locking protocols by using

1. locking rules LR1–LR4,
2. rules OS1 and OS2,
3. the two-phase property,
4. one of the eight compatibility tables shown in Figure 4.10.

O2PL

Notice that these protocols reduce to ordinary 2PL if ordered sharing is not used and locking is correspondingly based on table LT_1. If locking is based on LT_8, we will call the resulting protocol the *O2PL protocol* for a reason that will become clear shortly.

Example 4.18

Consider an O2PL scheduler which receives the following input:

$$s = r_1(x)w_2(x)r_3(y)w_2(y)c_2w_3(z)c_3r_1(z)c_1$$

It will produce the following output:

$$s' = rl_1(x)r_1(x)wl_2(x)w_2(x)rl_3(y)r_3(y)wl_2(y)w_2(y)wl_3(z)w_3(z)$$
$$ru_3(y)wu_3(z)c_3rl_1(z)r_1(z)ru_1(x)ru_1(z)wu_2(x)wu_2(y)c_2c_1$$

This resulting execution is also illustrated in Figure 4.11, where dashed lines correspond to waiting periods. Notice that t_2 is order-dependent on t_1 and t_3, so it is on hold when c_2 arrives. According to lock rule OS2, t_2 cannot release any lock until the others have, which here means it has to unlock last. □

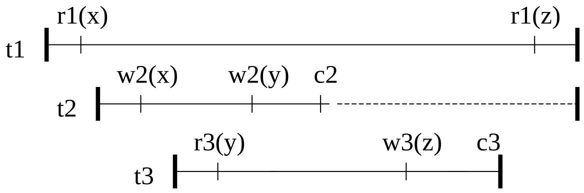

Figure 4.11: Example execution under the O2PL protocol.

The following can now be verified:

Lemma 4.19

Let x be locked by pl_i and ql_j in ordered shared mode, i.e., $pl_i(x) \rightarrow ql_j(x)$, and let $pl_i(x) < ql_j(x)$, then $p_i(x) < q_j(x)$.

The above follows from the lock acquisition rule OS1, and the following is a consequence of the unlock rule OS2:

Lemma 4.20

If $pl_i(x) \rightarrow ql_j(x)$ and $pl_i(x) < ql_j(x)$, then there exists an unlock operation of t_i preceding *all* unlock operations of t_j.

Finally, for truly incompatible locks, for which ordered sharing is disallowed according to the chosen lock compatibility table, the previous property can be strengthened as follows, corresponding to what can be said about standard 2PL:

Lemma 4.21

If $pl_i(x)$ is incompatible with $ql_j(x)$ (i.e., without allowing order sharing) and $pl_i(x) < ql_j(x)$, then $pu_i(x) < ql_j(x)$. That is, if t_i locks x before t_j and the two locks needed by these transactions are incompatible, then t_i has to release its lock on x before t_j can set one. (Note that this case does not occur in the O2PL protocol, as its lock compatibility table LT_8 allows ordered sharing for all conflict pairs.)

Now we can establish an argument similar to the one as we had for the correctness of 2PL. Indeed, let s be a history produced by a 2-phase protocol based on any of the new lock tables. Then an edge (t, t') in $G(s)$, the conflict graph of s, indicates that there exists an unlock operation of t preceding all unlock operations of t'. The same holds along a path in $G(S)$, so if $G(s)$ was cyclic, an unlock operation of some t along the path would precede all unlocks of t itself, a contradiction. Thus we have:

safety of ordered sharing

Theorem 4.22

Let $Gen(LT_i)$, $1 \leq i \leq 8$, denote the set of histories produced by a protocol defined as above and using lock table LT_i, then $Gen(LT_i) \subseteq$ CSR.

Theorem 4.22 in particular implies that O2PL (i.e., the protocol based on LT_8) generates histories in CSR. Let $s \in Gen$(O2PL). Then there is a serial history s' conflict equivalent to s in which all conflicting operations occur in the same order as in s, due to the use of the ordered shared mode now for *every* pair of conflicting operations. Since s' is serial, this proves membership of s even in OCSR. In conclusion, we have:

Theorem 4.23

Gen(O2PL) $\subseteq$ OCSR

An interesting point is that the converse of this latter statement is also true. To this end, consider some $s \in$ OCSR. For each data operation in s the corresponding lock can be set immediately prior to the operation; note that even O2PL allows this. Let $t_1 \ldots t_n$ be the order-preserving conflict equivalent serial order of the transactions in s; we may further assume that all unlock operations occur after all data operations in s, yet are in serialization order. Since O2PL permits that locks are released in exactly that order, it is possible to generate s using the O2PL protocol. Thus we have:

scheduling power of O2PL

Theorem 4.24

OCSR $\subseteq Gen$(O2PL)

Corollary 4.25

$Gen(\text{O2PL}) = \text{OCSR}$

Despite these intriguing properties and its theoretical superiority over 2PL, ordered sharing is not necessarily the method of choice in practice. The reason is that it is not a truly "pure" locking protocol in that its implementation cannot be solely based on a lock table. Rather, ordered sharing needs to enforce the proper ordering of data accesses whenever it allows the sharing of write locks. Thus, an implementation requires additional data structures for bookkeeping and would incur additional run-time overhead. Depending on application and workload characteristics, the gain in concurrency may not be worth this additional overhead.

4.3.6 Altruistic Locking

We next present an extension to the 2PL protocol which is based on the following observation. Suppose we are given a transaction of long duration, i.e., a transaction that reads and writes many database items, performs extensive computations, pauses for user input, maybe all this in combination, and hence lives longer than most other transactions. Processing such a transaction with 2PL may result in serious performance problems, due to the fact that locks are held by this transaction for long periods of time, and hence other transactions wishing to access the same items have to face delays. In addition, long transactions are more likely than short ones to get involved in deadlocks. One conceivable way of handling long transactions could be to use the approach of interleaving specifications we have described in Chapter 3, and to specify how long transactions can interleave with others, thus relaxing serializability. However, as pointed out in Chapter 3, such techniques require additional considerations in the application design, as opposed to the automatically provided guarantee of serializability.

long transactions

In this subsection, we look at another approach, which does preserve serializability and whose motivation derives from the following example.

Example 4.26

Consider a database $D = \{a, b, c, d, e, f, g\}$, and suppose that transaction t_1 needs to access all seven items, say, in alphabetical order a through g. Further assume that while t_1 is accessing item d, three short transactions with the following data requirements attempt to run:

t_2: access to a and b
t_3: access to c and e
t_4: access to f and g

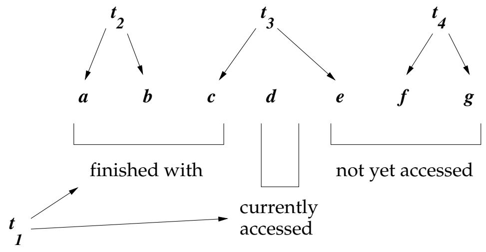

Figure 4.12: A long transaction delaying short ones.

If 2PL is used for scheduling these transactions, both t_2 and t_3 have to wait until t_1 finishes, since both want to access data items that are still locked by t_1; note that, under 2PL, t_1 cannot unlock an item until it has obtained a lock on the last item g. On the other hand, t_4 can obtain locks on f and g and hence execute concurrently with t_1. The situation is illustrated in Figure 4.12.

With a little additional information, the scheduler could also allow t_2 to run concurrently with t_1. Suppose that, although t_1 still holds locks on a and b, it has finished accessing these items. As long as t_2 (and similarly other transactions) accesses only such data items that t_1 is finished with, it can appear in a serialization of the schedule in question *after* t_1. □

The key point of the above example is that the short transactions t_2 through t_4 access only *subsets* of the data items that are read or written by the long transaction t_1, and that such access profiles are often (but not always) known in advance. What is needed for such situations is a mechanism by which transactions like t_1 can inform the scheduler that they are finished with operating on a data item. This information could then be used to let other transactions access that item.

idea of altruistic locking (AL)

Such a mechanism is delivered by an extension of the 2PL protocol called *altruistic locking* (AL). As before, a transaction locks and unlocks items in a two-phase manner. However, unlike 2PL and similar to ordered sharing we discussed in the previous subsection, several transactions may hold conflicting locks on an item simultaneously under certain conditions. In terms of Example 4.26, AL will allow t_2 to acquire locks on items a and b although t_1 holds locks on them. In order to make this work, AL uses a third access-control operation, besides lock and unlock, which is called *donate*. The donate operation is used to inform the scheduler that access to a data item is no longer required by the transaction that has currently locked the item, so that the item can be "donated" for access to another transaction. The donating transaction is free to acquire other locks in the future, so that lock and donate operations do not need to follow a two-phase rule, but other than that the transaction still has to be two-phase w.r.t. unlock operations.

The use of donate on item x by transaction t_i, subsequently denoted $d_i(x)$ has to follow certain rules, which are stated next:

AL rules

AL1: Items cannot be read or written by t_i once it has donated them, i.e., if $d_i(x)$ and $o_i(x)$ occur in a schedule s, $o \in \{r, w\}$, then $o_i(x) <_s d_i(x)$.

AL2: Donated items are eventually unlocked, i.e., if $d_i(x)$ occurs in a schedule s following an operation $o_i(x)$, then $ou_i(x)$ is also in s and $d_i(x) <_s ou_i(x)$.

AL3: Transactions cannot hold conflicting locks simultaneously, unless one has donated the data item in question, i.e., if $o_i(x)$ and $p_j(x)$, $i \neq j$, are conflicting operations in a schedule s and $o_i(x) <_s p_j(x)$, then either $ou_i(x) <_s pl_j(x)$, or $d_i(x)$ is also in s and $d_i(x) <_s pl_j(x)$.

Example 4.27

Consider again the situation from Example 4.26, and suppose that each access by one of the transactions is a write operation. Then the following schedule (in which t_1 and t_3 are unfinished) obeys the AL rules just stated:

$$\begin{aligned} s = \; & wl_1(a)w_1(a)d_1(a)wl_2(a)wl_1(b)w_2(a)w_1(b)d_1(b)wl_2(b)w_2(b) \\ & wl_1(c)ul_2(a)w_1(c)ul_2(b)c_2d_1(c)wl_3(c)w_3(c)wl_4(f)w_4(f) \\ & wl_1(d)wl_4(g)w_4(g)w_1(d)ul_4(g)ul_4(f)c_4d_1(d) \end{aligned}$$

Notice how t_2 can run concurrently with t_1, once t_1 starts donating items a and b. Transaction t_4 can even run independently of t_1, since t_1 has not yet reached items f and g. However, t_3 can not yet finish, as only part of the items it wants to access are already donated by t_1. □

Example 4.27 gives rise to some more terminology: Intuitively, if transaction t_j locks a data item that has been donated and not yet unlocked by transaction t_i, $i \neq j$, we say that t_j is *in the wake* of t_i. More formally, we have the following:

wake and indebtedness

1. (i) An operation $p_j(x)$ from transaction t_j is *in the wake* of transaction t_i, $i \neq j$, in the context of a schedule s if $d_i(x) \in \text{op}(s)$ and $d_i(x) <_s p_j(x) <_s ou_i(x)$ for some operation $o_i(x)$ from t_i.
2. (ii) A transaction t_j is *in the wake* of transaction t_i if some operation from t_j is in the wake of t_i. Transaction t_j is *completely* in the wake of t_i if all of its operations are in the wake of t_i.

3. (iii) A transaction t_j is *indebted* to transaction t_i in a schedule s if $o_i(x), d_i(x), p_j(x) \in \text{op}(s)$ s.t. $p_j(x)$ is in the wake of t_i and either $o_i(x)$ and $p_j(x)$ are in conflict or some intervening operation $q_k(x)$ s.t. $d_i(x) <_s q_k(x) <_s p_j(x)$ is in conflict with both $o_i(x)$ and $p_j(x)$.

The notions wake and indebtedness are the basis for another AL rule which is motived by the following examples. First consider the following history:

$$\begin{aligned} s_1 = \; & wl_1(a)w_1(a)d_1(a)rl_2(a)r_2(a)rl_2(b)r_2(b)ru_2(a)ru_2(b)c_2 \\ & rl_1(b)r_1(b)wu_1(a)ru_1(b)c_1 \end{aligned}$$

Clearly, s_1 is conflict serializable. Moreover, when $d_1(a)$ is dropped from s_1, the remaining history is in *Gen*(2PL). However, s_1 should not be admissible under altruistic locking, since transaction t_2 first enters the wake of t_1 by issuing $rl_2(a)$ and then *leaves* the wake again by issuing $rl_2(b)$, which should be prohibited. The intuitive reason is that, once a transaction has entered the wake of another transaction, it should get in there completely (and stay there) until that other transaction issues its first unlock operation, since otherwise it can no longer be guaranteed in general that the two get serialized in the appropriate order (determined by the wake relationship). For example, if in history s_1 above $r_1(b)$ were replaced by $w_1(b)$ the resulting schedule would no longer be conflict-serializable, but the locking protocol would not be able to prevent this conflict cycle as t_2 has already released its lock on b by the time when t_1 acquires its lock on b.

The solution to avoid this problem is to require t_2 to stay in the wake of t_1 once it has made use of t_1's donation and issued an operation that conflicts with t_1. Obviously, this makes sense only if it is known at this point that t_2 needs to access only a subset of the data items that have been or are going to be read by t_1. Most typically, this is the case when t_1 performs a table scan and t_2 is confined to access a few records of this table. If this kind of opportunity cannot be inferred at the time of the first conflict between t_1 and t_2, t_2 may choose to ignore the donation upfront and wait for t_1 to unlock the requested data item.

Now consider the following history:

$$\begin{aligned} s_2 = \; & rl_1(a)r_1(a)d_1(a)wl_3(a)w_3(a)wu_3(a)c_3rl_2(a)r_2(a)wl_2(b) \\ & ru_2(a)w_2(b)wu_2(b)c_2rl_1(b)r_1(b)ru_1(a)ru_1(b)c_1 \end{aligned}$$

History s_2 shows that if a transaction is forced into another transaction's wake only when one of its locks conflict with a donated lock, this can still result in incorrect behavior. Indeed, in s_2 transaction t_2 is now assumed not to enter the wake of t_1 when it reads data item a since t_1 also just reads this item. However, $s_2 \notin \text{CSR}$. This tells us that even *indirect conflicts* between t_1 and t_2 (i.e., the order $r_1(a)w_3(a)r_2(a)$ with a third transaction's intermediate operation being in conflict with both t_1 and

t_2) matter for the requirements on when a transaction must stay in the wake of another transaction.

The notion of indebtedness is intended to establish a condition under which a transaction should enter the wake of another transaction. We state this condition as another AL rule:

AL4: When a transaction t_j is indebted to another transaction t_i, t_j must remain completely in the wake of t_i until t_i begins to unlock items, i.e., for *every* operation $p_j(x)$ occuring in a schedule s either $p_j(x)$ is in the wake of t_i or there exists an unlock operation $ou_i(y)$ in s s.t. $ou_i(y) <_s o_j(x)$.

another AL rule

With these preparations, it is now easy to establish the *altruistic locking protocol*, which we again abbreviate by AL: A scheduler operates according to the AL protocol (or is an *AL scheduler*) if it is a 2PL scheduler and obeys the AL rules AL1–4. In the sequel, let *Gen*(AL) denote the set of histories the AL protocol can generate as output. Now the following is straightforward:

AL protocol

Theorem 4.28

$Gen(\text{2PL}) \subset Gen(\text{AL})$

The reason why the AL protocol can generate at least all histories that are admissable under 2PL is because the 2PL rules are contained in the AL protocol; indeed, if no donations are performed, AL reduces to 2PL. The following history shows that the inclusion is strict:

$$s = r_1(x)w_2(x)c_2w_3(y)c_3r_1(y)c_1$$

Under 2PL, a read lock on x set by t_1 would conflict with a write lock on x from t_2, so t_2 (or t_3, because of y) would not be allowed to run concurrently with t_1, so $s \notin Gen(\text{2PL})$. Under the AL protocol, however, transaction t_1 can donate x to t_2, thereby allowing t_2 to become indebted to t_1 and to run in its wake. □

Thus, AL provides more concurrency than 2PL. On the other hand, not all conflict serializable histories can be generated by the AL protocol.

correctness of the AL protocol

Theorem 4.29

$Gen(\text{AL}) \subset \text{CSR}$

The full proof that *Gen*(AL) is contained in CSR is left to the reader as an Exercise (see Exercise 10). It essentially follows a standard argument, namely that any

AL-generated history s has an acyclic conflict graph $G(s)$ (from which containment then follows by the serializability theorem from the previous chapter). It can be shown that each edge of the form $t_i \to t_j$ in such a graph $G(s)$ is either a "wake edge", indicating that t_j is completely in the wake of t_i, or a "crest edge", indicating that t_i unlocks some item before t_j locks some item. In addition, for every path $t_1 \to \ldots \to t_n$ in $G(s)$, there is either a wake edge from t_1 or t_n, or there exists some t_k on the path s.t. there is a crest edge from t_1 to t_k. These properties suffice to prove the claim. To see that the containment is strict, consider history

$$s^* = r_1(x)r_2(z)r_3(z)w_2(x)c_2w_3(y)c_3r_1(y)r_1(z)c_1.$$

Clearly, $s^* \approx_c t_3t_1t_2$, so $s^* \in \text{CSR}$. On the other hand, either x or y (or both) must be locked by t_1 between operations $r_1(x)$ and $r_1(y)$. By rule AL1, either x or y (or both) must be donated by t_1 for $w_2(x)$ and $w_3(y)$ to occur, so either t_2 or t_3 (or both) must be indebted to t_1. However, neither $r_2(z)$ nor $r_3(z)$ are allowed to be in the wake of t_1 if the latter is well-formed, since t_1 later reads z. Hence either t_2 or t_3 violate rule AL4. □

We mention (without proof) that the set *Gen*(AL) has other interesting properties. For example, it has a non-empty intersection with the class COCSR of commit-order-preserving histories (as any serial history is in both classes), but is otherwise incomparable w.r.t. set inclusion with COCSR. Indeed, a slight modification of history s^* above in which c_2 is moved to the very end makes s^* commit-order preserving, but still not acceptable under AL. Conversely, history

$$r_1(x)w_2(x)c_2c_1$$

is in *Gen*(ALT), but not in COCSR. We finally mention that a similar observation can be made for class OCSR discussed in the previous subsection and in Chapter 3; again it has a non-empty intersection with *Gen*(AL), but is otherwise incomparable to it. To this end, recall from the previous subsection that in a schedule s, $pl_i(x) \to ql_j(x)$ $(i \neq j)$ implies $pl_i(x) <_s ql_j(x)$ and $p_i(x) <_s q_j(x)$, and operations p_i and q_j (or, more precisely, transactions t_i and t_j) must not finish in arbitrary order. In particular, t_j has to wait with releasing its locks until t_i is finished, which is no longer the case under AL.

4.3.7 Non-Two-Phase Locking Protocols

A crucial point for most of the schedulers discussed so far has been a guarantee of the 2-phase property of transactions. We now present protocols that can do without this property, and which are even deadlock-free. We should note upfront, however, that these protocols are geared for very specific access patterns. Thus, although they are developed as general-purpose protocols, they perform well only under such specific conditions and may be susceptible to degradation in settings for which they are not suited.

As we will see, the two-phase property is essentially replaced by ordering the accesses performed on the data; in other words, transactions have to follow certain access patterns. To this end, we will first assume that the access patterns follow a hierarchical organization of the data items; that is, the data items are viewed as the nodes of a tree, and accesses have to follow a path down the tree. If all accesses performed are write operations only, it is easily seen that the 2-phase property can be relaxed. However, when transactions are allowed to perform read and write operations as before, we will see that additional measures are needed to ensure serializability, since one transaction could now bypass another while moving down the tree. The extra condition that avoids such races in the tree carries over to database organizations that are arbitrary directed acyclic graphs (DAGs).

4.3.7.1 Write-Only Tree Locking

In this subsection, we consider a restricted transaction model in which read operations are missing. Thus, a transaction can write a data item (or you may think of read and write applied to the same item as collapsed into one operation). Now consider a data-item tree shown in Figure 4.13. Note that this tree is a "virtual" data organization only in that the actual relationships (e.g., references) among data items may be quite different from such a hierarchy. The important point is that transactions exhibit tree-like access patterns and are essentially top-down traversals of the tree. So we will require that write accesses to data items have to follow the order given by the tree. For example, if an access to item d is requested in the tree of Figure 4.13, a and then b must be accessed first. We mention that the view that data accesses have to follow some order is frequently given in reality, for example through the existence of index structures such as B$^+$-trees, which are accessed before actual data is touched. Specific protocols for B$^+$-tree indices will be presented in Chapter 9, and we will see then that the generic tree-oriented protocols that we are going to derive now form valuable building blocks for concurrency control on search structures.

It turns out that the enforcement of a certain access order for data items can serve as a replacement for the two-phase property. Under the *write-only tree locking (WTL)* protocol, lock requests and releases must obey the locking rules LR1–LR4 and the following additional rules for each transaction:

write-only tree locking (WTL)

WTL1: If x is any node in the tree other than the root, $wl_i(x)$ can be set only if t_i currently holds a write lock on y, where y is the parent of x.

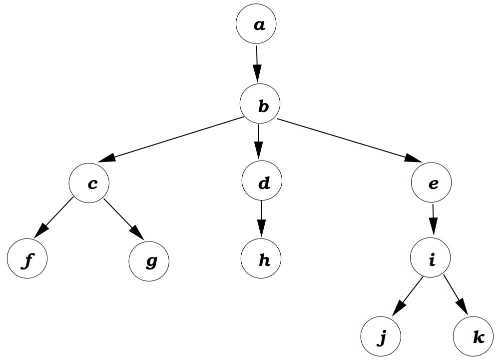

Figure 4.13: Tree organization of data items.

WTL2: After a $wu_i(x)$, no further $wl_i(x)$ is allowed (on the same data item x).

Rules WTL1 and WTL2 of the protocol imply that the scheduler can unlock an item x only after it has obtained locks on the relevant children of x.

Example 4.30

Consider the tree from Figure 4.13 as well as transaction $t = w(d)w(i)w(k)$. Under the WTL protocol, t will be transformed into the following locked transaction (where accesses are underlined):

$wl(a)wl(b)wu(a)wl(d)wl(e)wu(b)\underline{w(d)}wu(d)wl(i)wu(e)\underline{w(i)}wl(k)wu(i)\underline{w(k)}wu(k)$ □

safety of WTL

As a verification of the WTL protocol, let $(x, z_1, \ldots, z_n, v)$ be a path in a given database tree from x to v, and assume (i) that t_i locks x *before* t_j does, and (ii) that both t_i and t_j lock v. Then we can conclude that, due to the existence of $wl_i(x)$, lock $wl_j(x)$ cannot be set, and $wl_i(z_1) < wu_i(x)$ (Rule WTL1). Thus, $wl_i(z_1) < wl_j(z_1)$, and the same holds along the entire path, which proves the following:

Lemma 4.31

If transaction t_i locks x before t_j does, then each successor v of x in the data tree that is locked by both t_i and t_j is also locked by t_i before it is locked by t_j.

Now this implies:

Theorem 4.32

$Gen(\text{WTL}) \subseteq \text{CSR}$

Proof. Let (t_i, t_j) be an edge in the conflict graph G of an output of the WTL protocol. Then there are operations $w_i(x)$ and $w_j(x)$ in conflict s.t. $w_i(x)$ occurs before $w_j(x)$ in the output. According to the WTL rules, t_i unlocks x before t_j locks x. By Lemma 4.31, t_i locks the root of the data tree before t_j does, since x is a successor of the root and every transaction locks the root, and otherwise a contradiction to the lemma would result. But then t_i unlocks the root before t_j locks it. By induction it is then easily seen that the same applies to transactions t_i and t_j which are connected via a path in G. Now if G was cyclic, say, with cycle $(t_1, t_2, \ldots, t_n, t_1)$, it would follow that t_1 unlocks the root before t_1 locks it, an obvious contradiction to Rule WTL2. □

A similar consideration reveals that the WTL protocol has the nice property of being immune to deadlocks.

Theorem 4.33

The WTL protocol is deadlock-free.

Proof. If t_i waits for a lock on the root, it cannot have set any other lock yet. If t_i waits for a lock held by t_j (on a node which is not the root), then t_j unlocks the root before t_i locks it. Again by induction we may then conclude that if the wait-for graph has a cycle involving t_i, then t_i unlocks the root before t_i locks it, a contradiction. □

We mention that the WTL protocol can be generalized in various directions which may, however, require additional rules. For example, it could be allowed to lock *any* node in the data tree first and thereafter only data items in the subtree rooted at this node; in other words, locking the root initially is no longer required.

Finally note that, although the protocol is specifically geared for tree-like access patterns, it is a general-purpose protocol in the sense that it can be applied to arbitrary data access patterns, too. However, being forced to obey the protocol's four tree-oriented locking rules could then easily result in too many locks or unacceptable lock duration. For example, if a transaction were to access the nodes f, i, and b of the tree in Figure 4.13 in this order, these three accesses would have to be embedded in a virtual top-down tree traversal, requiring locks as if the transaction accessed all paths to these nodes. In particular, we would need to acquire a lock on the root a

first and this lock would have to be retained until we learn about the transaction's last access to b and acquire a lock on b; only then can we release the lock on the root a. So the lock on the critical root would be held for almost the entire duration of the transaction. Obviously, performance is likely to degrade in such situations. The bottom line is that tree locking is beneficial only for truly tree-like access patterns, but it can cope with occasional non-tree-like transactions.

4.3.7.2 Read-Write Tree Locking

We now generalize the WTL protocol introduced in the previous subsection "back" to read and write operations. To see what could cause a problem and why we looked at write operations only in the first place, consider once more the tree from Figure 4.13. Let t_1 and t_2 be two transactions, both accessing items a, b, e, i, and k in that order. Then the following could be a prefix of an acceptable schedule:

$$wl_1(a)w_1(a)wl_1(b)ul_1(a)wl_2(a)w_1(b)wl_1(e)w_2(a)\ldots$$

As we said in the previous subsection, the crucial points are that transactions need to move along the tree by at least keeping neighboring nodes locked (as in $wl_1(b)ul_1(a)$), and that two transactions doing the same thing must strictly follow each other. The important implication of this is that no transaction can pass another transaction; in the example, both t_1 and t_2 have to follow the $a - b - e - i - k$ pattern.

Now consider the case where reads and writes are again allowed, and assume we still follow the WTL rules from above. Since a transaction might not want to read *and* write each data item it encounters, it may just read some items, in order to get to those it wants to write. As an example, consider the following two transactions operating on the tree of Figure 4.13:

$$t_1 = r_1(a)r_1(b)w_1(a)w_1(b)r_1(e)r_1(i)c_1$$

$$t_2 = r_2(a)r_2(b)r_2(e)r_2(i)w_2(i)c_2$$

Using the WTL rules but allowing read locks to be shared, the two transactions could result in a schedule like the following:

$$rl_1(a)rl_1(b)r_1(a)r_1(b)wl_1(a)w_1(a)wl_1(b)ul_1(a)rl_2(a)r_2(a)w_1(b)rl_1(e)ul_1(b)rl_2(b)r_2(b)$$

$$ul_2(a)rl_2(e)rl_2(i)ul_2(b)r_2(e)r_1(e)r_2(i)wl_2(i)w_2(i)wl_2(k)ul_2(e)ul_2(i)rl_1(i)ul_1(e)r_1(i)\ldots$$

This example, albeit along the lines of tree locking, is no longer serializable, since there are conflicts between $w_1(a)$ and $r_2(a)$ as well as between $w_2(i)$ and $r_1(i)$, so the conflict graph is cyclic. The point is that we see here how the shared locks allow t_2 to pass t_1 in the schedule, since item i is the first item that t_2 writes, so it can read to this point faster than t_1 can, which starts writing items at the top and hence needs to spend more time on obtaining the appropriate locks.

We now introduce the *read-write tree locking* RWTL protocol as a way to overcome the problem shown above. The trick is to require the two-phase property on certain subtrees which are given as a "neighborhood" of read operations called a *pitfall*. Here having the two-phase property on a set of data items means that the projection of the schedule onto these data items satisfies the rules of the two-phase locking protocol. Let t be a transaction accessing a database tree by read and write operations. The *read set* $\mathrm{RS}(t)$ of t is the set of items read by t, and the *write set* $\mathrm{WS}(t)$ of t is the set of items written by t, resp. In the presence of a tree, $\mathrm{RS}(t)$ spawns a subtree that will generally split into a number of connected components, say, $C_1, \ldots, C_m$. A *pitfall* of t is a set of the form

pitfalls

$$C_i \cup \{x \in \mathrm{WS}(t) \mid x \text{ is a child or parent of some } y \in C_i\},$$

$1 \leq i \leq m$.

As an example, consider the database tree shown in Figure 4.14. We assume that transaction t is given by $\mathrm{RS}(t) = \{f, i, g\}$ and $\mathrm{WS}(t) = \{c, l, j, k, o\}$. Then $\mathrm{RS}(t)$ has two connected components $\{f, i\}$ and $\{g\}$, and t has two pitfalls $pf_1 = \{c, f, i, l, j\}$ and $pf_2 = \{g, c, k\}$, which are the shaded areas in Figure 4.14. Notice that pitfalls do not need to be disjoint.

We are now ready to state the RWTL protocol. The rules for each transaction t_i are those of the WTL protocol with the addition of one extra rule:

read-write tree locking (RWTL)

RWTL1: The transaction has the two-phase property on each of its pitfalls, i.e., for each pitfall the transaction follows a two-phase rule with respect to setting and releasing locks on elements of that pitfall.

As is easily seen, our last schedule above on the tree from Figure 4.13 exhibits for transaction t_1 a read set $\mathrm{RS}(t_1) = \{a, b, e, i\}$, which equals a pitfall of t_1; however, t_1 is *not* two-phase on that pitfall. The argument here is that this is a reason why WTL, if naively extended to reads and writes, fails to produce CSR schedules.

In order to show the safety of the RWTL protocol, the following observations need to be proven:

Lemma 4.34

Let T be a given database tree, and let t be a transaction on T that obeys the rules of the RWTL protocol. Let $V \subseteq \mathrm{RS}(t) \cup \mathrm{WS}(t)$ span a connected subtree of T. Then the restriction $\Pi_V(t)$ of t to V also follows the rules of RWTL.

This is simply because for doing all the accesses to items in V, t just needs to act as before, i.e., according to the RWTL rules.

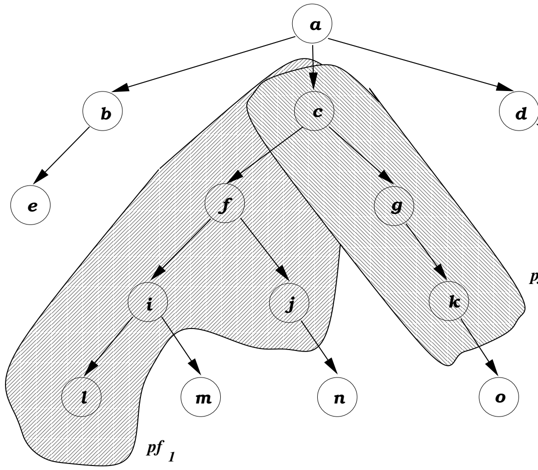

Figure 4.14: Another tree organization of data items.

Lemma 4.35

Assume that transaction t has the two-phase property on $V \subseteq \mathrm{RS}(t) \cup \mathrm{WS}(t)$, and let $W \subseteq \mathrm{RS}(t) \cup \mathrm{WS}(t)$. Then $\Pi_W(t)$ has the two-phase property on $V \cap W$.

These lemmata can be used to show the safety of RWTL:

Theorem 4.36

$Gen(\mathrm{RWTL}) \subseteq \mathrm{CSR}$

Once again, the proof assumes that the conflict graph of a schedule $s \in Gen(\mathrm{RWTL})$ has a cycle and derive a contradiction; it proceeds by induction on the length of a cycle.

We conclude this section by mentioning that the RWTL protocol can be generalized further, in particular to DAGs leading to the *DAG locking* protocol, essentially *g*

by allowing a transaction to lock any node in the given database graph first (the "entry point" of the transaction), and by replacing Rule WTL1, which makes sure the access pattern of a transaction obeys the node ordering of the given tree, by the following rule:

DAG1: A transaction may lock an item x (which is not the entry point into the graph) if it currently holds locks on the majority of predecessors of x.

4.3.8 On the Geometry of Locking

We conclude our discussion of locking schedulers by briefly mentioning a *geometric* interpretation of locking. To this end, we first give such an interpretation for the notion of conflict serializability and then extend that to transactions with lock and unlock operations.

We restrict our attention to the case of two transactions for reasons of simplicity; however, what we present in this subsection can be generalized to more than two transactions. We assume that our two transactions are completely known. For each transaction its sequence of steps is represented as a sequence of equi-distant integer points on one of the axes spanning the Euclidian plane. The two transactions then define a rectangular grid in that plane, and a schedule becomes a monotone curve starting in the origin of the coordinates. A grid point is a *conflict point* if the two steps corresponding to its coordinates (one on the x-axis, the other on the y-axis) are in conflict.

schedules as curves

Example 4.37

Consider the two transactions $t_1 = r_1(x)w_1(y)$ and $t_2 = w_2(x)r_2(y)$. The schedule $s = r_1(x)w_2(x)r_2(y)w_1(y)$ can be represented geometrically as shown in Figure 4.15. Conflict points are $(r_1(x), w_2(x))$ and $(w_1(y), r_2(y))$. □

A schedule for two transactions *separates* given conflict points if, informally, these points occur on both sides of the curve; for example, such a situation is given in Figure 4.15. We note that the schedule shown in that figure has a cyclic conflict and does hence not belong to CSR. That this is not a coincidence is shown by:

Theorem 4.38

A schedule s for two transactions t_1 and t_2 is in CSR iff the curve representing s does not separate any two conflict points.

The geometric interpretation can directly be generalized to transactions comprising lock and unlock operations: All steps of both transactions are now interpreted as

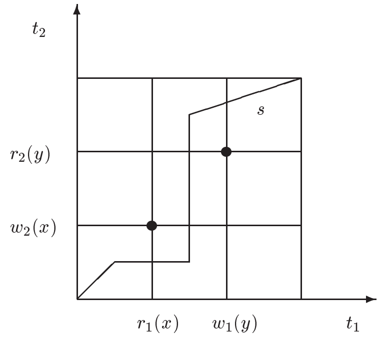

Figure 4.15: Geometrical representation of a schedule.

points along the axes of a coordinate system spanning a plane, where only the data operations have integer coordinates. A schedule is a curve as before. Conflict points turn into *conflict regions*, which are delimited by pairs of conflicting locks of the two transactions together with their respective unlock operations. Blurring the distinction between read and write locks for a moment, let us for the sake of simplicity use a single type "$l(x)$" of lock operations and a single type "$u(x)$" of unlock operation; then any two lock operations of the two transactions referring to the same data item are in conflict. Moreover, the conflict region ("of x") is the square defined by the following pairs of coordinates:

locked transactions

$$
\begin{array}{l}
(l_1(x), l_2(x)) \\
(u_1(x), l_2(x)) \\
(l_1(x), u_2(x)) \\
(u_1(x), u_2(x))
\end{array}
$$

Example 4.39

Consider the following schedule consisting of two transactions:
$s = l_2(z)w_2(z)l_1(x)w_1(x)l_1(y)w_1(y)u_1(y)l_2(y)u_2(z)l_1(z)w_1(x)$
$\quad u_1(x)l_2(x)w_2(y)w_2(x)u_2(x)w_2(y)w_1(z)u_1(z)c_1u_2(y)c_2$
(The fact that write operations may occur multiple times is allowed and of course determines the duration for which a lock needs to be held.) The geometrical interpretation of s in terms of conflict regions is shown in Figure 4.16.

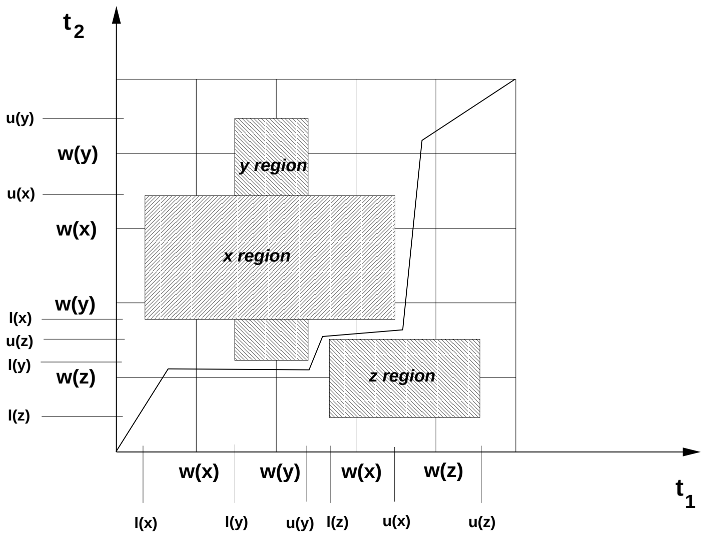

Figure 4.16: Schedule separating conflict regions.

Notice that each region corresponds to exactly one data item, and that each point within a conflict region represents a state in which both transactions hold conflicting locks on the same data item. □

Now it is clear that a schedule should avoid conflict regions, since, as we noticed, every point inside such a region means that two conflicting locks are set simultaneously. In other words, a schedule has to "bypass" conflict regions; however, it cannot do so in an arbitrary fashion due to the following generalization of the previous theorem:

Theorem 4.40

Let s be a schedule for two transactions t_1 and t_2 containing lock and unlock operations for the transactions. Then $\mathrm{DT}(s) \in \mathrm{CSR}$ iff the curve representing s does not separate conflict regions.

These results have interesting applications from a theoretical point of view, since they allow one to reason about the "safety" of a system of locked transactions as well as on the scheduling power of locking protocols. Call a *locked* transaction a transaction in which lock and unlock operations have been placed in a syntactically correct manner (i.e., following the locking well-formedness rules LR1 through LR3 of Subsection 4.3.1). For a given set T of transactions, let $L(T)$ denote a set of locked transactions s.t. there is a transaction in T for each element of $L(T)$ and vice versa; note that $L(T)$, which is called a *locking* of T, is not uniquely determined.

safety of lockings

A schedule $s \in \text{shuffle}(L(T))$ is *legal* if a step of the form $u(x)$ occurs between any two steps of the form $l(x)$ (i.e., a data item must be unlocked before it can be locked again). For example, the schedule shown in Example 4.39 is legal. Next, a schedule $s \in \text{shuffle}(T)$ is *legal w.r.t.* $L(T)$ if there exists a legal schedule $s' \in \text{shuffle}(L(T))$ s.t. $\mathrm{DT}(s') = s$. A locking $L(T)$ of T is *safe* if *all* legal schedules of T w.r.t $L(T)$ are (conflict-) serializable. As an illustration, since the schedule from Example 4.39 is not in CSR, the locking it is based on, i.e.,

$$l_1(x)w_1(x)l_1(y)w_1(y)u_1(y)l_1(z)w_1(x)u_1(x)w_1(z)u_1(z)$$
$$l_2(z)w_2(z)l_2(y)u_2(z)l_2(x)w_2(y)w_2(x)u_2(x)w_2(y)u_2(y)$$

is not safe.

The interest in efficient algorithms for deciding whether a given locking $L(T)$ is safe can be motivated as follows: Suppose a scheduler S is given a set T of transactions; from T, scheduler S produces a locking $L(T)$. If $L(T)$ was known to be safe, S could output *any* schedule satisfying the simple syntactic condition of legality, since the safety of $L(T)$ guarantees the CSR membership of any legal schedule. Thus, if all transactions were completely given in advance (in the form of partially ordered read and write actions on data items), the scheduler could be based on generating appropriate locks for conflict regions among transaction pairs, without requiring explicit locks on data items at run time. In terms of the allowed concurrency, this method could possibly outperform the more traditional data-oriented locking protocols. However, the method would require a potentially expensive pre-analysis of the transactions and a "compile-time" test for safety along with the appropriate lock generation. The latter could be based on exploiting the geometric interpretation as follows:

pre-analysis locking (PAL)

A legal schedule avoids conflict regions. However, not every legal schedule is serializable, since it could avoid, yet separate such regions. On the other hand, the previous theorem says that a legal schedule is in CSR iff its curve does not separate conflict regions. Moreover, it can be shown that a locking is safe iff the geometrical "closure" of the conflict regions is connected. This opens a door for the use of computational geometry in the context of database transactions, further details of which can be found in the references cited at the end of the chapter.

4.4 Non-Locking Schedulers

In this section we present various alternatives to locking schedulers; the protocols discussed next can all guarantee the safety of their output *without* using locks. Their use in commercial database systems is limited, but they may be suited for *distributed* systems and can also be used in *hybrid* protocols (see Section 4.5 below).

4.4.1 Timestamp Ordering

A first approach to get rid of locks is to use *timestamps*. Generally, these are values from a totally ordered domain; for simplicity, we will use natural numbers. The transaction manager assigns to each transaction t_i a unique timestamp $ts(t_i)$, e.g., by retrieving a clock value at the beginning of t_i or by maintaining a counter that is incremented every time a new timestamp is needed. The timestamp of a transaction is inherited by every operation of that transaction. A scheduler based on the use of timestamps now has to order conflicting operations based on their timestamps; therefore, it is said to operate according to a *timestamp-ordering* (TO) protocol. The central "TO rule" is as follows:

timestamp ordering rule (TO rule)

If $p_i(x)$ and $q_j(x)$, $i \neq j$, are operations in conflict, the following has to hold:
$p_i(x)$ is executed *before* $q_j(x)$ iff $ts(t_i) < ts(t_j)$

Theorem 4.41

$Gen(\text{TO}) \subseteq \text{CSR}$

Proof. Let s be a history generated by the TO protocol, and let (t_i, t_j) be an edge in the conflict graph G of s. Then there are operations $p_i(x)$ and $q_j(x)$ in conflict s.t. $p_i(x) <_s q_j(x)$. By the TO rule, this implies $ts(t_i) < ts(t_j)$.

Now if a transaction t_k is involved in a cycle of G, by induction we find that $ts(t_k) < ts(t_k)$, a contradiction. Thus, G is acyclic, which means that $s \in \text{CSR}$. □

We next briefly discuss an approach for "implementing" the TO rule. A simple, optimistic implementation is to transfer each operation submitted by the TM to the data manager for execution right away, with the only exception of those that arrive "too late": Operation $p_i(x)$ is *too late* if it arrives *after* the scheduler has already output a conflicting operation $q_j(x)$ s.t. $i \neq j$ and $ts(t_j) > ts(t_i)$.

If p_i is too late it can no longer be output without violating the TO rule. Thus, $p_i(x)$ must be rejected, which implies that t_i has to be aborted. It can then be restarted later, at which point it will receive a timestamp with a larger value so that some, or ideally all, of the previous conflicts will now appear in the proper order.

In order to determine whether an operation has arrived too late, a *basic* TO (BTO) scheduler has to record the following timestamps for every data item x:

1. max–r–scheduled(x): the value of the largest timestamp of a read operation on x already sent to the data manager;
2. max–w–scheduled(x): the value of the largest timestamp of a write operation on x already sent to the data manager.

basic timestamp ordering (BTO)

When some operation $p_i(x)$ arrives, then $ts(t_i)$ is compared to max–q–scheduled(x) for each operation q that is in conflict with p. If $ts(t_i) <$ max–q–scheduled(x) holds, $p_i(x)$ is rejected, since it has arrived too late. Otherwise it is sent to the data manager, and max–p–scheduled(x) is updated to ts(t_i) if $ts(t_i) >$ max–p–scheduled(x).

The behavior of the BTO protocol is illustrated in Figure 4.17, where transactions t_2 and t_1 are eventually aborted because they issue operations "too late" relative to their timestamps (i.e., the begin of these transactions).

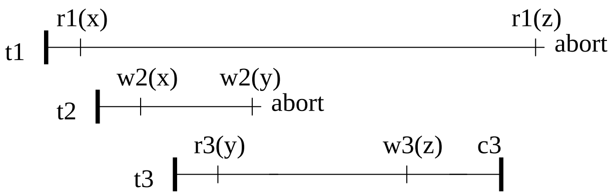

Figure 4.17: Example execution under the BTO protocol.

An important point here is that the scheduler has to make sure that the data manager executes all operations in the order the scheduler sends them. Note that 2PL does enforce this automatically, since an operation is not output as long as conflicting operations already scheduled still hold their locks. A BTO scheduler can send an operation $p_i(x)$, ready for being scheduled, to the data manager only if every conflicting and previously sent $q_j(x)$ has been executed.

The data manager hence has to *acknowledge* the execution of every operation, and the scheduler has to await that acknowledgement for operations in conflict that are to be scheduled in timestamp order; such an alternating way of operation is commonly called a *handshake*.

If a BTO scheduler receives new operations in an order that is vastly different from the timestamp order, it is likely to reject many of them, and to abort the respective transactions. This suggests the use of more conservative variants, such as one in which operations could be blocked artificially: When $o_i(x)$ is received, a conflicting operation with a smaller timestamp could be received later. Thus, if $o_i(x)$ is retained, such conflicting operating hopefully arrive "on time." Clearly, the choice of the time slice for which an operation gets blocked artificially can be a critical performance factor.

4.4.2 Serialization-Graph Testing

The next class of scheduling protocols we are going to present follows immediately from the serializability theorem from Chapter 3, which characterizes conflict serializability via the absence of cycles in the conflict graph. A *serialization-graph tester* (*SGT protocol* for short) maintains a conflict graph in which nodes and edges are added or removed dynamically depending on the operations that arrive at the scheduler. The CSR property of the output is preserved by making sure that this graph remains acyclic at any time.

serialization graph testing (SGT scheduler)

In more detail, an SGT protocol works as follows: Whenever a new operation $p_i(x)$ arrives from the transaction manager, the scheduler

1. creates a new node for transaction t_i in the current graph G if $p_i(x)$ is the first operation it sees from t_i;
2. inserts edges of the form (t_j, t_i) into G for each operation $q_j(x)$ which is in conflict with $p_i(x)$, $i \neq j$, and which has been output previously; now two cases can arise:
   (a) The resulting graph G is cyclic. If $p_i(x)$ was executed, the resulting schedule would no longer be serializable. Thus, $p_i(x)$ is rejected and t_i aborted, and the node for t_i and all its incident edges are removed from G.
   (b) G is (still) acyclic. Then $p_i(x)$ can be output, i.e., added to the schedule already output, and the tentatively updated graph is kept as the new current one.

In an implementation, it would, in analogy to the BTO protocol, be essential to output a $p_i(x)$ (case 2b) only if all conflicting operations previously output have been acknowledged by the data manager. Otherwise, $p_i(x)$ needs to be blocked until these acknowledgements have arrived.

Theorem 4.42

$Gen(\mathrm{SGT}) = \mathrm{CSR}$

For the proof, note that "$\subseteq$" is trivial. The inverse inclusion can be proven by induction on the length of a schedule $s \in \mathrm{CSR}$.

The description of SGT given above suggests that the conflict graph maintained by the scheduler grows indefinitely. However, from time to time information can also be removed from that graph. Unfortunately, the most obvious idea that a transaction can be removed from the graph when it commits does not work correctly. To see why, consider the following example:

$$s = w_1(x)w_2(x)w_2(y)c_2 \ldots w_1(y)$$

This schedule is clearly not in CSR. If we remove the node for t_2 upon the transaction's commit, c_2, then it would later be impossible to recognize the serialization-graph cycle when transaction t_1 issues its $w_1(y)$ action. Note that the latter could possibly be a very long time after the commit of t_2. So this very simple idea is not feasible. To obtain more insight when nodes can safely be removed from the graph, let us look at another example.

Example 4.43

Consider

$$s = r_{k+1}(x)\,\underbrace{w_1(x)w_1(y_1)c_1}_{\text{all of } t_1}\,\underbrace{w_2(x)w_2(y_2)c_2}_{\text{all of } t_2}\ldots\underbrace{w_k(x)w_k(y_k)c_k}_{\text{all of } t_k}\ldots$$

The serialization graph of s is shown in Figure 4.18. Now suppose the next operation the scheduler receives is $w_{k+1}(z)$. According to what we stated above, $w_{k+1}(z)$ can be output if $z \notin \{x, y_1, \ldots, y_k\}$ holds. In order to test whether this is the case the information on the t_i, $1 \leq i \leq k$, must still be available, *although* these transactions are already committed. □

As the example suggests, an SGT scheduler can remove a transaction t_i from its conflict graph provided t_i will not get involved in a cycle anymore — not even at an arbitrary point in the future. Now a node on a cycle has at least one incoming and one outgoing edge, and as the example shows it may happen that *outgoing* edges are added to a node even after the respective transaction has terminated already. On the other hand, for a finished transaction there will be no more *incoming* edges added to its node; if t_i does not have incoming edges, it cannot be involved in a cycle. So

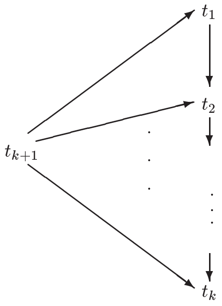

Figure 4.18: Serialization graph for Example 4.44.

the rule for eliminating a transaction from the conflict graph of an SGT scheduler is simply the following: *If t_i is finished and is a source (a node with outgoing edges only) in G, then it can be removed from G.*

We finally mention that an SGT scheduler, although attractive from a theoretical point of view since it is a simple approach to generating *all* of CSR, is not practical to implement. For example, the (worst-case) *space* required to maintain the serialization graph grows with the square of the number of relevant transactions, some of which are not even active any more. We even need to keep around the information about the readsets and writesets of such already committed transactions. In addition, and even more importantly, the testing of cycles in the graph, albeit only linear in the number of edges (i.e., quadratic in the number of nodes in the worst case), may be unacceptable at run time simply because it would be invoked very frequently. After all, concurrency control measures are part of a data server's innermost loops, and their run time overhead is thus an extremely critical issue.

4.4.3 Optimistic Protocols

So far we have tacitly assumed that schedulers run in a transactional environment where conflicts are *frequent* events, so that measures must always be available for coping with them. Indeed, we have assumed that a scheduler needs to decide on the spot for each newly arriving step whether to execute, to reject, or to block it. We call schedulers based on this assumption *pessimistic* schedulers. A different class of schedulers results from the opposite assumption that conflicts are *rare* events.

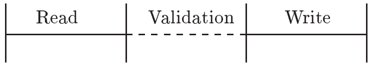

Figure 4.19: The 3 phases of a transaction under an optimistic scheduler.

Consider, for example, a product catalog application where 99 % of the transactions just read price information and descriptions of products. From time to time, prices are updated or new products are added, but this occurs with a very low frequency compared to their read events. If our application scenario were like that, a 2PL scheduler that keeps acquiring locks would be too restrictive, since it would waste a considerable amount of time on the management of locks instead of reading data items.

validation protocols

Under the assumption that conflicts between transactions and their operations are rare, we are able to design *optimistic* schedulers which essentially let newly arriving operations simply pass, but test from time to time whether the schedule produced so far is still a serializable one. In other words, they have to *validate* their output occasionally, which is why they are also known as *validation protocols* (sometimes also called *certifiers*). Clearly, we do not interrupt the production of a schedule in order to validate what and how the scheduler has been doing so far, but we build this validation into the transactions themselves. To this end, it makes sense to perceive a transaction's execution as taking place in three phases (cf. Figure 4.19):

1. *Read phase*: The transaction is executed, but with all writes applied to a workspace that is private to the transaction only (not to the database). So the private "versions" of data items written by a transaction are not visible to other transactions (yet).

   Notice that at the end of the read phase of a transaction t, its *read set* RS(t) as well as its *write set* WS(t) are known.

2. *Validation phase*: A transaction that is ready to commit is validated, i.e., the scheduler tests whether its execution has been "correct" in the sense of conflict serializability, and whether the transaction's result can be copied into the database. If this is not the case, the transaction is aborted; otherwise, the next phase is entered.

3. *Write phase*: The workspace contents are transferred into the database to conclude the transaction's commit.

We here assume as a simplification that Phases 2 and 3 are executed indivisibly as a non-interruptible "critical section", meaning that all other transactions are suspended during such a critical section. We correspondingly call this combined phase the *val-write* phase. Note that the indivisibility of this phase may be quite problematic in a real implementation. If the write phase merely needs to modify a few entries in a page allocation table, for example, then the critical section is not really harmful. However, if the write phase needs to copy records from a working space into disk pages, suspending all concurrent transactions for such an extended time period would inevitably result in significant performance losses. If, on the other hand, the indivisibility condition were to be relaxed, one would face the problem that the write phase of one transaction would be concurrent with the read phases of others. With the validation preceding the write phase, it is clear that additional measures would now be needed to ensure that these writes do not lead to conflict cycles. One possibility would be to introduce some limited form of exclusive locking during the write phase, but this would require read locks during the read phases as well and would obviously render the whole family of optimistic protocols much less attractive. To keep things simple, we thus assume indivisible val-write phases and leave the further discussion of relaxations as an exercise (see Exercise 18).

Since we are still interested in conflict serializable schedules or, equivalently, schedules with an acyclic conflict graph, we can reduce the goal of transaction validation to keeping an acyclic graph acyclic after the insertion of a new node. The following lemma will turn out to be a key observation for what follows:

Lemma 4.44

Let G be a DAG. If a new node is added to G in such a way that no edges start from the new node, then the resulting graph is still a DAG.

The simple reason why the lemma is true is that if a new node is added to an acyclic graph and the graph thereby becomes cyclic, then the new node must be involved in the cycle. But in order to be on a cycle, there would have to be at least one edge going into the node, and at least one coming out.

We will use Lemma 4.44 as a guideline for the validation of transactions under an optimistic protocol. Indeed, we will establish "validation rules" by which acyclic serialization graphs will be kept acyclic if newly validated transactions are added.

We will essentially use two different types of validation approaches, based on the available options:

backward-oriented and forward-oriented validation (BOCC and FOCC)

- Under *backward-oriented optimistic concurrency control* (BOCC), a transaction under validation executes a conflict test against all those transactions that are already committed.

- Under *forward-oriented optimistic concurrency control* (FOCC), a transaction is validated against all transactions that run in parallel, but which are still in their read phase.

Let us look at the two options in turn. Under BOCC validation, a transaction t_j is positively validated (or "accepted") if one of the following holds for each transaction t_i that are already committed (and hence have previously been accepted):

- t_i has ended before t_j has started (which implies that all of t_i has been executed before all of t_j); in this case, if there is a conflict between operations from t_j and t_i, the corresponding conflict edge will be of the form (t_i, t_j) and not vice versa.
- Otherwise, $\mathrm{RS}(t_j) \cap \mathrm{WS}(t_i) = \emptyset$, and the val-write phase of t_i has ended prior to the val-write phase of t_j (assuming a "critical section" as discussed earlier). Thus, t_j has not had a chance to read from t_i, and again if there are conflicts at all, they give rise to an edge from t_i to t_j, but one in the opposite direction is impossible.

So we may conclude, using Lemma 4.44, the following:

Theorem 4.45

BOCC validation produces acyclic conflict graphs only, i.e. $Gen(BOCC) \subseteq CSR$.

If the second case above fails, i.e., if $\mathrm{RS}(t_j) \cap \mathrm{WS}(t_i) \neq \emptyset$, it may be the case that t_j still did not read from t_i, since the schedule could look like this:

$$\ldots r_j(x) \ldots w_i(x) \ldots\ldots \text{validate}(t_j)$$

Following BOCC, we would abort t_j during its validation, although it has been clear much earlier (at the time t_i wrote x) that it would eventually fail. To accommodate such situations, the alternative to BOCC is FOCC, which validates a transaction t_j against all concurrent transactions t_i that are still reading. Let $\mathrm{RS}^n(t_i)$ denote the read set of t_i at some time n. Then t_j is accepted under FOCC at that time n if the following holds for all transactions t_i that are still reading at time n:

$$\mathrm{WS}(t_j) \cap \mathrm{RS}^n(t_i) = \emptyset$$

Notice that this condition is immediately satisfied if t_j is read-only, a particularly nice property. The following can now be verified:

Theorem 4.46

FOCC validation produces acyclic conflict graphs only, i.e., $Gen(FOCC) \subseteq CSR$.

In order to prove this theorem, the idea is again to use Lemma 4.44 for showing that, under the given prerequisites, an insertion of t_j into a conflict graph that has so far been acyclic does not destroy this property. Details are left to the reader (see Exercise 13). Note that FOCC even guarantees that all output schedules are commit-order-preserving conflict serializable (COCSR). This is an immediate consequence of the fact that transactions validate against ongoing transactions, and the validation order corresponds to the commit order, with a critical section encompassing both the validation and the write phase.

FOCC ensures COCSR

As example executions under the BOCC and FOCC protocols, consider Figures 4.20 and 4.21. Both figures show the read, validation, and write phases of several concurrent transactions with time proceeding from left to right.

In the BOCC scenario of Figure 4.20, transaction t_1 is trivially validated as there is no concurrent, previously committed transaction. t_2 then is successfully validated against t_1 as its read set, $RS(t_2) = \{y, z\}$, does not overlap with the write set of t_1, $WS(t_1) = \{x\}$. Next, t_3 must validate against both t_1 and t_2; its read set, $RS(t_3) = \{x, y\}$, is disjoint with the write set of t_2 but has data item x in common with the write set of t_1, hence t_3 must be aborted. Finally, t_4 has started after the commit of t_1, so it must validate only against the committed, concurrent transaction t_2. Its read set, $RS(t_4) = \{x\}$ does not overlap with the write set of t_2; thus, t_4 is successfully validated.

In the (different) FOCC scenario of Figure 4.21, transaction t_1 must validate against the concurrent transaction t_2 by comparing its write set, $WS(t_1) = \{x\}$, with the *current* read set of t_2, $RS(t_2) = \{y\}$, as of the time of t_1's validation phase. Later, when t_2 initiates its validation, it detects that its write set, $WS(t_2) = \{z\}$ is disjoint with the *current* read sets $RS(t_4) = \{x, y\}$ and $RS(t_5) = \{x\}$, but does overlap with the current read set of t_3 which is $RS(t_3) = \{z\}$. Now, unlike under a BOCC protocol, FOCC allows t_2 to choose among several options. In the specific situation, t_2 could either abort itself, or it could enforce the abort of the "troublesome" concurrent transaction t_3. Since t_2 has been running for quite a while whereas t_3 has just begun, we assume that t_3 will be aborted to minimize the amount of wasted work. A third option turns out to be profitable for the validation of transaction t_4, as that transaction detects that its write set, $WS(t_4) = \{y\}$, overlaps with the current read set of the concurrent transaction t_5, $RS(t_5) = \{x, y\}$. In addition to either aborting t_4 or t_5, the third option for t_4 is to wait (indicated by the dashed line in Figure 4.21) and let the conflicting transaction t_5 validate first, in the optimistic anticipation that t_5 would not lead to a (potential) conflict cycle and could thus successfully validate. Indeed, t_5 turns out to be a read-only transaction, i.e.,

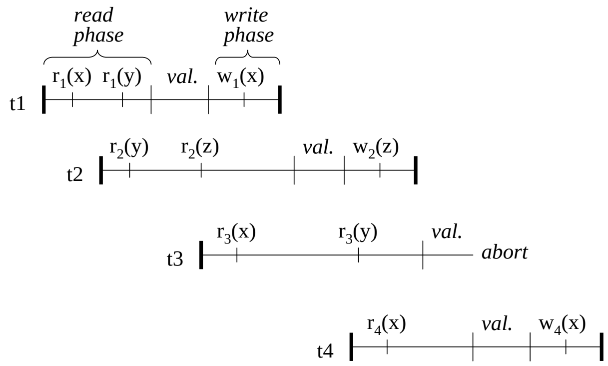

Figure 4.20: Example execution under the BOCC protocol.

its write set is empty, and therefore trivially validates. Subsequently, the validation of t_4 is resumed and now succeeds as there are no more concurrent transactions. As the example shows, a significant practical advantage of FOCC over BOCC lies in the flexibility of choosing abort victims or even avoiding some aborts by forcing transactions to wait in their validation phase.

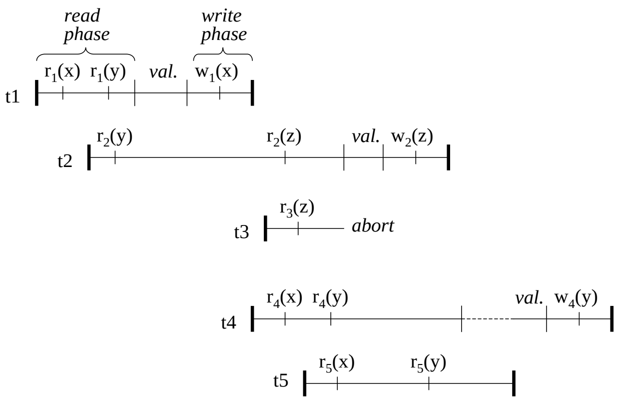

Figure 4.21: Example execution under the FOCC protocol.

4.5 Hybrid Protocols

Figure 4.22 summarizes the protocols we have presented so far. The key distinction between the two major classes, pessimistic and optimistic protocols, is that the pessimistic methods conservatively perform tests that guarantee an acyclic conflict graph as early as possible (e.g., lock conflict tests, or searching for cycles in the conflict graph whenever a new edge is added), whereas optimistic methods postpone this testing until a transaction is ready to commit.

The number of protocols shown in this figure could even be squared by decomposing the concurrency control problem into the following two subproblems, which could then be solved individually by possibly distinct protocols:

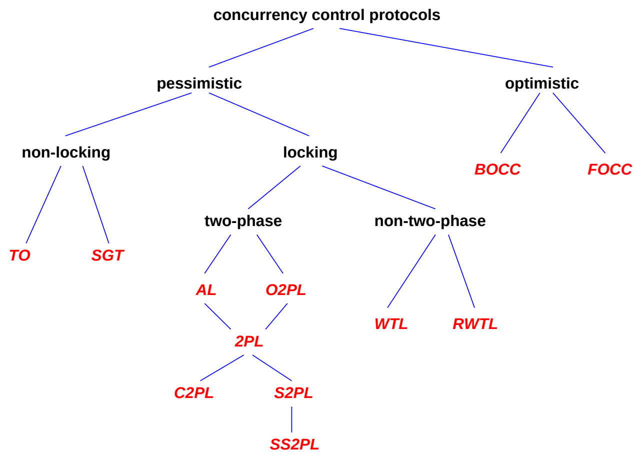

Figure 4.22: Summary of concurrency control protocols.

idea of hybrid protocols

1. rw (and wr) synchronization: read operations are synchronized against write operations or vice versa;
2. ww synchronization: write operations are synchronized against other write operations, but not against reads.

If these synchronization tasks are distinguished, a scheduler can be thought of as consisting of two components, one for each of the respective synchronization tasks. Since the two components need proper integration (as demonstrated by way of examples shortly), such a scheduler is called a *hybrid* scheduler.

For correctly integrating possibly distinct protocols for the two subtasks, the definition of conflicts known from the previous chapter needs to be modified in order to account for the intended synchronization:

for rw (and wr) synchronization: Two operations are in conflict if they access the same data item, and one is an r step, the other a w step;

for ww synchronization: two operations are in conflict if they access the same data item, and *both* are write operations.

Example 4.47

Consider a hybrid scheduler that uses 2PL for rw synchronization and SGT for ww synchronization. The 2PL scheduler will now block an operation $w_i(x)$ only if some t_j, $j \neq i$, has a read lock on x. Thus, if several transactions hold write locks on the same data item, the 2PL scheduler does not care, since this is now the task of the other scheduler. Conversely, for the SGT scheduler an edge (t_i, t_j) in the conflict graph only indicates that there is a ww conflict between t_i and t_j. □

To prove a hybrid scheduler correct, an obvious approach is to consider the two conflict graphs $G_{rw}(s)$ and $G_{ww}(s)$ of an output s we now have and which represent *distinct* conflicts, whereas the single conflict graph we have so far been considering abstracted from the particular nature of conflicts. Since all protocols that we have discussed are CSR-safe, we only need to make sure that the two graphs we now have represent *compatible* serialization orders, i.e., if t_i occurs "before" t_j in one of the graphs, then the same holds for the other. In other words, the *union* of the two graphs now has to be acyclic. This may require additional preparations, as we now see.

SS2PL + TO

Consider a hybrid scheduler that uses SS2PL for rw (and wr) synchronization, and that wants to use TO for ww synchronization. Observe that if a TO scheduler receives a step $w_i(x)$ *after* a step $w_j(x)$, where $ts(t_i) < ts(t_j)$, then $w_i(x)$ should be rejected, since the two are out of timestamp order. However, since the scheduler is interested in ww synchronization *only*, a rejection is not necessary. Indeed, if $w_i(x)$ is simply ignored, the effect will be the same, namely that $w_j(x)$ writes x last. Thus, executing a sequence of write operations in timestamp order produces the same result as executing exclusively the write operation that carries the largest timestamp. This observation leads to a ww synchronization approach known as *Thomas' Write Rule (TWR)*: Let t_j be the transaction that has written x and that has the largest timestamp prior to the arrival of $w_i(x)$. If $ts(t_i) > ts(t_j)$, $w_i(x)$ is processed as usual; otherwise, it is ignored, and the execution of t_i just continues. Notice that TWR applies to pairs of write steps only.

Thomas' Write Rule (TWR)

Example 4.48

Consider $w_1(x)r_2(y)w_2(x)w_2(y)c_2w_1(y)c_1$, where $ts(t_1) < ts(t_2)$. $w_1(y)$ is ignored (i.e., not executed at all) by the TWR. The resulting output schedule (with $w_1(y)$ omitted) is conflict serializable.
On the other hand, consider the following variation of the example:

$$w_1(x)r_2(y)w_2(x)w_2(y)c_2r_1(y)w_1(y)c_1.$$

The hybrid SS2PL / TWR protocol would still allow this schedule, and again the TWR would suppress the $w_1(y)$ step of t_1. However, the $r_1(y)$ step would be executed,

leading to a conflict cycle between t_1 and t_2. So although each of the two protocols works correctly as far as its limited perspective is concerned, the combination does not properly guarantee conflict serializability. □

As the second example indicates, an integration of SS2PL and TO or TWR has to be done cautiously. If the protocols are used as described above, the following holds for every output s: (i) $G_{rw}(s)$ is acyclic (due to the use of SS2PL), and (ii) $G_{ww}(s)$ is acyclic (due to TWR). To guarantee that $G(s) = G_{rw}(s) \cup G_{ww}(s)$ is acyclic as well, it apparently suffices to state the following:

(1) $(t_i, t_j) \in G_{rw}(s) \Rightarrow ts(t_i) < ts(t_j)$

Since $G_{ww}(s)$ already satisfies the corresponding condition, we conclude that $G(s)$ is acyclic, provided (1) is holds. In order to guarantee (1) we note that if $(t_i, t_j) \in G_{rw}(s)$, then t_j cannot terminate before t_i releases a lock that t_j needs. Under SS2PL, every transaction holds its locks until commit, so $(t_i, t_j) \in G_{rw}(s)$ implies that t_i ends successfully before t_j ends. Thus, condition (1) can be met by

(2) if t_i commits before t_j does, then $ts(t_i) < ts(t_j)$.

For assuring (2) the scheduler needs to block the assignment of a timestamp to t_i until it reaches it commit point. Such commit-order-preserving timestamping entails deferring also the setting of timestamps in the written data items. This can be accomplished with a workspace concept, in analogy to the deferred-write approach of optimistic protocols. All writes are performed in a transaction's private workspace, and transferred into the database only upon the transaction's commit.

hybrid protocols for partitioned data

Another, practically intriguing, form of hybrid protocols is based on partitioning the set of data items, D, into disjoint subsets $D_1, D_2, \ldots, D_n$ (with $n \geq 2$, $D_i \cap D_j = \emptyset$ for $i \neq j$, and $\bigcup_{i=1}^{n} D_i = D$). Then, different concurrency control protocols could be used for each of the n partitions, provided, and this may be the crux, that the union of the conflict graphs over all partitions remains acyclic at each point. Such a setting naturally arises in a distributed federation of heterogeneous data servers. We will discuss this case in detail in the book's Part IV.

Another application case for hybrid protocols within a centralized data server is to intentionally classify data items into disjoint classes according to their specific access characteristics. For example, a data server may manage a small set of very frequently modified data items, so-called "hot spots", that constitute subset D_1, a large set of less frequently modified data items that form subset D_2, and a remaining subset D_3 of data items that are never updated online. Such a setting suggests using the protocol with the highest possible concurrency for D_1 regardless of its overhead (as D_1 is small), a low-overhead concurrency control protocol for the bulk of data items D_2, and no concurrency control at all for D_3. For example, we could combine SGT for D_1 with FOCC for D_2. However, we need to ensure that this combination preserves the acyclicity of the global conflict graph that is formed by the union of the two conflict graphs for D_1 and D_2. This is all but trivial for the two considered protocols, and may in fact require some additional cycle testing across both partitions. Such global overhead may, however, render the use of the "light-weight" FOCC protocol pointless, as the overhead of cycle testing would eventually be incurred for all of $D_1 \cup D_2$. A less adventurous combination would thus be using SS2PL for D_1 and *FOCC* for D_2. In this case, the fact that both protocols are commit-order-preserving conflict serializable (COCSR) eliminates the need for global checks. More considerations along these lines will be presented in the book's Part IV in the context of distributed systems.

4.6 Lessons Learned

In this chapter, we have seen a variety of concurrency control protocols for the page model of transactions. There are two broad categories: locking schedulers, which require the explicit locking and unlocking of data items for the purpose of read or write access, and non-locking schedulers that can do without locks. Moreover, locking schedulers can be two-phase, meaning that, on a per-transaction basis, locks precede unlocks, or they can be non-two-phase, provided access ordering is controlled in some other way.

We will see in subsequent chapters that, as far as concurrency control is concerned, these are the fundamental ideas around. Indeed, we will apply the ideas underlying the protocols we have seen in this chapter to various other (often more general) contexts as well. For example, an SGT protocol is immediately available, once a serializability theorem has been proven that relates conflict-based serializability to the acyclicity of a suitably defined graph.

We will also see that, although there exists a variety of algorithmic approaches to concurrency control, the strict variant of 2PL, S2PL, is actually *the* protocol around and hence the one most commonly found in commercial transactional servers, for its versatility as well as robustness and the observation that it outperforms the other protocols in most practical situations.

Exercises

1. For each of the following (input) schedules, show the output produced by 2PL, S2PL, SS2PL, O2PL, BTO, and SGT:

$s_1 = w_1(x) r_2(y) r_1(x) c_1 r_2(x) w_2(y) c_2$
$s_2 = r_1(x) r_2(x) w_3(x) w_4(x) w_1(x) c_1 w_2(x) c_2 c_3 c_4$

2. A *lock point* of a transaction t denotes a point in time at which t has obtained all locks it needs, but has not yet released any. Show that for each history s produced by a 2PL scheduler there exists a conflict equivalent serial history s' in which all transactions occur in the same order of lock points as in s.

3. i Under the 2PL protocol it is possible for transactions to "starve" in the following sense: A transaction gets involved in a deadlock, is chosen and the victim and aborted. After a restart, it gets again involved in a deadlock, is chosen as the victim and aborted, and so on. Provide a concrete example for such a situation, and describe how 2PL could be extended in order to avoid starvation of transactions.

4. Describe the wait-for graphs resulting from the use of 2PL for each of the histories in Exercise 1.

5. Show that the wait-die and wound-wait approaches to deadlock prevention both guarantee an acyclic WFG at any point in time.

6. Consider $s = w_1(x) w_1(y) r_2(u) w_2(x) r_2(y) w_2(y) c_2 w_1(z) c_1$. Extend this history by syntactically valid lock and unlock operations, and represent the result as a curve in the plane. Then determine the conflict regions and decide whether DT(s) is in CSR.

7. Consider the schedule whose geometrical interpretation is shown in Figure 4.16. Exhibit a point in the plane of that figure which represents a deadlock.

8. Consider the following input schedules to the O2PL protocol (i.e., the ordered-sharing generalization of 2PL, based on the lock compatibility table LT_8):
(a) $s_1 = w_1(x) r_2(x) c_2 r_3(y) c_3 w_1(y) c_1$
(b) $s_2 = w_1(x) r_2(x) r_3(y) c_3 r_2(z) c_2 w_1(y) c_1$
Which are the corresponding output schedules produced by O2PL? For each of the two schedules, give the details about when locks are requested, granted, attempted to be released, and eventually released.

9. Show that O2PL is susceptible to deadlocks (i.e., it is not deadlock-free).

10. Show that *Gen*(AL) $\subseteq$ CSR, i.e., complete the proof of Theorem 4.29.

11. Investigate the relationship between *Gen*(BTO) and *Gen*(2PL). Is one of the two more powerful than the other?

12. Consider the following condition for removing nodes (and edges) from the serialization graph in the SGT protocol: remove t_i when it is finished and none of the transactions that were active at the commit or abort of t_i are active anymore. Show that this condition, albeit seemingly natural, would lead to incorrect behavior of the SGT protocol.

13. Prove Theorem 4.46, thereby showing that FOCC is a CSR-safe protocol.

14. Consider the following alternative variant of a BOCC protocol. Transaction t_j is successfully validated if one of the two following conditions holds for all other transactions t_i that are already successfully validated:

    (i) t_i terminates before t_j starts its read phase,

    (ii $RS(t_j) \cap WS(t_i) = \emptyset$ and t_i finishes its read phase before t_j enters its write phase (i.e., relaxing the original BOCC condition that t_i must finish its write phase before t_j initiates its validation).

    Construct an example of a non-serializable schedule that would be allowed under this BOCC variant. On the other hand, prove that this variant does guarantee conflict serializability under the additional constraint that $WS(t_k) \subseteq RS(t_k)$ holds for all transactions t_k.

15. Consider an alternative variant of BOCC where a transaction validates by comparing its write set against the read sets of concurrent, previously validated transactions. Would such a variant work correctly, i.e., ensure that all output schedules are CSR?

16. Consider the following input schedule of three concurrent transactions:

    $$r_1(x) r_2(x) r_1(y) r_3(x) w_1(x) w_1(y) c_1 r_2(y) r_3(z) w_3(z) c_3 r_2(z) c2$$

    Which are the resulting output schedules under the BOCC and FOCC protocols? Remember that write steps are actually performed on private workspaces, the commit requests initiate the validation, and the write steps are performed on the shared database only after a successful validation.

17. Construct an example execution that demonstrates that FOCC may produce non-serializable schedules if the critical-section condition for the val-write phases were dropped.

18. Develop variants of the BOCC and FOCC protocols that do no longer need the critical section for the val-write phases of transactions. Discuss the pros and cons of such a relaxation, in comparison to locking protocols.

Bibliographical and Other Remarks

The first major comprehensive presentation of concurrency-control protocols, which has also influenced the presentation in this chapter, was given by Bernstein et al. (1987). Theoretical questions, such as the scheduling power of protocols or the geometry of locking, are discussed further by Papadimitriou (1986). Implementation issues are studied by Bernstein and Newcomer (1997), Claybrook (1992), and in particular by Gray and Reuter (1993).

The 2PL protocol goes back to Eswaran et al. (1976). Its generalization O2PL, which can generate exactly OCSR as well as the idea of using ordered sharing as a third locking mode, are due to Agrawal and El Abbadi (1990, 1995); a performance study of locking based on ordered sharing appears in Agrawal et al. (1994). Agrawal et al. (1987) provide a survey of techniques for handling deadlocks; the specific wound-wait and wait-die techniques are from Rosenkrantz, Stearns, and Lewis (1978), and the running priority technique is from Franaszek and Robinson (1985). Implementation issues of deadlock detection can be found in Beeri and Obermarck (1981) and Jiang (1988). Altruistic locking is from Salem et al. (1994). Early studies of giving up on the two-phase property, and of replacing it by tree or DAG structures are by Kedem and Silberschatz (1980, 1983), Silberschatz and Kedem (1980, 1982), Buckley and Silberschatz (1984, 1985), Dasgupta and Kedem (1983, 1990), and Mohan et al. (1984, 1985). Croker and Maier (1986) present a dynamic tree-locking protocol.

Further details on the geometrical interpretation of locking can be found in Lausen et al. (1984, 1990) or in Papadimitriou (1982, 1983, 1986); proofs of theorems 4.38 and 4.40 can be found in Papadimitriou (1986). Lausen et al. (1986) study the exploitation of the geometrical interpretation for a form of pre-analysis locking. Kung and Papadimitriou (1983) show that a two-phase locking scheduler is "optimal" among schedulers that only use syntactic information. Wolfson (1986, 1987) uses the geometric techniques for studying algorithms that allow early unlocking of database items. Yannakakis (1982, 1984) studies the safety of locking policies along these lines.

The TO procotol is from Thomas (1979); see also Bernstein and Shipman (1980) as well as Bernstein et al. (1980), or Leu and Bhargava (1987). Serialization-graph testing is from Casanova (1981); the basic idea appeared in Schlageter (1978) as well. The rule for deleting completed transaction from a serialization graph is from Hadzilacos and Yannakakis (1989). The concepts of optimistic concurrency control algorithms were developed by Kung and Robinson (1981) as well as Härder (1984). The concept of private workspaces and deferred writes has also been studied with regard to other concurrency control protocols by Gold et al. (1985, 1986). Mohan (1992) provides a rather critical account of implementation and performance issues for optimistic concurrency control.

The idea of integrating multiple protocols by taking the various forms of synchronization problems apart is from Bernstein et al. (1983). Various approaches along these lines have been proposed, for example, by Lausen (1982), Boral and Gold (1984), and Yu and Dias (1992). Dynamically switching from one type of protocol to another for adaptation to evolving workloads has been proposed by Bhargava and Riedl (1989). Thomas' write rule is due to Thomas (1979).

A comprehensive experimental study on the performance benefits and drawbacks of the various concurrency control protocols under different workload settings is provided by Agrawal, Carey, and Livny (1987), focusing mostly on optimistic protocols versus two-phase locking.