Chapter 3 - Solutions to Exercises

Exercise 3.1 :

Consider the following histories:

$s = r_1(x)r_2(y)w_1(y)r_3(z)w_3(z)r_2(x)w_2(z)w_1(x)c_1c_2c_3$
$s' = r_3(z)w_3(z)r_2(y)r_2(x)w_2(z)r_1(x)w_1(y)w_1(x)c_3c_2c_1$

Compute $H[s]$ and $H[s']$ as well as the respective RF and LRF relations. Show the step graph of these histories.

a) Let us consider s first:
$H[s](x) = H_s(w_1(x)) = f_{1x}(H_s(r_1(x))) = f_{1x}(H_s(w_0(x))) = f_{1x}(f_{0x}())$
$H[s](y) = H_s(w_1(y)) = f_{1y}(H_s(r_1(x))) = f_{1y}(H_s(w_0(x))) = f_{1y}(f_{0x}())$
$H[s](z) = H_s(w_2(z)) = f_{2z}(H_s(r_2(x)), H_s(r_2(y))) = f_{2z}(H_s(w_0(x)), H_s(w_0(y))) = f_{2z}(f_{0x}(), f_{0y}())$
$RF(s) = \{(t_0, x, t_1), (t_0, y, t_2), (t_0, z, t_3), (t_0, x, t_2), (t_1, x, t_\infty), (t_1, y, t_\infty), (t_2, z, t_\infty)\}$
$LRF(s) = \{(t_0, x, t_1), (t_0, y, t_2), (t_0, x, t_2), (t_1, x, t_\infty), (t_1, y, t_\infty), (t_2, z, t_\infty)\}$

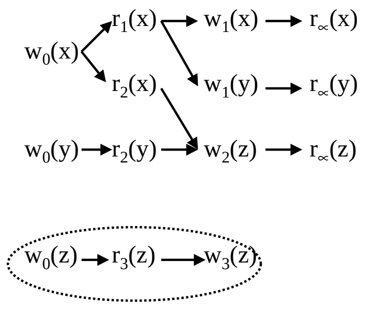


Figure 1: Step graph for schedule s.

The circled part in Figure 1 represents dead steps which are removed in the reduced step graph.

b) The solution for s' is analogous:
$H[s'](x) = H_{s'}(w_1(x)) = f_{1x}(H_{s'}(r_1(x))) = f_{1x}(H_{s'}(w_0(x))) = f_{1x}(f_{0x}())$
$H[s'](y) = H_{s'}(w_1(y)) = f_{1y}(H_{s'}(r_1(x))) = f_{1y}(H_{s'}(w_0(x))) = f_{1y}(f_{0x}())$
$H[s'](z) = H_{s'}(w_2(z)) = f_{2z}(H_{s'}(r_2(x)), H_{s'}(r_2(y))) = f_{2z}(H_{s'}(w_0(x)), H_{s'}(w_0(y))) = f_{2z}(f_{0x}(), f_{0y}())$
$RF(s') = \{(t_0, z, t_3), (t_0, y, t_2), (t_0, x, t_2), (t_0, x, t_1), (t_1, x, t_\infty), (t_1, y, t_\infty), (t_2, z, t_\infty)\}$
We observe:
$H[s'] = H[s]$, which means $s \approx_f s'$ according to Definition 3.6, which in turn implies $LRF(s) = LRF(s')$ according to Theorem 3.1. We also observe $RF(s') = RF(s)$ implying $s \approx_v s'$. As a consequence we obtain for the step graphs: $D(s) = D(s')$

Exercise 3.3 :

Suppose that in a given schedule the functions corresponding to the write steps represent increments of a counter, i.e., $f(x) = x + 1$. Compute the (Herbrand) semantics of the following schedules using this semantic information:

$s = r_3(z)r_1(y)w_3(z)w_1(y)r_1(x)r_2(y)w_2(y)w_1(x)r_2(x)w_2(x)c_1c_2c_3$
$s' = r_3(z)w_3(z)r_2(y)w_2(y)r_1(y)w_1(y)r_2(x)w_2(x)r_1(x)w_1(x)c_3c_2c_1$

a) Consider s:

$H[s](x) = H_s(w_2(x)) = f_{2x}(H_s(r_2(x))) = f_{2x}(H_s(w_1(x))) = 1 + f_{1x}(H_s(r_1(x))) = 1 + f_{1x}(H_s(w_0(x))) = 2 + f_{0x}()$

$H[s](y) = H_s(w_2(y)) = f_{2y}(H_s(r_2(y))) = f_{2y}(H_s(w_1(y))) = 1 + f_1 y(H_s(r_1(y))) = 1 + f_{1y}(H_s(w_0(y))) = 2 + f_{0y}()$

$H[s](z) = H_s(w_3(z)) = f_{3z}(H_s(r_3(z))) = f_{3z}(H_s(w_0(z))) = 1 + f_{0z}()$

b) Consider s':

$H[s'](x) = H_{s'}(w_1(x)) = f_{1x}(H_{s'}(r_1(x))) = f_{1x}(H_{s'}(w_2(x))) = 1 + f_{2x}(H_{s'}(r_2(x))) = 1 + f_{2x}(H_{s'}(w_0(x))) = 2 + f_{0x}()$

$H[s'](y) = H_{s'}(w_1(y)) = f_{1y}(H_{s'}(r_1(y))) = f_{1y}(H_{s'}(w_2(y))) = 1 + f_2 y(H_{s'}(r_2(y))) = 1 + f_{22y}(H_{s'}(w_0(y))) = 2 + f_{0y}()$

$H[s'](z) = H_{s'}(w_3(z)) = f_{3z}(H_{s'}(r_3(z))) = f_{3z}(H_{s'}(w_0(z))) = 1 + f_{0z}()$

So the two schedules s and s' are final state equivalent under the semantics where writer are known to be increments, however, they are not final state equivalent under the general Herbrand semantics.

Exercise 3.4 :

Consider the following history:

$s = r_1(x)r_3(x)w_3(y)w_2(x)r_4(y)c_2w_4(x)c_4r_5(x)c_3w_5(z)c_5w_1(z)c_1$

Into which of the classes FSR, VSR, CSR does this schedule fall?

In order to show that $s \in FSR$, it is sufficient to construct a serial schedule s' with $s' \approx_f s$. Consider

$$s' = t_5t_3t_1t_2t_4 = r_5(x)w_5(z)c_5r_3(x)w_3(y)c_3r_1(x)w_1(z)c_1w_2(x)c_2r_4(y)w_4(x)c_4,$$

For schedule s we have:

$RF(s) = \{(t_0, x, t_1), (t_0, x, t_3), (t_3, y, t_4), (t_4, x, t_5), (t_1, z, t_\infty), (t_4, x, t_\infty), (t_3, y, t_\infty)\}$
$LRF(s) = \{(t_0, x, t_1), (t_0, x, t_3), (t_3, y, t_4), (t_1, z, t_\infty), (t_4, x, t_\infty), (t_3, y, t_\infty)\} = LRF(s')$
Therefore $s \in FSR$ holds.

Assume $s \in VSR$, i.e., there is a serial schedule s_1 with $s \approx_v s_1$ and thus $RF(s) = RF(s_1)$. Then the following must hold for s_1:

- $t_2 <_{s_1} t_4$ for $(t_4, x, t_\infty) \in RF(s)$ and $w_2(x) \in op(s)$
- $t_4 <_{s_1} t_5$ for $(t_4, x, t_5) \in RF(s)$
- $t_5 <_{s_1} t_1$ for $(t_1, z, t_\infty) \in RF(s)$ and $w_5(z) \in op(s)$

Thus, we obtain $t_2 <_{s_1} t_1$ which contradicts $(t_0, x, t_1) \in RF(s)$. Therefore s is not in VSR. Clearly s is also not in CSR since $CSR \subset VSR$ according to Theorem 3.8. $s \notin CSR$ can also be verified by observing a cycle ($t_1 \rightarrow t_4 \rightarrow t_5 \rightarrow t_1$) in the conflict graph of s as you can see in Figure 2.

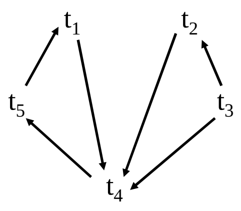


Figure 2: Conflict graph for schedule s.

Exercise 3.6 :

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

Show that $s \in FSR - VSR$.

a) We first show, that $s \in FSR$:
$RF(s) = \{(t_0, x, t_1), (t_1, x, t_2), (t_0, y, t_2), (t_1, x, t_\infty), (t_1, y, t_\infty)\}$
$LRF(s) = \{(t_0, x, t_1), (t_1, x, t_\infty), (t_1, y, t_\infty)\}$
Now let us consider $s' = t_2t_1$ with:
$LRF(s') = \{(t_0, x, t_1), (t_1, x, t_\infty), (t_1, y, t_\infty)\} = LRF(s)$
that is, $s \approx_f s'$ and thus $s \in FSR$.

b) Consider all possible serial schedules for t_1 and t_2 and verify that none of them is view equivalent to s.
$s_{12} := t_1t_2$: $s_{12} \approx_v s$ is false for $(t_1, y, t_2) \in RF(s_{12})$ but $(t_1, y, t_2) \notin RF(s)$.
$s_{21} := t_2t_1$: $s_{21} \approx_v s$ is false for $(t_0, x, t_2) \in RF(s_{21})$ but $(t_0, x, t_2) \notin RF(s)$.
Therefore $s \notin VSR$.

Exercise 3.8 :

Show that $VSR = CSR$ in the absence of blind writes, i.e., if each write step on a data item x is preceded by a read step on x of the same transaction.

Let s be a schedule without blind writes.

a) We first show $s \in CSR \Rightarrow s \in VSR$. This follows immediately from Theorem 3.8 proven in the book (p. 95).

b) We have to show $s \in VSR \Rightarrow s \in CSR$. Assume that s is in VSR. Thus, there is a serial schedule s' for which $s \approx_v s'$ holds. Since s' is serial, s' is trivially conflict serializable as well.

Now, let us assume s would not be conflict serializable.

In such a case there is obviously at least one conflicting pair of steps, for which the order in s and s' differ. Otherwise s would result in the same conflict graph as s', i.e., s would be conflict serializable. Without loss of generality assume $t_i < t_j$ in s'.

$s' = w_0(x) \dots r_i(x) \dots w_i(x) \dots r_{i+1}(x) \dots w_{i+1}(x) \dots r_j(x) \dots w_j(x) \dots r_\infty(x)$

Case 1: s has a different order of write steps $w_i(x)$ and $w_j(x)$ (i.e.,
$w_j(x) <_s w_i(x)$). For every schedule that is view equivalent to s' (e.g., for s) must hold: $r_{i+1}(x)$ reads from $w_i(x)$. Note that the order of steps is unchangeable within a transaction. So we have $w_i(x) <_s r_{i+1}$ and $r_{i+1} <_s w_{i+1}$, and therefore $w_i(x) <_s w_{i+1}(x)$ for all i. Since $<_s$ is a transitive relation we also conclude $w_i(x) <_s w_j(x)$, which is a contradiction to the assumption above.

Case 2: s has a different order with regard to a read step $r_i(x)$ and a write step $w_j(x)$ (i.e., $w_j(x) <_s r_i(x)$). As for s' we have:
$r_i(x) <_{s'} w_i(x) <_{s'} r_j(x) <_{s'} w_j(x)$, because s' is a serial schedule. However we obtain for s: $w_j(x) <_s r_i(x)$ and $r_i(x) <_s w_i(x)$ imply $w_j(x) <_s w_i(x)$, i.e. the order of the write steps must be different as well. Then we can construct the same contradiction as in case 1.

Exercise 3.9 :

Let $s = r_1(z)r_3(x)r_2(z)w_1(z)w_1(y)c_1w_2(y)w_2(u)c_2w_3(y)c_3$.

Show that $s \in VSR - CSR$.

Consider the conflict graph for s, which obviously has a cycle ($t_1 \rightarrow t_2 \rightarrow t_1$) and therefore $s \notin CSR$:

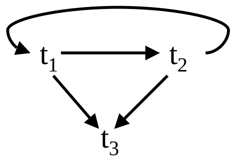


Figure 3: Conflict graph of schedule s

In order to show, that $s \in VSR$ we have to provide a serial schedule view equivalent to s. Consider s':

$$s' = t_2t_1t_3 = r_2(z)w_2(y)w_2(u)c_2r_1(z)w_1(z)w_1(y)c_1r_3(x)w_3(y)c_3,$$

$s \approx_v s'$ because:

$RF(s) = \{(t_0, z, t_1), (t_0, x, t_3), (t_0, z, t_2), (t_0, x, t_\infty), (t_3, y, t_\infty), (t_1, z, t_\infty), (t_2, u, t_\infty)\} = RF(s')$

Exercise 3.10 :

Consider $s = r_1(x)w)1(z)w_2(z)w_1(y)c_1r_3(y)w_2(z)c_2w_3(x)w_3(y)c_3$. Using the conflict graph of s as an argument, show that $s \in CSR$. Does $s \in OCSR$ also hold?

Since we have got an acyclic conflict graph for s, $s \in CSR$ holds.

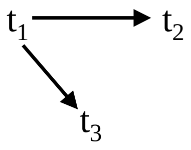


Figure 4: Conflict graph of schedule s

By topological sorting of the conflict graph we obtain a conflict equivalent serial schedule $s' = t_1t_2t_3$. Only the transactions t_1 and t_3 are ordered in the original schedule s; since their order is preserved in s', we obtain $s \in OCSR$.

Exercise 3.11 :

Show:

a) Membership in class CSR is monotone.

b) $s \in$ CSR $\Longleftrightarrow (\forall\, T \subseteq \text{trans}(s))\ \Pi_T(s) \in$ VSR
(i.e., CSR is the largest monotone subset of VSR).

Let s be in CSR and s' be an arbitrary subschedule of s. $s \in CSR$ implies that the conflict graph $G(s)$ is acyclic (Theorem 3.10).
$op(s') \subset op(s) \Rightarrow conf(s') \subseteq conf(s)$. Thus $G(s')$ is a subgraph of $G(s)$, so it must be acyclic too, which in turn implies $s' \in CSR$ as well. We showed that $a)$ holds.

$b)$ was shown by Mihalis Yannakakis in *Yannakakis, M. (1984): Serializability by Locking. Journal of the ACM 31, pp. 227-244.* See Section 3 on pp.232-233

Chapter 4 - Solutions to Exercises

Exercise 4.1 :

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

$$s_1 = w_1(x)r_2(y)r_1(x)c_1r_2(x)w_2(y)c_2$$
$$s_2 = r_1(x)r_2(x)w_3(x)w_4(x)w_1(x)c_1w_2(x)c_2c_3c_4$$

(a) We first consider s_1:

- **2PL, S2PL or SS2PL:**
Figure 1 shows an example execution, which could be produced by a scheduler using all of these three variations of two-phase-locking protocol because both read and write locks are being released immediately before committing the corresponding transaction. The dashed line represents the time interval during

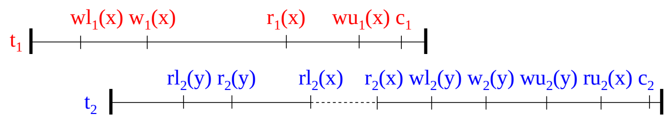


Figure 1: 2PL, S2PL or SS2PL output for s_1

which the lock request is being blocked. This leads us to the following execution order for the extended schedule s_1':

$$s_1' = wl_1(x)w_1(x)rl_2(y)r_2(y)r_1(x)wu_1(x)rl_2(x)r_2(x)c_1wl_2(y)w_2(y)wu_2(y)ru_2(x)c_2$$

- **TO:**
We observe $ts(t_1) < ts(t_2)$. For the only conflicting pair $(w_1(x), r_2(x))$ we have $w_1(x) <_{s_1} r_2(x)$, which implies that s_1 can be executed as is under the TO protocol.

- **SGT:**
Since s_1 comprises only one conflicting pair no cycle can arise for any prefix of s_1, i.e., s_1 can be executed as is by an SGT scheduler, too.

(b) Now consider s_2:

- **2PL, S2PL or SS2PL:**
Figure 2 shows a possible execution of s_2. When $wl_2(x)$ is issued and not granted because $rl_1(x)$ is still being held by t_1 , a cycle arises in the waits-for graph (WFG), because t_1 itself is already waiting for t_2 to release $rl_2(x)$. In order to resolve this deadlock the scheduler has to abort either t_1 or t_2. In our case we assume t_1 is aborted. Then the extended output schedule s_2' is as follows:

$$s_2' = rl_1(1)r_1(x)rl_2(x)r_2(x)a_1wl_2(x)w_2(x)wu_2(x)wl_3(x)w_3(x)c_2$$
$$wu_3(x)wl_4(x)w_4(x)c_3wu_4(x)c_4$$

Once again the locks are held until the transactions terminate, so that S2PL and SS2PL could produce exactly the same output schedule.

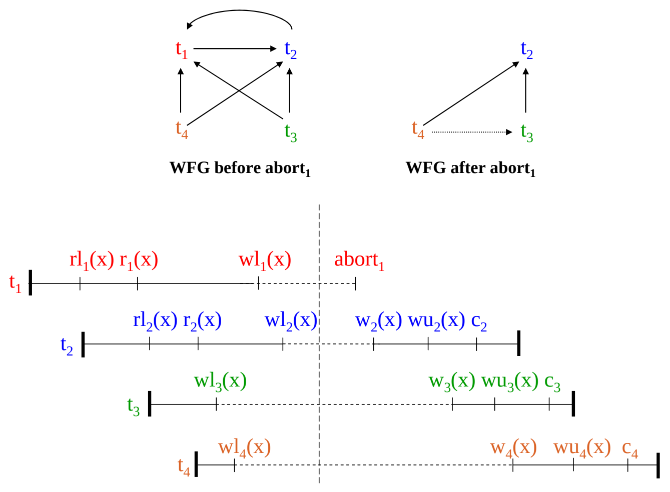


Figure 2: 2PL, S2PL or SS2PL output for s_2

- **TO:**
The input schedule is executed as is until $w_1(x)$ occurs for which we obtain $w_4(x) <_{s_2} w_1(x)$ but $ts(t_4) > ts(t_1)$ as you can observe in Figure 3. Therefore t_1 must be aborted. Analogously, $w_2(x)$ also comes too late, so that t_2 must be aborted as well. The output schedule s_2' then is as follows:

$$s_2' = r_1(x)r_2(x)w_3(x)w_4(x)a_1a_2c_3c_4$$

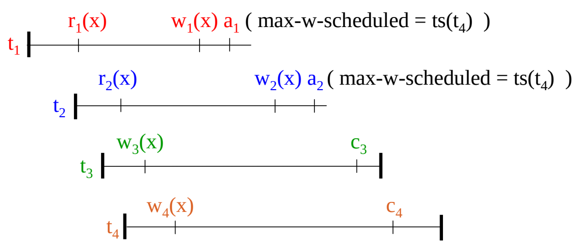

t1
r1(x)
w1(x) a1 (max-w-scheduled = ts(t4))
t2
r2(x)
w2(x) a2 (max-w-scheduled = ts(t4))
t3
w3(x)
c3
t4
w4(x)
c4

Figure 3: BTO output for s_2

- **SGT:**
In the case of a SGT scheduler the input schedule s_2 is also executed without any changes until $w_1(x)$ occurs, because no cycle arises in the serialization graph so far. However at the point $w_1(x)$ should be passed to the data manager several cycles come into existence as you can see by analyzing the serialization graph in Figure 4. So the scheduler rejects $w_1(x)$ and aborts t_1. With $w_2(x)$ a new edge is added to the serialization graph closing a new cycle $t_3 \rightarrow t_2 \rightarrow t_3$, and t_2 is aborted as well. The resulting output schedules is as follows:

$$s_2' = r_1(x)r_2(x)w_3(x)w_4(x)a_1a_2c_3c_4$$

Exercise 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

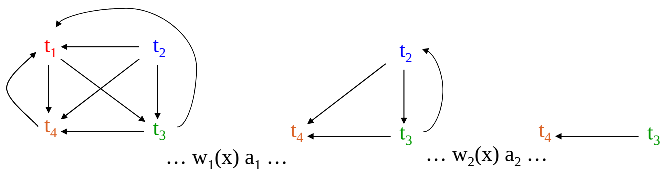


Figure 4: Serial (conflict) graph of s_2

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.

Let $trans(s) = \{t_1, ..., t_n\}$ with the corresponding lock points $lp_1, ..., lp_n$. For each $(p_i, q_j) \in conf(s)$ we observe:

$$p_i <_s lp_i <_s pu_i <_s ql_j <_s q_j <_s lp_j$$

Thus, if we simply put the transactions in the same order as the lock points occur in s, we obtain a serial schedule s' with $conf(s) = conf(s')$, i.e., s' is conflict equivalent to s' .

Exercise 4.3 :

Under the 2PL protocol it is possible for transactions to “starve” in the following sense: A transaction gets involved in a deadlock, is chosen as the victim and aborted. After a restart, it again gets 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.

Let $t_1 = r_1(x)w_1(y)$, $t_2 = r_2(y)w_2(x)$. Consider following schedule assuming that the scheduler aborts the youngest transaction whenever a deadlock occurs:

$$s = [rl_1(x)r_1(x)rl_2(y)r_2(y)wl_1(y) \text{ ... } wl_2(x) \text{ ... } a_2w_1(y)c_1]^+$$

The expression in the squared brackets is repeated several times. A scenario, we observe is the following: imagine you have a typical application which generates transactions with the same operation pattern, e.g., many instances of t_1 . Occasionally another application issues a request resulting in t_2 . Convince yourselves that any interleaving of t_1 and t_2, which is very likely on a multi-user server, is not conflict serializable. Thus, t_2 gets involved in a deadlock with an instance of t_1 over and over again and is permanently aborted by the scheduler.

In order to prevent the starvation the scheduler should take the number of transaction retries into account when choosing a deadlock victim. For example, let p_i be a priority that the scheduler would assign to transaction t_i under “normal” circumstances and r_i be the number of its restarts (transactions with lower priority are chosen as deadlock victims). So let the scheduler work with a weighted priority $pw_i = r_i \cdot p_i$. Each time the transaction is involved in a deadlock it automatically gets a higher priority than usual, so that the likelihood of being aborted once again decreases with increasing number of restarts. Because of this monotonicity property, a transaction that has suffered many aborts, must eventually have the highest weighted priority among all transactions.

Exercise 4.4 :

Describe the wait-for graphs resulting from the use of 2PL for each of the histories in Exercise 4.1.

We first consider the WFG for s_1. By observing its execution in Figure 1 you can deduce how the WFG evolves:

1. When rl_2 is issued the edge $t_2 \rightarrow t_1$ is added to the WFG because $rl_1(x)$ is not released yet.
2. When $wu_1(x)$ occurs, $rl_1(x)$ can be granted and the edge $t_2 \rightarrow t_1$ is removed from the WFG.

Now let us turn to s_2. As you can see in Figure 2 the WFG of s_2 evolves as follows:

1. Upon $wl_3(x)$ the edges $t_3 \rightarrow t_1$ and $t_3 \rightarrow t_1$ are added because $rl_1(x)$ and $rl_2(x)$ have already been granted at that time.
2. Analogously, the issuing of $wl_4(x)$ leads to adding the edges $t_4 \rightarrow t_1$ and $t_4 \rightarrow t_1$.
3. When $wl_1(x)$ is issued but not granted because of $rl_2(x)$, a new edge $t_1 \rightarrow t_2$ is added to the WFG.
4. With $wl_2(x)$ waiting for $rl_1(x)$ a cycle $t_1 \rightarrow t_2 \rightarrow t_1$ comes into existence.
5. The scheduler resolves the deadlock by aborting t_1 and thus the corresponding node along with its incoming and outgoing edges is removed from the WFG.
6. Then the scheduler selects $wl_2(x)$ to be granted implying that the blocked requests $wl_3(x)$ and $wl_4(x)$ are now waiting for $wl_2(x)$. This way $t_3 \rightarrow t_2$ and $t_4 \rightarrow t_2$ are added to the WFG.
7. After $wu_2(x)$ is executed, $t_3 \rightarrow t_2$ and $t_4 \rightarrow t_2$ are removed.
8. $wl_3(x)$ is granted, so $wl_4(x)$ has to wait and we add $t_4 \rightarrow t_3$.
9. Finally, after $wu_3(x)$ there are no more pending lock requests which could not be granted and the WFG has no edges at all.

Exercise 4.5 :

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

We first consider the *wound-wait* approach. We observe that a transaction can be blocked only by an older transaction, i.e., if t_i is blocked by t_j (or in other words t_i waits for t_j) then $ts(t_j) < ts(t_i)$ holds. If there were a cycle with a transaction t_k involved in it, we would obtain $ts(t_k) < ts(t_k)$, which is obviously false.

For the *wait-die* approach ensuring that a transaction can be blocked only by a younger transaction, we can reason analogously, so that the same contradiction $ts(t_k) < ts(t_k)$ is constructed.

Exercise 4.8 :

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) $s_1 = w_1(x)r_2(x)c_2r_3(y)c_3w_1(y)c_1$

b) $s_2 = w_1(x)r_2(x)r_3(y)c_3r_2(z)c_2w_1(y)c_1$

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.

In Figure 5 you can view the details of processing s_1 under O2PL. A resulting schedule s'_1 is as follows:

$$wl_1(x)w_1(x)rl_2(x)r_2(x)rl_3(y)r_3(y)ru_3(y)c_3$$
$$wl_1(y)w_1(y)wu_1(x)ru_2(x)c_2wu_1(y)c_1$$

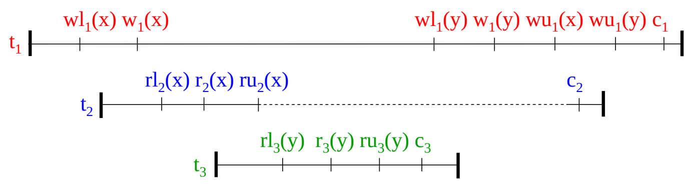


Figure 5: O2PL output for s_1

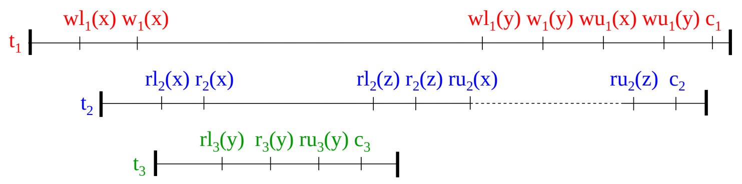


Figure 6: O2PL output for s_2

Figure 6 shows corresponding processing of s_2. One of the possible resulting schedules is:

$$wl_1(x)w_1(x)rl_2(x)r_2(x)rl_3(y)r_3(y)ru_3(y)c_3$$
$$rl_2(z)r_2(z)wl_1(y)w1(y)wu_1(x)ru_2(x)ru_2(z)wu_1(y)c_2c_1$$

Exercise 4.9 :

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

Consider the following schedule:

$$\begin{array}{cccccc} r_1(x) & & & & w_1(y) & \text{ul} \\ & r_2(y) & w_2(x) & & & \text{ul} \end{array}$$

This is a deadlock: both transactions are on hold and wait for the other to start releasing locks.

Exercise 4.11 :

Investigate the relationship between $Gen(BTO)$ and $Gen(2PL)$. Is one more powerful than the other?

a) We first show that $Gen(BTO) \not\subset Gen(2PL)$. Consider the following simple schedule that can be produced by a 2PL scheduler but not by one running BTO:
$r_1(z)r_2(y)w_2(x)c_2w_1(x)c_1$
With BTO t_1 is aborted when it attempts to write x, because $ts(t_1) < ts(t_2)$ and t_2 has already written x. However, this schedule can be executed as is under 2PL.

b) Now we show that $Gen(2PL) \not\subset Gen(BTO)$ also holds. This can be verified by considering the following schedule:
$r_1(z)r_2(z)w_2(x)r_3(x)r_1(y)w_2(y)c_1c_2c_3$
This schedule can be produced by a BTO scheduler because the order of conflicting steps coincides with the order of the transaction timestamps. However, the following would have to hold, if this schedule were a 2PL output:
   - $wl_2(x) < w_2(x) < wu_2(x) < rl_3(x) < r_3(x)$, i.e, $wu_2(x) < r_3(x)$

- $rl_1(y) < r_1(y) < ru_1(y) < wl_2(y) < w_2(y)$, i.e., $r_1(y) < wl_2(y)$

  Since we have $r_3(x) < r_1(y)$, we also obtain $wu_2(x) < wl_2(y)$, which contradicts the two-phase property. We deduce, that this schedule cannot be produced by a 2PL scheduler.

So the two protocols are incomparable with regard to this "power".

Exercise 4.12 :

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

Consider the following prefix of a schedule: $w_1(z)r_2(x)w_1(x)c_1r_3(y)w_2(y)c_2$. The corresponding conflict graph looks as follows:

$$\begin{tikzcd}
t_1 & & t_2 \arrow[ll] \\
& t_3 \arrow[ur] &
\end{tikzcd}$$

Since the condition is true for t_1, we could already remove t_1 at this point. Now assume the next pending operation in the schedule is $r_3(z)$, which would add a new edge $t_1 \rightarrow t_3$ to the conventional conflict graph so that a cycle arises. In the modified graph we are no more able to detect this cycle, because t_1 has already been removed. Thus, this condition cannot assure a CSR-safe execution.

Exercise 4.13 :

Prove Theorem 4.18, thereby showing that FOCC is a CSR safe protocol.

Let s be a schedule produced by a FOCC scheduler. Consider two arbitrary transactions $t_i, t_j \in trans(s)$ with $val(t_i) <_s val(t_j)$. Let us consider all possible conflicts:

- $r_i(x) <_s val(t_i) <_s val(t_j) \Rightarrow r_i(x) <_s w_j(x)$, if there is a read-write conflict.
- If there is a write-read conflict $val(t_i) <_s r_j(x)$ must hold, because otherwise t_i could not be validated. Hence $w_i(x) <_s r_j(x)$
- For a write-write conflict we trivially have $w_i(x) <_s w_j(x)$ which results from $val(t_i) <_s val(t_j)$

Summarizing these observations we obtain: if $G(s)$ has an edge (t_i, t_j) then $val(t_i) <_s val(t_j)$ holds.

Assume $G(s)$ has a cycle $t_{i_1} \rightarrow t_{i_2} \rightarrow \ \dots \ \rightarrow t_{i_n} \rightarrow t_{i_1}$. This implies, however, $val(t_{i_1}) <_s val(t_{i_2}) <_s \ \dots \ <_s val(t_{i_n}) <_s val(t_{i_1})$. Since $<_s$ is transitive, we obtain $val(t_{i_1}) <_s val(t_{i_1})$ which is a contradiction. Thus $G(s)$ must be acyclic.

Exercise 4.14 :

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:

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

(b) $RS(t_j) \cap WS(t_i) = \emptyset$ 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.

As an example consider the following schedule:

$$\begin{array}{lll} r_i(z)val_iw_i(x) & & w_i(y)c_i \\ & r_j(z)val_jw_j(x)w_i(y)c_j & \end{array}$$

t_i can be clearly validated, since there is no other transaction running at that time. The validation of t_j is approved as well, because $RS(t_j) \cap WS(t_i) = \emptyset$ holds and t_i is already writing. However, since we now allow the write phase of t_i to be interleaved with the validation of t_j , several write-write conflicts can occur which can lead to a cycle in the conflict graph. In particular, $(w_i(x), w_j(x))$ and $(w_j(y), w_i(y))$ produce such a cycle.

But, if we have $WS(t_k) \subseteq RS(t_k)$ for all k , then we obtain for every validated transaction t_j:

$$RS(t_j) \cap WS(t_i) = \emptyset \Longrightarrow WS(t_j) \cap WS(t_i) = \emptyset$$

This implies that we can commute (applying rule C3, p. 99) concurrent write steps such that $val_i < writes_i < val_j < writes_j$ holds. Clearly this results in a conflict equivalent schedule and this is exactly the schedule that would have been produced under the original BOCC protocol. Thus, we obtain, by applying Theorem 4.17, that with $WS(t_k) \subseteq RS(t_k)$ the modified BOCC provides CSR-safe execution.

Exercise 4.15 :

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?

No, such a protocol would not be correct. As an example, we consider the following execution that could happen under BOCC:

$$\begin{array}{cccccc} & w_i(x) & w_i(y) & c_i & & \\ r_j(x) & & & & r_j(y) & c_j \end{array}$$

When t_i is validated, there is no other validated transaction, so the validation is trivially successful. When t_j is validated afterwards, the only concurrent and already validated transaction is t_i, whose read set is empty, so the validation of t_j is successful, too.

However, there is a cycle in the schedule's conflict graph: there is an edge from t_j to t_i, because t_j reads x and t_i later writes it, and there is an edge from t_i to t_j, because t_i writes y and t_j later reads it. So this schedule is not in CSR, but can be generated by this variant of BOCC.

Exercise 4.16 :

Consider the following input schedule of three concurrent transactions:

$r_1(x)r_2(x)r_1(y)r_3(x)w_1(x)w_1(y)c_1r_2(y)r_3(z)w_3(z)c_3r_2(z)c_2$

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.

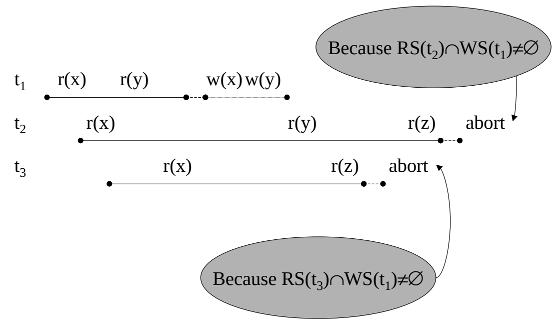


Figure 7: Resulting schedule under the BOCC protocol

See Figures 7 and 8 for the output schedules under BOCC and FOCC, respectively.

$$BOCC \rightarrow r_1(x)r_2(x)r_1(y)r_3(x)w_1(x)w_1(y)c_1r_2(y)r_3(z)a_3r_2(z)a_2$$

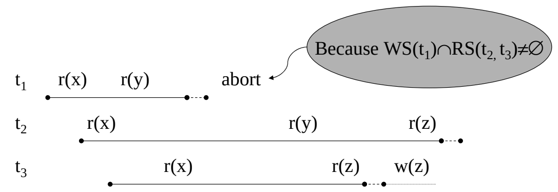


Figure 8: Resulting schedule under the FOCC protocol

$$FOCC \rightarrow r_1(x)r_2(x)r_1(y)r_3(x)a_1r_2(y)r_3(z)w_3(z)c_3r_2(z)c_2$$

Exercise 4.17 :

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

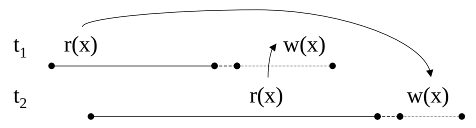


Figure 9: Incorrect FOCC execution without the critical section

See Figure 9 for an example execution where the critical-section condition for the val-write phase is dropped. We get a cycle in the conflict graph of the schedule, because $r_1(x)$ is executed before $w_2(x)$, and $r_2(x)$ is executed before $w_1(x)$. Therefore, this execution is not serializable.

The problem is that t_2's read operation takes place after the validation phase of t_1, but before (or during) the write phase and before the commit of t_1.

Exercise 4.18 :

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.

a) BOCC uses only weak critical sections: no two transactions are allowed to be simultaneously in their val-write phases. To further relax this, one can use C2PL (i.e., 2PL with preclaiming) during the write phase (only exclusive locks on write sets). This serializes the write phases of those transactions that have a ww conflict, but allows concurrent execution of the write phases of transactions without overlapping write sets.

b) FOCC needs a strong critical section: no progress allowed for concurrent transactions during val-write phase, not even reads. To relax this, one can use C2PL for the write set during the write phase; other transactions need to acquire read locks for further progress. This introduces a wr-synchronization between concurrent transactions, additionally to the ww synchronization, but no rw-synchronization is necessary.

   The implementation for such a protocol needs only write locks, readers are blocked upon the first conflict.

Compared to the original BOCC and FOCC protocols, relaxing the need for critical sections increases concurrency, but the additional locks during validation and write phases incur overhead.

As only write locks need to be explicitly held, the overhead is still somewhat lower than that of pure locking protocols.

Chapter 5 - Solutions to Exercises

Exercise 5.1 :

For the following three histories, test whether they are monoversion histories or members of MVSR or MCSR; in case a schedule is a member of MVSR, additionally find out for which values of $k > 0$ the history is in class kVSR:

$$\begin{aligned}
m_1 &= w_0(x_0)w_0(y_0)w_0(z_0)c_0r_3(x_0)w_3(x_3)c_3w_1(x_1)c_1r_2(x_1)w_2(y_2)w_2(z_2)c_2 \\
m_2 &= w_0(x_0)w_0(y_0)c_0w_1(x_1)c_1r_3(x_1)w_3(x_3)r_2(x_1)c_3w_2(y_2)c_2 \\
m_3 &= w_0(x_0)w_0(y_0)c_0w_1(x_1)c_1r_2(x_1)w_2(y_2)c_2r_3(y_0)w_3(x_3)c_3
\end{aligned}$$

For schedules in MVSR, also give an appropriate version function for a final transaction t_∞.

a) m_1 is a serial monoversion schedule. Thus, we trivially have $m_1 \in$ MCSR and $m_1 \in$ MVSR. Furthermore $m_1 \in$ 1VSR implies $m_1 \in k$VSR for all $k > 0$.

   $t_\infty = r_\infty(x_1)r_\infty(y_2)r_\infty(z_2)$

b) We can transform the schedule m_2 into a schedule m_2' by commuting non-conflicting operations of t_3 and t_2:

$$m_2' = w_0(x_0)w_0(y_0)c_0w_1(x_1)c_1r_2(x_1)w_2(y_2)c_2r_3(x_1)w_3(x_3)c_3$$

   m_2' is obviously a serial monoversion schedule, i.e. m_2 is multiversion reducible, and applying Theorem 5.5 we obtain $m_2 \in$ MCSR. $m_2 \in$ MVSR holds as well, because MVSR is a superset of MCSR.

   Now let us determine the smallest k, for which $m_2 \in k$VSR holds. We observe that $r_2(x_1)$ follows $w_3(x_3)$, which implies that at least two versions must be stored at the same time. Thus, for all $k \geq 2$ we have $m_2 \in k$VSR.

   $t_\infty = r_\infty(x_3)r_\infty(y_2)$

c) As for m_3 we observe first that there is only one conflict pair in this schedule, namely, $(r_2(x_1), w_3(x_3))$, i.e., its multiversion conflict graph is acyclic. Therefore m_3 is a member of both MCSR and MVSR. We obtain an equivalent serial monoversion schedule by commuting t_0's operations to the right, just before t_3, resulting in the schedule $t_1t_2t_0t_3$. While it may appear to be counter-intuitive putting the initializing transaction into the middle of the schedule, it is allowed, if the order of conflicting steps remains unaffected.

   Finally, by observing $r_3(y_0)$ in the original schedule m_3 we conclude that also for m_3 at least two versions must be stored simultaneously.

   $t_\infty = r_\infty(x_3)r_\infty(y_0)$

Exercise 5.2 :

For the multiversion schedule

$$m = w_0(x_0)w_0(y_0)c_0r_1(x_0)w_1(x_1)r_2(x_1)w_2(y_2)w_1(y_1)w_3(x_3)$$

test whether there exists a version order $\ll$ such that MVSG$(m, \ll)$ is acyclic. If there is an acyclic graph, find an appropriate version function for a final transaction t_∞ such that the graph remains acyclic.

We choose the version orders $x_0 \ll x_1 \ll x_3$ and $y_0 \ll y_1 \ll y_2$. Now we can start building up the graph MVSG(m, $\ll$):

- We add the edge $t_0 \to t_1$ because of $r_1(x_0)$.
- Then we consider $r_1(x_0)$ and $w_3(x_3)$. Since $x_3 \gg x_0$, the edge $t_1 \to t_3$ is added.
- The edge $t_1 \to t_2$ is added, because $r_2(x_1) \in op(m)$
- By considering $r_2(x_1)$ and $w_0(x_0)$ we add the edge $t_0 \to t_1$, since we have $x_0 \ll x_1$.
- Finally, we consider $r_2(x_1)$ and $w_3(x_3)$, and add the edge $t_2 \to t_3$, because $x_3 \gg x_1$.

Thus we obtain an acyclic multiversion serialization graph depicted in Figure 1. The original schedule m then can be extended by $r_\infty(x_3)$ and $r_\infty(y_2)$. The dashed

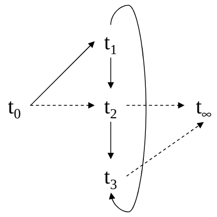


Figure 1: MVSG(m, $\ll$) for Exercise 5.2

arrows represent the new edges in the updated MVSG. As you see the graph is still acyclic.

Exercise 5.3 :

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

Let m be a schedule without "blind writes":

a) According to Theorem 5.6 stating MCSR $\subset$ MVSR we immediately obtain: $m \in$ MCSR $\Longrightarrow m \in$ MVSR.

b) Now, assume $m \in$ MVSR. That is, there is a serial monoversion schedule m' with $m \approx_v m'$. Without loss of generality assume m' is as follows (with $0 < i < j$):

$$w_0(x_0)\ldots r_i(x_{i-1})\ldots w_i(x_i)\ldots r_{i+1}(x_i)\ldots w_{i+1}(x_{i+1})\ldots r_j(x_{j-1})\ldots w_j(x_j)\ldots$$

m' is trivially conflict serializable. Assume again, m itself is not in MCSR. That is, there is at least one write-read pair, such that $w_i(x_i) <_{m'} r_j(x_{j-1})$, but $r_j(x_{j-1}) <_m w_i(x_i)$, because otherwise they would have identical multiversion conflict graphs. We observe $(t_k, x, t_{k+1}) \in$ RF(m')=RF(m) for $i \le k < j$. Since a version can be read only after it is created, we derive:

$$w_i(x_i) <_m r_{i+1}(x_i) <_m w_{i+1}(x_{i+1}) \ldots <_m r_j(x_{j-1})$$

Finally, we obtain by transitivity $w_i(x_i) <_m r_j(x_{j-1})$ which contradicts the assumption.

Exercise 5.5 :

Consider the following schedule, given in "conventional" form without a specific version function:

$$r_1(x)r_2(x)r_3(y)w_2(x)w_1(y)c_1w_2(z)w_3(z)r_3(x)c_3r_2(y)c_2$$

Show that this schedule is multiversion serializable, i.e., could be allowed by a multiversion concurrency control. Give a feasible version function, and also a feasible version order. How do the resulting executions (i.e., output schedules) under the MVTO and the 2V2PL protocols look like?

First note that there are the following conflict pairs in the schedule: $(r_1(x), w_2(x))$ and $(r_3(y), w_1(y))$. So it is clear that the multiversion conflict graph associated with this schedule has no cycles, and, thus, the schedule is in MCSR and MVSR. By considering the two conflict pairs we can deduce the serialization order

$$t_0 < t_3 < t_1 < t_2 < t_\infty$$

These considerations lead us to the following mutiversion schedule:

$$w_0(x_0)w_0(y_0)w_0(z_0)r_1(x_0)r_2(x_0)r_3(y_0)w_2(x_2)w_1(y_1)w_2(z_2)$$
$$w_3(z_3)r_3(x_0)c_3r_2(y_1)c_2r_\infty(x_2)r_\infty(y_1)r_\infty(z_2)$$

The version order obtained this way is as follows: $x_0 \ll x_2$; $y_0 \ll y_1$; $z_0 \ll z_3 \ll z_2$. Figure 2 depicts the corresponding MVSG, which is acyclic as expected.

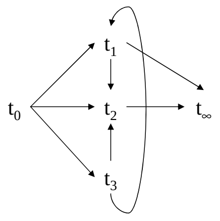


Figure 2: MVSG (Exercise 5.5)

Figure 3 shows the output schedule under the MVTO protocol. As you can see the write step $w_1(y_1)$ comes too late because $r_3(y_0)$ has already been executed at that time. Since $ts(t_0) < ts(t_1) < ts(t_3)$, $w_1(y_1)$ is rejected and t_1 must be aborted. The commit of t_3 is delayed until the commit of t_2 because of $r_3(x_2)$.

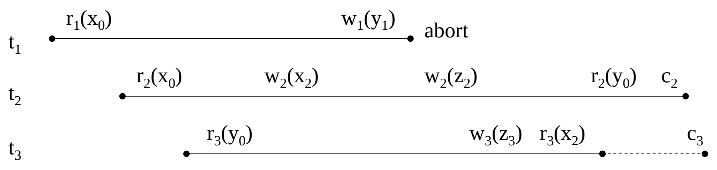


Figure 3: MVTO output (Exercise 5.5)

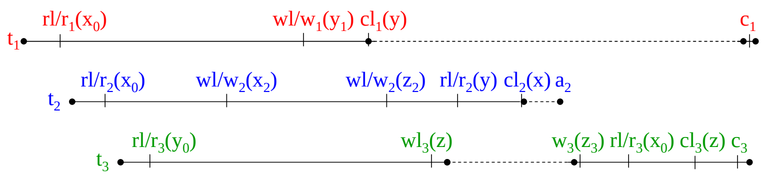


Figure 4: 2V2PL output (Exercise 5.5)

The resulting output under the 2V2PL protocol is shown in Figure 4. The write lock request $wl_3(z)$ could not be granted immediately because t_2 still holds $wl_2(z)$ and only one write lock on a data item is allowed at the same time under 2V2PL. Thus, the edge $t_3 \rightarrow t_2$ is added to the WFG. Then, $cl_1(y)$ has to wait for the release of $rl_3(y)$, so that $t_1 \rightarrow t_3$ is added to the WFG. And finally, as t_2 tries to acquire $cl_2(x)$ which is delayed because of $rl_1(x)$ still being present, the edge $t_2 \rightarrow t_1$ is added to the WFG closing a cycle. Thus, a deadlock is detected, and the scheduler rejects $cl_2(x)$ and t_2 is aborted, so that the execution can eventually come to an end.

This exercise gives another example that Gen(MVTO) and Gen(2V2PL) are proper subsets of MVSR.

Exercise 5.6 :

Consider the input schedule:

$$w_1(x)c_1r_2(x)r_3(x)c_2r_4(x)w_3(x)c_4c_3$$

Give the resulting output schedule under the MVTO protocol.

See Figure 5 for the solution.

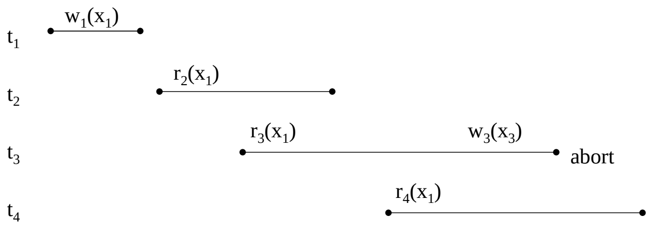


Figure 5: MVTO output (Exercise 5.6)

Exercise 5.7 :

Consider the input schedule of the MV2PL Example 5.9:

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

Apply the specialized 2V2PL protocol to this input and give the resulting output.

Figure 6 depicts the output schedule under 2V2PL.

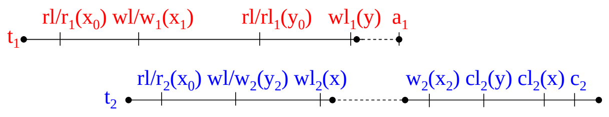

$rl/r_1(x_0)$ $wl/w_1(x_1)$ $rl/rl_1(y_0)$ $wl_1(y)$ a_1
t_1
$rl/r_2(x_0)$ $wl/w_2(y_2)$ $wl_2(x)$ $w_2(x_2)$ $cl_2(y)$ $cl_2(x)$ c_2
t_2

Figure 6: 2V2PL output (Exercise 5.7)

Exercise 5.9 :

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

This protocol is not correct anymore, as it allows the following schedule:

$$\begin{array}{llll} r_1(x_0)r_1(y_0) & & w_1(x_1)c_1 & \\ & r_2(x_0)r_2(y_0) & & w_2(y_2)c_2 \end{array}$$

Based on the timestamps we should obtain the serialization order $t_1 < t_2$. t_2, however, reads x from t_0 instead. The other serialization, $t_2 < t_1$, is obviously not possible as well, since t_1 reads y from t_0. Therefore, this schedule is not in MVSR.

So the relaxed ROMV variant cannot guarantee the consistency of the data. Assume, for example, that x and y are two counters (e.g., stock investments) with the constraint $x + y \leq 100$ (e.g., for risk limitation). The initial values are $x = 40$ and $y = 40$. Then, t_1 could set x to 60, and t_2 could set y to 60, thus arriving in a state that violates the constraint. The relaxed ROMV variant would erroneously allow this schedule.

Chapter 6 - Solutions to Exercises

Exercise 6.1 :

Consider the two-level schedule shown in Figure 6.12, which consists of two funds transfer transactions and a balance lookup transaction. Assume that the GetBalance operations do not commute with Withdraw operations nor with Deposit operations. Is this schedule tree reducible? If not, how would the schedule have to be changed (as little as possible) so as to render it tree reducible?

To see if the schedule is tree reducible, we first try to isolate subtrees. We can isolate the subtree with root $Withdraw(a)$ by commuting its operations $r(r)$ and $r(p)$ to the right (denoted by the arrows in the following figure). Additionally, we can commute $r(r)$ in the subtree with root $GetBalance(c)$ to the right, so that the subtree with root $Withdraw(a)$ is isolated. Analogously, we commute $r(r)$ in the subtree rooted by t_1's $Deposit(c)$ operation to the right. Figure 1 shows all those steps:

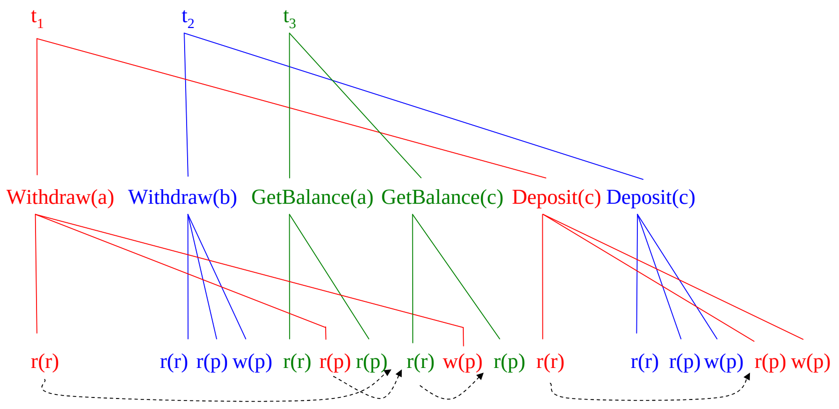


Figure 1: Commutations at the page access level

Now all subtrees are isolated, and we can prune them. The result is a one-level schedule. Using commutativity of the operations on this level, we see that we can commute them in such a way that all subtrees are isolated. Figure 1 shows the necessary steps:

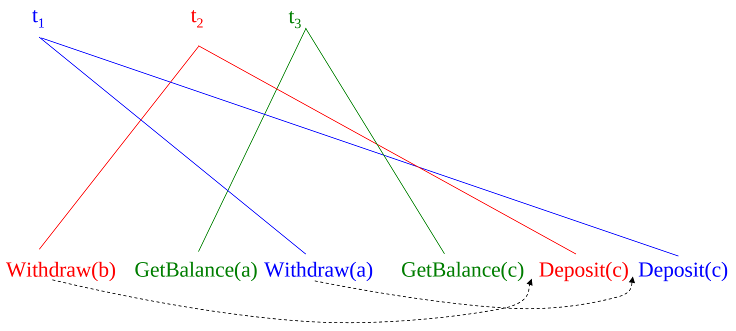


Figure 2: Commutation at level L_1

Now all subtrees are isolated, we can prune them and get the equivalent schedule $t_3 t_2 t_1$.

Exercise 6.2 :

Consider a data server with an SQL-style interface (see Chapter 1) and a database with a *Person* table whose rows contain a unique *Name* attribute and a *City* attribute. The server executes operations like

- `Select * From Person Where City=c`, where `c` is a parameter, for looking up all persons that live in a given city, and
- `Update Person Set City=c Where Name=n`, for recording that a given Person has moved to a new city.

Let us abreviate these two operation types as *select(c)* and *update(n,c)*, respectively. The server executes these operations by decomposing them into index lookups (*search(key)*), record fetches (*fetch(rid)*), record modifications (*modify(rid)*), and index maintenance steps (*insert(key,rid)* and *delete(key,rid)*). Assume that B$^+$-tree index structures exist for each of the two relevant attributes, and that both of these trees have depth two, i.e., consist of a root node and a leaf level. All of the mentioned record and index operations are finally transformed into page reads and writes.

Now consider a transaction that finds all persons from two different cities, say Los Angeles and New York, and a second transaction that records the move of a couple, say, Jerry and Liz Smith, from Los Angeles to New York. Model the executions of both transactions as three-level transactions, and discuss possible schedules for them. Give (non-trivial, e.g., non-serial) examples for three-level schedules that are a) tree reducible and b) not tree reducible and thus not admissible.

$$t_1 = select('LA')\ select('NYC')$$
$$t_2 = update('Jerry\ Smith',\ 'NYC')\ update('Liz\ Smith',\ 'NYC')$$

Let r denote a root node page and l denote a leaf node page of the B^+ tree, p denote a data page. Now we can start modelling the operations:

- $select(c) = search(c)\ (foreach\ rid \in hits\ fetch(rid))$
- $update(c) = search(c)\ (foreach\ rid \in hitsfetch(rid)\ modify(rid))$
- $search(c) = r(r)r(l)$
- $insert(c, rid) = r(r)r(l)w(l)$
- $delete(c, rid) = r(r)r(l)w(l)$
- $fetch(rid) = r(p)$
- $modify(rid) = r(p)w(p)$

For simplicity we assume that the database initially contains no persons with the residence in New York City and the couple Smith are the only residents of Los Angeles registered in the databases.

Figure 3 depicts a non-reducible object schedule, where the above operations are instantiated with data pages p_1, p_2, and index pages r_1, l_1 for the City-index as well as r_2, l_2 for the Name-index. When we isolate and prune subtrees at the levels L_0 and L_1, we obtain

$$select_1('LA') <_{L_2} update_2('JerrySmith','NYC') <_{L_2} select_1('NYC')$$

that is, the transaction trees cannot be further isolated, since none of the operations is commutable. Note that the *update* operation removes a row from the result set of the first *select*, and it adds a new entry to the result set of the second one. This is a typical example for the so-called *phantom problem* discussed in Section 8.2.

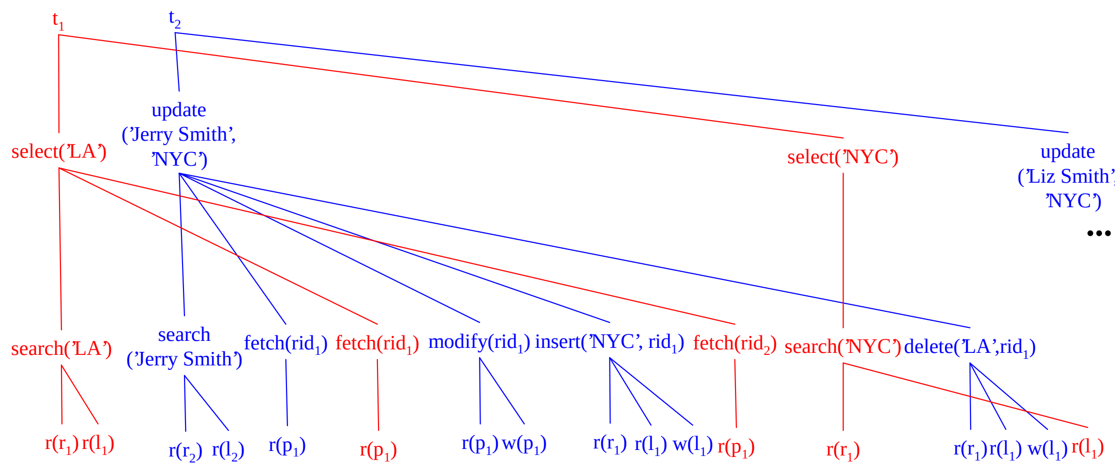


Figure 3: Non-reducible schedule

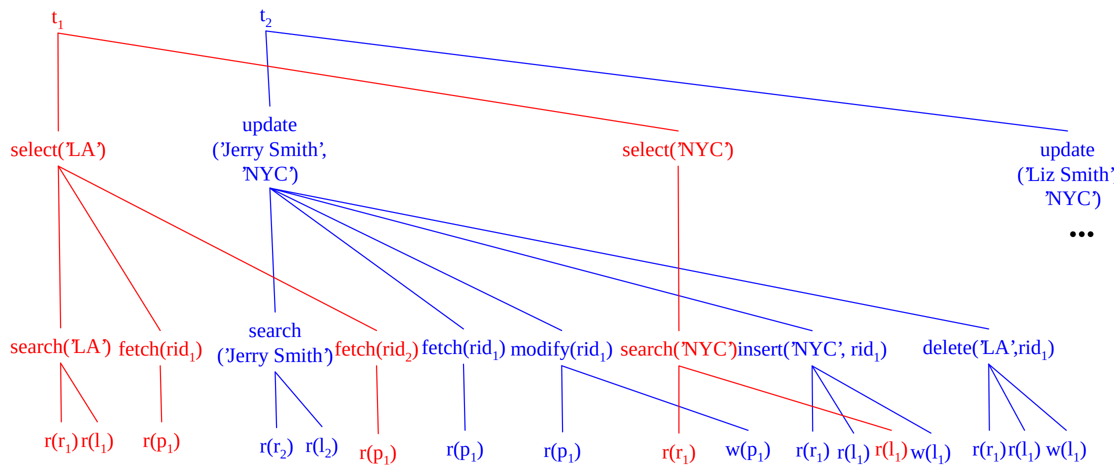


Figure 4: Reducible schedule

The schedule presented in Figure 4 is tree reducible with the serialization $t_1 <_s t_2$. First we isolate $search_1('NYC')$ by commuting its page level descendants to the left such that the corresponding subtree appears entirely between $fetch_2(rid_1)$ and $modify_2(rid_1)$ rooted by $update_2('JerrySmith', 'NYC')$. Thereafter all L_1 operations are isolated, so that we are able to prune the L_0 operations. Then we commute $fetch_1(rid_2)$ belonging to $select_1('LA')$ as well as $search_1('NYC')$ attached to $select_1('NYC')$ to the left because there is no conflict on that way, which results in the transaction nodes t_1 and t_2 being isolated.

Exercise 6.3 :

Consider the two non-layered schedules of Figure 6.2. Are these tree reducible?

a) We first consider the schedule depicted in Figure 5. At the level L_0 we can isolate $Withdraw_2(b)$ by pushing $r_2(q)$ to the right behind $w_1(t)$. All L_1 operations are isolated and we can prune the operations at the level L_0. Then we put $Deposit_1(c)$ between $Withdraw_1(a)$ and $Withdraw_2(b)$. Thus we obtain the serializaion $t_1 <_s t_2$.

b) The schedule depicted in Figure 6 is also tree reducible, which can be verified by observing that we can, analogously to the previous schedule, isolate both withdrawals, but we don't prune the corresponding subtrees so far. Then, we push $Withdraw_1(a)$ such that its subtree is entirely placed between $w_2(r)$ and

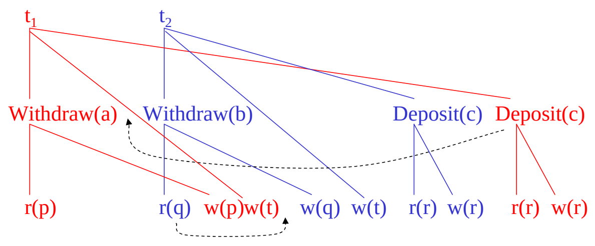


Figure 5: Transformation of 1st schedule in Figure 6.2

$r_1(r)$, which is allowed because no page level conflicts occur on this way. This way we isolate t_1 and t_2 with the serialization order $t_2 <_s t_1$.

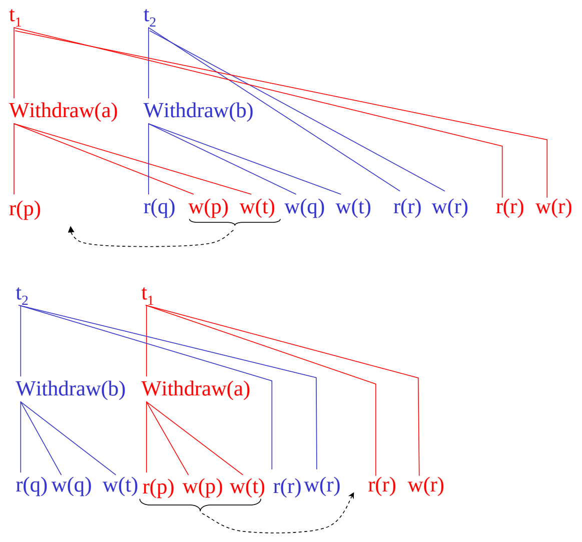


Figure 6: Transformation of the 2nd schedule in Figure 6.2

Note that in the first example both orderings of t_1 and t_2 are acceptable, whereas only the one discussed in b) is legal for the second schedule; this happens due to the fact that we don't know the "L_1" nature (semantics) of reads and writes that directly originate from the transaction nodes.

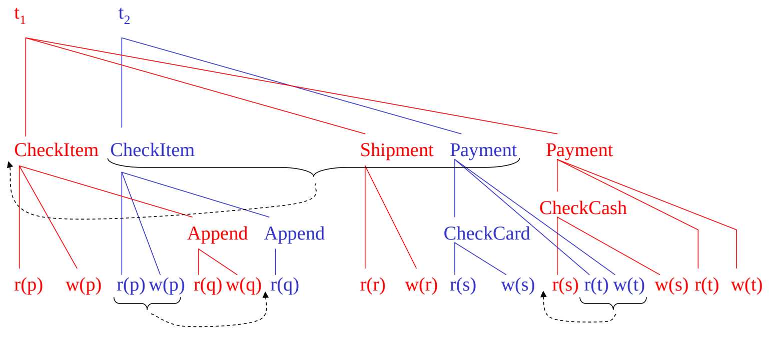


Figure 7: Schedule for Exercise 6.4

Exercise 6.4 :

Consider again the example of Figure 6.6. Give the necessary transformation steps, using the commutativity and the tree pruning rules, for proving that the schedule is tree reducible.

The solution to this exercise is shown in Figure 7. First we can isolate *CheckItem* operations by moving $r_2(p)w_2(p)$ to the right such that they immediately follow the $w_1(q)$ operation that belongs to $Append_1$. The *Payment* operations can be isolated by exchanging the order of $r_2(t)w_2(t)$ and $r_1(s)$. All L_2 operations are already isolated at that time, so we can prune the corresponding subtrees. In order to isolate the transaction nodes, we have to commute $CheckItem_2$ and $Payment_2$ to the left thus obtaining the serialization $t_2 <_s t_1$.

Exercise 6.6 :

Consider a counter object with operations with lower and upper bounds as constraints. The counter interface supports three operations: *Increment* adds a specified value Δ to the counter and returns "OK" if the upper bound is not exceeded; otherwise it returns "No" and leaves the counter unchanged. Analogously, *decrement* subtracts a given value from the counter or returns "No" if the lower bound was violated. Finally, *getvalue* returns the counter value. Give a return value commutativity table for these three operations. Discuss possible improvements of concurrency by taking into account explicit constraints among operation parameters.

For convenience let us abbreviate *increment, decrement* and *getvalue* as *inc, dec* and *gv* respectively. The return value commutativity is given in the following table.

	$inc \uparrow ok$	$inc \uparrow no$	$dec \uparrow ok$	$dec \uparrow no$	$gv \uparrow res$
$inc \uparrow ok$	$+$	$-$	$-$	$+$	$-$
$inc \uparrow no$	$+$	$+$	$-$	$+$	$+$
$dec \uparrow ok$	$-$	$+$	$+$	$-$	$-$
$dec \uparrow no$	$-$	$+$	$+$	$+$	$+$
$gv \uparrow res$	$-$	$+$	$-$	$+$	$+$

To address the second issue of this exercise, let us investigate the nature of conflicts between decrements and increments. Many additional operation pairs would be commutable (i.e., not in conflict), if we knew the counter's initial value x_0:

- if $x_0 + \Delta_2 \leq upper\_bound$, then $dec_1(x, \Delta_1) \uparrow ok \;\; inc_2(x, \Delta_2) \uparrow ok$ are commutable.
- if $x_0 + \Delta_2 - \Delta_1 \leq lower\_bound$, then $dec_1(x, \Delta_1) \uparrow no \;\; inc_2(x, \Delta_2) \uparrow ok$ are commutable.
- if $x_0 + \Delta_2 \geq upper\_bound$, then $inc_1(x, \Delta_1) \uparrow ok \;\; inc_2(x, \Delta_2) \uparrow no$ are commutable.
- if $x_0 - \Delta_2 \geq lower\_bound$, then $inc_1(x, \Delta_1) \uparrow ok \;\; dec_2(x, \Delta_2) \uparrow ok$ are commutable.
- if $x_0 - \Delta_2 + \Delta_1 \geq upper\_bound$, then $inc_1(x, \Delta_1) \uparrow no \;\; dec_2(x, \Delta_2) \uparrow ok$ are commutable.
- if $x_0 - \Delta_2 \leq lower\_bound$, then $dec_1(x, \Delta_1) \uparrow ok \;\; dec_2(x, \Delta_2) \uparrow no$ are commutable.

These considerations influenced the *escrow locking* method, which is studied in Section 7.6 of the book.

Exercise 6.7 :

Construct a sample schedule with operations on FIFO queues that demonstrate the higher concurrency that return-value commutativity can achieve over general commutativity.

The following schedule

$enq_1(q,\ x) \uparrow one\ enq_1(q,\ y) \uparrow ok\ deq_2(q) \uparrow ok\ enq_1(q,\ z) \uparrow ok\ deq_2(q) \uparrow ok$
$deq_2(q) \uparrow ok$

can be transformed into the equivalent serial schedule $t_1 t_2$ using return-value commutativity and may therefore be allowed by a scheduler, while it is not serializable using general commutativity.

Exercise 6.8 :

Construct the return value commutativity table for a semi-queue. Show, by means of sample schedules, that semi-queues allow higher concurrency than FIFO queues.

The key difference between regular queues and semi-queues is that order of enqueue and dequeue operations doesn't matter, unless the queue becomes empty. Therefore, enqueues and dequeues can always be executed without conflict if the queue is not empty, so semi-queues allow a much higher concurrency. The following return-value commutativity table for semi-queues reflects this (combinations that cannot occur are marked as *imp*, i.e. impossible):

	$enq \uparrow ok$	$enq \uparrow one$	$deq \uparrow ok$	$deq \uparrow empty$
$enq \uparrow ok$	+	imp	+	imp
$enq \uparrow one$	−	imp	−	imp
$deq \uparrow ok$	+	−	+	−
$deq \uparrow empty$	imp	−	imp	+

The schedule given in the next exercise shows the impact of the semi-queue semantics on concurrency.

Exercise 6.9 :

Consider the following execution of operations on an initially empty queue q where a, b, and c are entries added to the queue:

$$enq_1(q,a)enq_2(q,b)deq_3(q)enq_1(q,c)deq_3(q)$$

Discuss whether this schedule is serializable assuming (a) general commutativity, (b) return-value commutativity, and (c) return value commutativity for semi-queues with nondeterministic selection of entries by Dequeue operations. In the last setting, which entries should be selected by the two dequeue operations of t_3 to produce an intuitively correct execution?

First, we add return values to the schedule, so that we can use return-value commutativity:

$enq_1(q,a) \uparrow one\ \ enq_2(q,b) \uparrow ok\ \ deq_3(q) \uparrow ok\ \ enq_1(q,c) \uparrow ok\ \ deq_3(q) \uparrow ok$

Obviously, this schedule is not serializable if we assume general commutativity. For example, it is not possible to commute t_1's last enqueue operation outside of t_3's operations.

The schedule isn't return-value serializable, either: While it is possible to isolate t_3's operations at the end of the schedule, we have t_2's enqueue operation in between t_1's enqueue operations and can't commute it outside.

However, assuming semi-queue semantics, all operations are commutative with each other except the first enqueue, so we get the following equivalent serial schedule:

$$enq_1(q,a) \uparrow one \;\; enq_1(q,c) \uparrow ok \;\; enq_2(q,b) \uparrow ok \;\; deq_3(q) \uparrow ok \;\; deq_3(q) \uparrow ok$$

(Note that the schedule $t_1 t_3 t_2$ is not equivalent, because t_2's enqueue would return *one* and not *ok* as it should.)

Exercise 6.10 :

Design an abstract data type "mailbox" with operations like *Send* (i.e., add a message to the mailbox), *Receive* (i.e., extract a message from the mailbox), and so on. Discuss the operations' commutativity relation, for (a) general (i.e., state independent) commutativity and (b) return value commutativity. Devise an implementation of mailbox objects in terms of queues, records, and ultimately pages. Sketch the resulting transaction trees. Give an example of a concurrent execution, with as much concurrency as possible, for two transactions where one sends two messages to two different mailboxes and the other receives all messages from one of the two mailboxes.

In the scenario considered in this exercise we assume that all messages stored in a mailbox get a unique identifier *msgid* based on a timestamp of the message arrival.

Two *Send* operations do not commute because the order in that the messages arrive might be relevant. For example, when several persons apply for a job, and two or more of them have identical qualification, then the order in which applications have been submitted will be one of the possible decision criteria. The *Send* and *Receive* are clearly not exchangeable, if the mailbox is empty prior to *Send* or if it is full before *Receive*. Two *Receive* operations are not commutable as well, because this is exactly a situation in which the order determines, whether we may see junk mails or not as explained in the following.

At first sight you might be surprised why may we need concurrent *Recieve* operations on the same mailbox at all. Imagine that you have a conventional interactive mail client and also an automatic program periodically scheduled on the mail server. The task of the latter is to remove junk messages from all mailboxes.

a) These considerations lead us to the following state independent commutativity table.

	Send	*Rec*
Send	−	−
Rec	−	−

b) A possible state dependent commutativity table follows. Note that *Receive* (abbreviated as *Rec*) can either return *ok* or it returns *one* signalling that this was the only message in the mailbox. This is relevant only for the scheduler for testing commutativity; at the application layer *one* is perceived just as normal *ok*. If the mailbox is empty, *Receive* returns *no*. If the mailbox is full, then no other message can be appended to it. In such a situation *Send* returns *no*.

	$Send \uparrow ok$	$Send \uparrow no$	$Rec \uparrow ok$	$Rec \uparrow one$	$Rec \uparrow no$
$Send \uparrow ok$	−	−	+	−	imp
$Send \uparrow no$	+	+	−	−	+
$Rec \uparrow ok$	+	+	−	−	−
$Rec \uparrow one$	+	+	imp	imp	−
$Rec \uparrow no$	−	+	imp	imp	+

Now we are ready to model the operations, mapping mailbox operations onto queues and queue operations onto SQL.

- $Send(mb, message) = enq(mbq, message)$
- $Receive(mb) = deq(mbq)$
- $enq(mbq, message) = ins(relation, next\_id, message)$

- $deq(mbq) = del(relation, lowest\_id)$
- $ins(relation, next\_id, message) = r(p)w(p)$
- $del(relation, lowest_id) = r(p)w(p)$

Figure 8 shows a sample schedule on a mail server. This schedule is tree reducible if the state dependent commutativity defined above is used. First we can isolate ins_2 and del_1 by commuting $r_1(p)$ and $w_2(p')$, so that we are able to subsequently prune the L_2 operations. Then we commute the first Rec_1 to the right and obtain this way the serialization $t_2 <_s t_1$

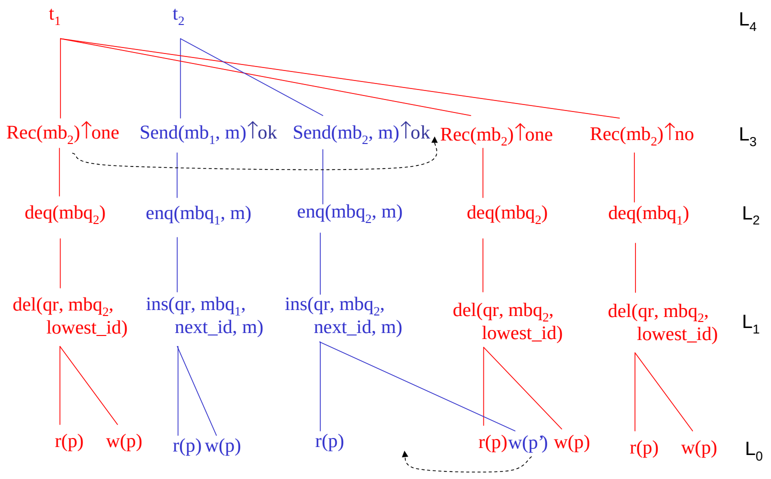


Figure 8: Sample tree reducible schedule

Exercise 6.11 :

Even if two operations do not commute in the strict sense, the different effects that result from the two possible orderings may be considered as "effectively equal" from an application viewpoint. We could then simply declare the two operations as *compatible* and treat them as if they were commutative. An example would be a Deposit operation together with a new style of Withdraw operation that allows overdrafting but claims a penalty of, say, \$10 each time the account balance is below the specified threshold and money is withdrawn. So the code for this relaxed variant of withdraw would be

```
relaxed withdraw (x, Δ):
   x.balance := x.balance - Δ;
   if x.balance < 0 then
     x.balance := x.balance - 10 fi;
```

Give a sample schedule with these operations that is conflict serializable relative to the compatibility of the operations, but is not conflict serializable under a strict interpretation of commutativity. Discuss the acceptability of the resulting (worst-case) situations from an application viewpoint, of both the bank and the customer.

Assume we have two accounts a and b, and two transactions t_1 and t_2. Initial balance of a is \$ 1000, and that of b is \$ 0. Transaction t_1 moves \$ 1000 from a to b, and transaction t_2 moves \$ 1000 from b to a, so that executing t_1 first and then t_2 results in a do-nothing operation. The following schedule is valid using compatibility (and is not allowed using commutativity):

$$Withdraw_1(a, 1000)Withdraw_2(b, 1000)Deposit_1(b, 1000)Deposit_2(a, 1000)$$

The effect of this schedule is no longer a do-nothing operation, but account b ends up with a balance of \$-10. While the bank would easily accept such an anomaly, a customer may not at all be happy with this. (Note that if the system used commutativity, t_2 would be blocked until the commit of t_1, and a real do-nothing operation would be the effect.)

Chapter 7 - Solutions to Exercises

Exercise 7.1 :

Investigate what kind of deadlocks can arise in a layered system with layered 2PL, and how they can be detected and eliminated. Under what conditions is it sufficient that each layer tests only for "local" deadlocks caused by lock waits of one layer only? Under what conditions is it necessary to consider lock waits of all layers "globally" to detect all deadlocks?

First consider layered schedules. When a transaction starts executing an operation at level L_i, it first acquires a corresponding lock at this level. At that time the transaction does not hold any locks at lower levels. Then the L_i operation issues requests at level L_{i-1} (i.e., it starts a subtransaction). Upon completion of the subtransaction all locks at level L_{i-1} are freed, while the lock at level L_i is still held until the operation at the level above is finished. Based on this observation and using an induction argument we can conclude that when a transaction is blocked at level L_i, it doesn't hold any locks at lower levels, so it cannot block any other transaction beneath L_i.

Now assume that we have a "global" deadlock, such that t_1 waits for t_2 at L_{i_1}, t_2 waits for t_3 at L_{i_2} and so on, and finally t_n waits for t_1 at L_{i_n}, with $n \geq 2$. Thus, t_2 holds a lock at L_{i_1} and requests a lock at L_{i_2}. That is, we have $i_1 \geq i_2$. By induction, we get $i_1 \geq i_2 \cdots \geq i_n \geq i_1$. This implies that all locks involved in that deadlock appear at the same level. So we showed that no cross-level deadlock testing is necessary.

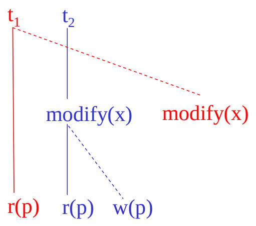


Figure 1: "Global" deadlock in a non-layered schedule

If we allowed non-layered schedules, we would however have to consider "global" deadlocks. Consider a two-level architecture with a record and a page level. The schedule given in Figure 1 shows an example schedule, in which a cross-level deadlock occurs. Transaction t_1 first reads a page p (after the corresponding read lock has been granted). Then t_2 executes a record-level operation $modify(x)$ operation that in turn reads and tries to write the page p. The write operation $w(p)$ is blocked when requesting a write lock for p. After that, t_1 tries to execute $modify(x)$, which is in conflict with t_2's modify operation. Hence, t_1 is blocked by t_2 at the record level, while it still holds its lock at the page level which prevents t_2 from making any progress.

Exercise 7.2 :

Discuss the pros and cons of the selective layered 2PL. As an example, consider query-, record-, and page-level operations along the lines of Figure 7.4. Does having all three levels involved in locking rather than only two result in performance advantages? If locking should be limited to two levels, are there arguments for choosing certain levels over others?

It is clear that transactions hold locks shorter (at a particular level) if there are locks at another level above it. Locks can be released after the parent operation is finished, and they need not be held until the end of the transaction. Obviously, this is a useful property, so it is better to have multiple levels.

However, from a practical point of view, each level requires its own lock management, which adds overhead to the overall system.

In the example, we have the choice between query-level locks and record-level locks (page level locks are always required). If we choose to acquire record-level locks, these locks must be held until the end of the transaction, while page-level locks can be released as soon as a record-level operation is finished. Other transactions accessing objects that reside on these pages, but are not in conflict with those accessed by the first transaction, can therefore resume their work quickly. When we use query-level locks, page-level locks are held much longer (especially for long queries like table scans), so concurrent transactions may be blocked longer.

So the above consideration suggests that it is more beneficial to use locks at the record level rather than at the query level.

However, there are also scenarios where the opposite is true, namely, when decision-support queries include complex search predicates. In such cases locking at the record level, which includes locks on individual index keys (see Chapter 9), may end up being unnecessarily restrictive and thus hampering concurrency. As an example consider a query of the form:

```
select name from persons
where city='Seatle' and age=29
```

With locking at the query level (see Chapter 8, predicate locking) this query would not conflict with statements like:

```
insert into persons(name, city, age)
values('Jennifer', 'Seattle', 22)
```

or

```
insert into persons(name, city, age)
values('Janis', 'Austin', 29)
```

With locking at the record level, however, the first insert statement would exhibit conflict a lock conflict on the index key `'Seattle'` of the city index, and the second query would exhibit a lock conflict on the index key `29` of the age index.

So in this case, where the second predicate of the query is a conjunction of two conditions, record level locking has the negative effect of effectively locking the disjunction of the two conditions, simply by locking the index keys for both conditions. Higher-level predicate locking would thus gain concurrency in such scenarios.

Exercise 7.3 :

Trace the lock requests, conflict tests, lock acquisitions, and lock releases for the execution of the schedule in Figure 7.7 under the general object model 2PL with an SS2PL variant. How would the sample execution change if the Append operations were dropped and t_2's decrement operation were replaced by an incr(a) operation that commutes with other increments? How would the execution change if t_2 started first and t_1 was spawned after all of t_2's operations but before the commit of t_2?

Figure 2 shows a sample execution of the schedule under the general object model 2PL. First the schedule is executed as is until t_2 tries to acquire a *Decr* lock on object a. This lock, however, is not granted, for t_1 still holds the retained *Incr* lock on a and the ancestors of $Decr_2(a)$ and $Incr_1(a)$ either have not terminated or do

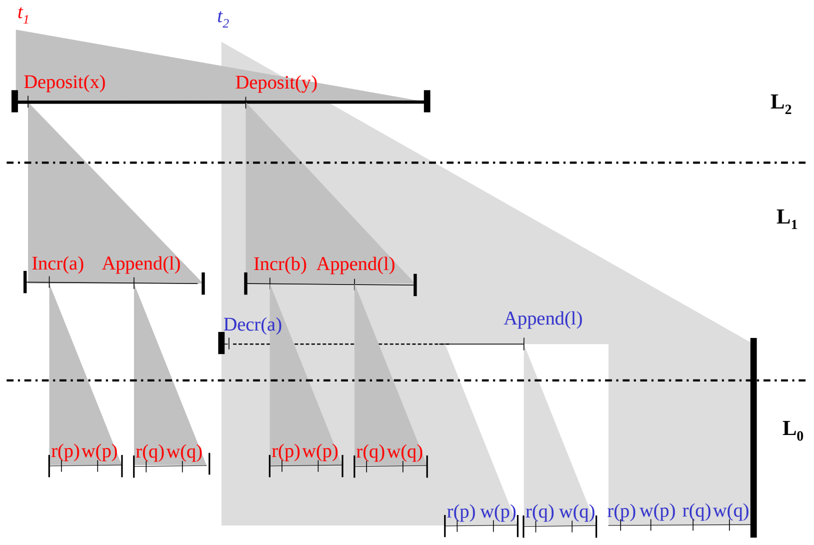


Figure 2: Sample execution of the original schedule

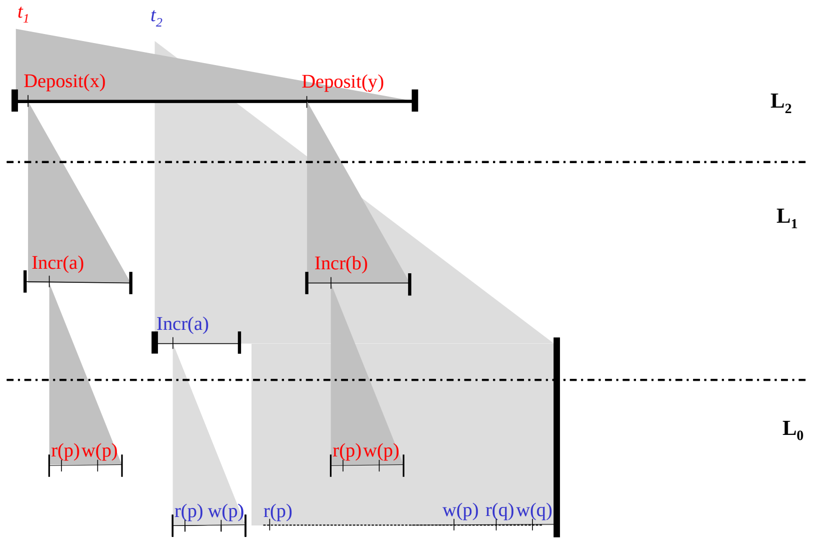


Figure 3: Sample execution with the first modification

not commute. This implies that the scheduler can resume the execution of t_2 only after t_1 is finished, which results in a serial schedule s with $t_1 <_s t_2$.

Figure 3 shows a sample execution with the first modification of the schedule (dropping the *Append* operation and and replacing $Decr_2(a)$ by $Incr_2(a)$) . When t_2 starts, it first tries to obtain an *Incr* lock on a. This lock is granted because now there is no conflict between the two increment operations. Then t_2's *Incr* successfully obtains read and write locks on p, since only t_1's increment holds conflicting lock on p, they are in retained mode, and the increments themselves are commutable by definition. After t_2 completes the execution of $Incr(a)$ it tries to acquire a read lock on p for the direct access. This lock is not granted because t_1 is still running. From now on t_2 has to wait for the end of t_1. So the scheduler resumes the execution of t_1, which now successfully acquires an *Incr* lock on b. $Incr_1(b)$ in turn obtains all required page level locks, because the corresponding conflicting locks are in retained mode and they are held by $Incr_2(a)$, which clearly commutes with $Incr_2(b)$. After

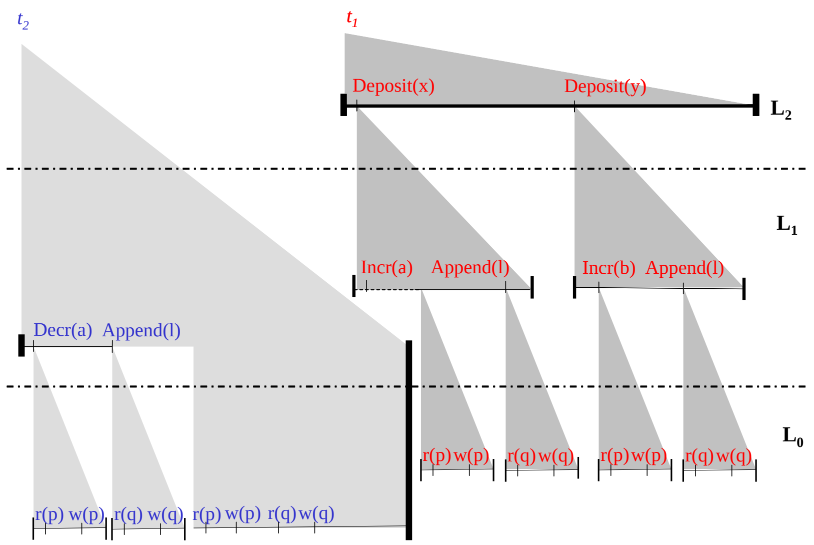


Figure 4: Sample execution with the second modification

t_1 is finished and t_2 can resume execution.

Finally Figure 4) shows the schedule with the second modification (with t_2 started first). When the scheduler starts the execution of t_1, t_2 has not yet released any locks. Thus, whereas t_1 successfully obtains a *Deposit* lock, the corresponding *Deposit* operation of t_1 cannot acquire an *Incr* lock on a, because the conflicting *Decr* held by t_2 lock is still active due to the SS2PL protocol. Therefore t_1 has to wait for the commit of t_2, upon which all its locks are released.

Exercise 7.4 :

Develop pseudocode for a restricted version of the object model 2PL, where a generalized form of two-level schedules is considered, with some transactions bypassing the object level L_1 and invoking L_0 operations directly. This setting is, for example, of interest in the context of federated information systems, where global transactions access a variety of databases through a federation layer, using commutativity based 2PL for high-level operations exported by the underlying databases, but local transactions are still allowed to access a database directly. Such an architecture is sometimes called a federation of “autonomous” databases.

Transactional federations are treated in detail in Chapter 18.

The following discussion is based on:

> A. Deacon, H.-J. Schek, G. Weikum:
> *Semantics-based Multilevel Transaction Management in Federated Systems*, International Conference on Data Engineering, Houston, TX, USA, 1994.

When an object-level operation of a global transaction (a global subtransaction) is finished, its page-level locks are not released, but converted into “retained” mode. Retained locks are compatible with locks of other global transactions (so they don’t block them), but local transactions must respect them. Upon the commit (or rollback) of the global transaction, all its retained page-level locks (and its object-level locks) are released.

In order to implement this approach in a system, we have to distinguish between local and global subtransactions, and we must know about the commit (or abort) of a global transaction. We introduce two new lock modes *retained read* and *retained write*.

We use the following lock-compatibility matrix ('gr' means read lock request by global transaction, 'lw' means write lock request by local transaction, and so on):

	gr	gw	lr	lw
read	+	−	+	−
write	−	−	−	−
retained read	+	+	+	−
retained write	+	+	−	−

In addition, global transactions acquire high-level locks at the "semantic" object level.

The following pseudocode shows the necessary steps.

```
BeginGlobalTransaction(GlobalTransID) { add global transaction to
list of global transactions; }

BeginGlobalSubtransaction(GlobalTransactionID, SubtransactionID)
{ add subtransaction to global transaction
      in list of global transactions;
  acquire object-level lock;
}

AcquireGlobalLock(GlobalTransID,SubtransactionID,page,mode)
{ check if mode is compatible with currently held locks
        for this page;
  if so, grant the lock (and add it to the necessary lists);
  if not, update the waits-for graph and wait;
}

EndGlobalSubtransaction(GlobalTransactionID, SubtransactionID)
{ convert all locks of this subtransactions into retained locks;
  delete it from the necessary lists;
}

EndGlobalTransaction(GlobalTransactionID)
{ release all locks of this transaction;
  delete it from the list of global transactions;
}

BeginLocalTransaction(LocalTransID)
{ add local transaction to list of local transactions; }

AcquireLocalLock(LocalTransID,page,mode)
{ check if mode is compatible with currently held locks
        for this page;
  if so, grant the lock (and add it to the necessary lists);
  if not, update the waits-for graph and wait;
}

EndLocalTransaction(LocalTransactionID)
{ release all locks of this transaction;
  delete it from the list of local transactions;
}
```

Exercise 7.5 :

Apply the following hybrid protocols to the schedule of Figure 7.3 in a selective way so that only two levels are involved in concurrency control:

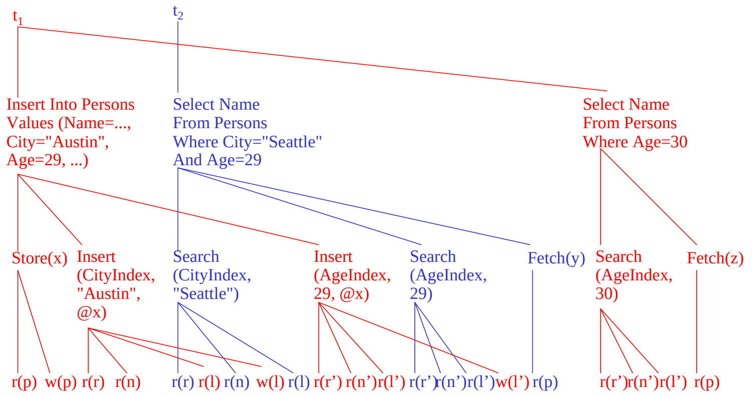
a) the forward-oriented optimistic concurrency control (FOCC, see Chapter 4) at the page level, and strong 2PL at the record level,

b) the forward-oriented optimistic concurrency control at the page level, and strong 2PL at the query level,

c) the hybrid multiversion protocol ROMV (see Chapter 5) at the page level, and strong 2PL at the record level,

d) the hybrid multiversion protocol ROMV at the page level and strong 2PL at the query level.

Sketch the resulting executions.



t1
t2
Insert Into Persons Values (Name=..., City="Austin", Age=29, ...)
Select Name From Persons Where City="Seattle" And Age=29
Select Name From Persons Where Age=30
Store(x)
Insert (CityIndex, "Austin", @x)
Search (CityIndex, "Seattle")
Insert (AgeIndex, 29, @x)
Search (AgeIndex, 29)
Fetch(y)
Search (AgeIndex, 30)
Fetch(z)
r(p) w(p) r(r) r(n) r(r) r(l) r(n) w(l) r(l) r(r') r(n') r(l') r(r') r(n') r(l') w(l') r(p) r(r') r(n') r(l') r(p)

Figure 5: Corrected schedule from Figure 7.3

The original schedule (Figure 7.3 in the book) is shown in Figure 5.

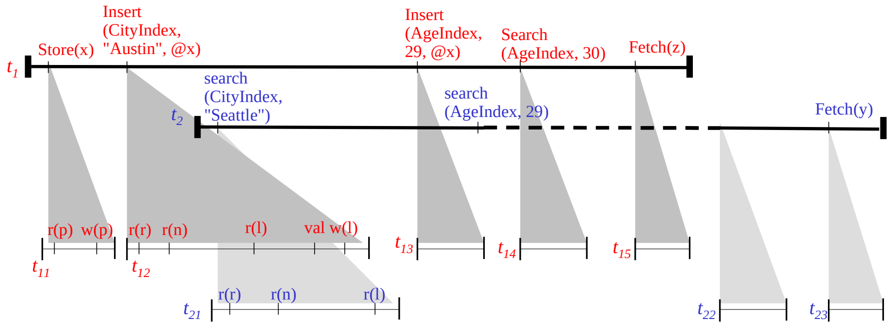


Figure 6: FOCC at the page level and SS2PL at the record level

Figure 6 shows the execution for case (a). With this setting given in (a) the schedule is executed as is until t_2 tries to execute `Search(AgeIndex, 29)`, the lock for which cannot be granted immediately, since t_1 still holds the conflicting lock for `Insert(AgeIndex, 29, @x)`. That is, t_2 may be resumed only after the end of t_1.

Figure 7 depicts an output schedule under conditions given in (b). The schedule is executed without any delays until t_1's first subtransaction enters the validation phase, in which it has to be aborted because l is in both $WS(t_{11})$ and $RS(t_{21})$. After t_2 is finished, t_{11} is restarted and t_1 finally comes to an end.

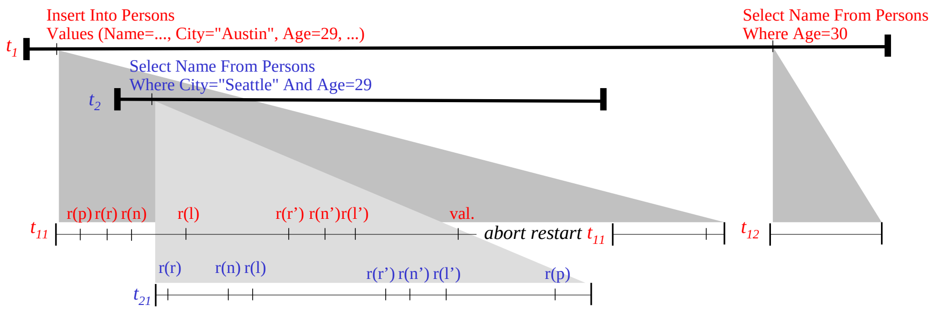

Insert Into Persons
Values (Name=..., City="Austin", Age=29, ...)
Select Name From Persons
Where Age=30
Select Name From Persons
Where City="Seattle" And Age=29
t1
t2
t11
t12
t21
r(p) r(r) r(n) r(l) r(r') r(n')r(l') val.
abort restart t11
r(r) r(n) r(l) r(r') r(n') r(l') r(p)

Figure 7: FOCC at the page level and SS2PL at the query level

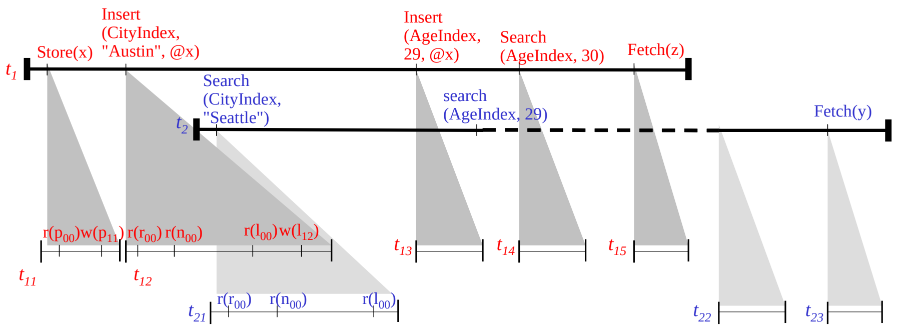


Figure 8: ROMV at the page level and SS2PL at the record level

Figure 8 shows the situation given in (c), which is very similar to (a) because of the dominating influence of SS2PL at the record level.

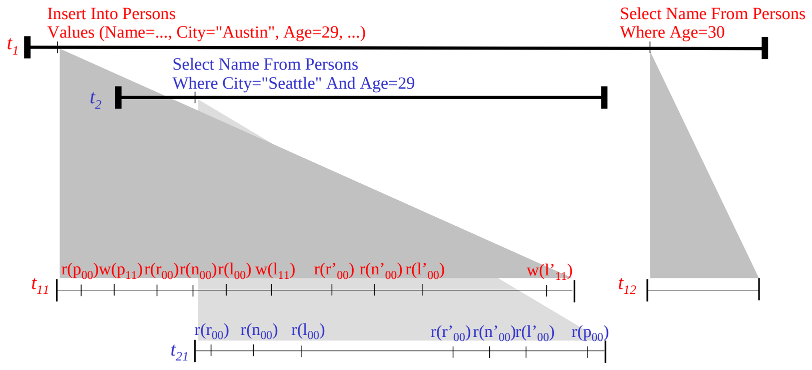

Insert Into Persons
Values (Name=..., City="Austin", Age=29, ...)
Select Name From Persons
Where Age=30
Select Name From Persons
Where City="Seattle" And Age=29
t1
t2
t11
t12
t21
r(p00)w(p11)r(r00)r(n00)r(l00) w(l11) r(r'00) r(n'00) r(l'00) w(l'11)
r(r00) r(n00) r(l00) r(r'00)r(n'00)r(l'00) r(p00)

Figure 9: ROMV at the page level and SS2PL at the query level

In contrast to the situation in (b) the schedule with the setting given in (d) can be executed without any subtransaction aborts as shown in Figure 9. As you see the read-only transaction t_{21} largely benefits by using ROMV at the page level because it simply reads last committed page versions without having to acquire read locks.

Exercise 7.6 :

Prove the correctness of the 2PL protocol for two-level schedules with deferred lock conflict testing based on return value commutativity sketched in Section 7.6.

Consider an arbitrary two-level schedule s which has been produced by an appropriate scheduler. Since all subtransactions, i.e., L_1 operations, acquire page level write and read locks according to the 2PL protocol, we know that L_1-to-L_0 schedule is in OCSR (Theorem 4.2). So regardless of whether some L_1 operations had to be rolled back or not, the output schedule looks as if all transactions were able to obtain their L_1 locks according to the 2PL prior to the execution of the L_1 operations. This implies that the L_2-to-L_1 schedule is in OCSR as well. Thus, by Theorem 6.2 the schedule s is tree reducible.

Exercise 7.7 :

Trace the intermediate states of the escrow data structure (i.e., the infimum and supremum values) for the counter object x in the example execution of Figure 7.9.

	x.inf	x.sup	status
	100	100	
$decr_1(x, 75)$	25	100	
$decr_2(x, 10)$	15	100	
$incr_3(x, 50)$	15	150	
c_1	15	75	
$decr_4(x, 20)$			waits!
a_2	25	75	
$decr_4(x, 20)$	5	75	
c_3	55	75	
c_4	55	55	

Exercise 7.8 :

Consider two counter objects x and y, with initial values $x = 100$ and $y = 50$. Both counters have zero as a lower bound and no upper bound. Apply the escrow locking method to the following schedule of three transactions one of which aborts: $decr_1(x, 60) incr_2(x, 20) incr_1(x, 10) decr_3(x, 50) decr_2(y, 60)$
$incr_2(x, 20) a_2 decr_1(y, 10) c_1 c_3$.

	x.inf	x.sup	y.inf	y.sup	status
	100	100	50	50	
$decr_1(x, 60)$	40	100	50	50	
$incr_2(x, 20)$	40	120	50	50	
$incr_1(x, 10)$	40	130	50	50	
$decr_3(x, 50)$					waits!
$decr_2(y, 60)$					forced to abort!
a_2	40	110	50	50	
$decr_1(y, 10)$	40	110	40	50	
c_1	50	50	40	40	
$decr_3(x, 50)$	0	50	40	40	
c_3	0	0	40	40	

Exercise 7.9 :

Escrow operations can be further generalized into conditional increment and decrement operations, where the conditions may be stronger than the actual resource quantity needed, taking the form:

```
conditional_decr (x, $\epsilon$, $\Delta$):
  if x $\geq \epsilon$ then x := x - $\Delta$ fi;
```

and an analogous *Conditional_Incr* operation. For the *Conditional_Decr*, the value of the ϵ parameter would be larger than or equal to the operation's Δ value, and the situation would be the other way around for the *Conditional_Incr* operation.

In addition to such operation-specific conditions, the global invariants for $x.low$ and $x.high$ must be satisfied at all times. Discuss the notions of general and return value commutativity for these generalized escrow operations.

The intuition behind the conditional operations is to introduce 'private' bounds for each operation. If we set $\epsilon := x.low + \Delta$ in a *Conditional_Decr* operation, we get exactly the same semantics as for the operations without the extra condition.

Let's consider *Conditional_Decr* with arbitrary value of ϵ (*Conditional_Incr* is analogous). We can generalize the escrow locking method as follows. In the escrow test, we have to compare the current infimum $x.inf$ of the counter with ϵ. If $x.inf \geq \epsilon$, we can execute the decrement and update $x.inf$ and x. If $x.inf < \epsilon$, but $x.sup \geq \epsilon$, we have a chance that the value of the counter increases after some transactions commit or abort, so we wait. Otherwise, the transaction is aborted.

Obviously, conditional operations are not generally commutative, but we can apply the escrow techniques. If $\epsilon > x.low + \Delta$, we impose stricter conditions on the counter's value, so we allow a lower degree of concurrency compared to the setting $\epsilon := x.low + \Delta$ (i.e., the original unconditional decrement). However, there are applications, which require this stricter semantics, for example, in sophisticated warehouse administration.

Chapter 8 - Solutions to Exercises

Exercise 8.1 :

Discuss how predicate locking can be extended to disjunctive conditions such as queries of the form

```
SELECT Name FROM Emp
WHERE Position='Manager' OR Department='Research'.
```

Also discuss how join queries such as

```
SELECT Emp.Name, Emp.Department FROM Emp, Dept
WHERE Emp.Position='Manager' AND Dept.City='Toronto'
AND Emp.Department=Dept.Department
```

could be (conservatively) handled by predicate locking.

A general way to extend predicate locking to arbitrary Boolean expressions (and thereby including disjunctions as a special case) is the following: each Boolean expression is converted to its equivalent disjunctive normal form (DNF). For each of the monoms in this DNF, a predicate lock is acquired. Each tuple that satisfies the original expression satisfies at least one of these monoms, so locking the monoms is sufficient to lock all relevant tuples. However, as we split the process of acquiring one "complex" lock into acquiring several "simpler" locks, we increase the likelihood of deadlocks. For example, imagine that two concurrent transactions try to lock overlapping complex predicates. Both transactions may be able to lock some monoms, but each may become blocked by the other transaction before it can acquire locks for all monoms. If they acquired only a single complex lock, only one would be blocked.

A join query is difficult to handle because its query predicate contains logic conditions on attributes of multiple tables. Because we acquire locks for one table at a time only, we have to build a conservative approximation about which tuples of each table could be hit by a join. For each table, we lock that part of the predicate that concerns attributes of this table (like `Emp.Position='Manager'` in the example), and disregard the join condition, because we have no knowledge about join partners in advance. Thus, we probably lock more records than really needed, but all of the relevant records are definitely locked. If a join query contains only join conditions and no other filter conditions, we have to lock the entire tables (but such queries are infrequent in online transaction systems).

Exercise 8.2 :

The sample histories we studied for IDM transactions in this chapter mostly referred to a database with one relation and one attribute. Show the following: If the database has more than one relation, the serializability of each projection of a given history onto an individual relation does *not* imply serializability of that history in general. Use the following scenario as a counterexample: Let database schema D contain relation schemata R and S, where R has attributes A and B, and where S has attributes B and C, and consider the following transactions and history:

$t_1 = m^R(A = 1; A = 2)m^S(C = 5; C = 6)$
$t_2 = m^R(A = 2; A = 3)m^S(C = 4; C = 5)$
$s = m_1^R(A = 1; A = 2)m_2^R(A = 2; A = 3)m_2^S(C = 4; C = 5)m_1^S(C = 5; C = 6)$

Assume that the initial attribute values are $A = 1$, $B = 0$, $C = 4$. Then we obtain $eff(t_1; t_2) = \{A = 3, B = 0, C = 5\}$, and $eff(t_2; t_1) = \{A = 2, B = 0, C = 6\}$. However $eff(s) = \{A = 3, B = 0, C = 6\}$, which does not match any of the two possible serialization effects. The projections onto the individual relations are trivially serial but they correspond to different serialization orders.

Exercise 8.7 :

Consider the transactions from Example 8.13 again. Determine whether decomposing t_{11} any further still results in a correct chopping.

If we decompose t_{11} into $t_{111} = r_{111}(x)$ and $t_{112} = w_{112}(x)$, then we will immediately obtain an *sc* cycle, because both of these two resulting transactions contain an operation in conflict with t_2's $w_2(x)$, i.e., they are both connected by a c edge to t_2. And they are also connected by an s edge with each other. Thus, further chopping of t_{11} does not make sense.

Exercise 8.8 :

Consider the following transactions:

$$t_1 = r_1(x)w_1(x)r_1(y)w_1(y)$$

$$t_2 = r_2(x)$$

$$t_3 = r_3(y)w_3(y)$$

Try to decompose t_1 into three pieces s.t. the result is a correct chopping.

It is quite natural to chop t_1 into two pieces, one piece for the operations on each of the data objects x and y. To get three pieces, we have to decompose one of these parts into two new parts consisting of only one operation. The best candidate is obviously the part dealing with x, because the other transactions only read x.

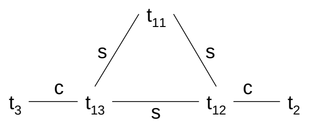


Figure 1: Chopping graph

Hence, consider the chopping of t_1 into $t_{11} = r_1(x)$, $t_{12} = w_1(x)$, and $t_{13} = r_1(y)w_1(y)$. This is indeed a correct chopping, because its chopping graph, which is depicted in Figure 1, contains no *sc* cycles.

Exercise 8.9 :

Prove Theorem 8.5.

What we need to show is: if the chopping graph is acyclic then any possible interleaving within the given set of chopped transactions results in an acyclic conflict graph having the original transactions as nodes. Remember, we have assumed that the proper ordering of transaction pieces (i.e., s siblings) is self-guaranteed.

Consider a set of chopped transactions $T = \{t_{11}, \ldots, t_{1k_1}, \ldots, t_{n1}, \ldots, t_{nk_n}\}$ for which we know its chopping graph is acyclic. Assume that there is a schedule s that is not in CSR with regard to the complete transactions, but s is in CSR and thus, without loss of generality, serial with regard to the chopped transactions. So there are m transactions involved in a cycle (say $t_i \rightarrow \ldots \rightarrow t_m \rightarrow t_i$). We know, however, that only conflict serializable schedules are generated with regard to the transaction pieces. This implies that at least for one transaction two transaction pieces are touched (without loss of generality t_{ik} and t_{il}) by the cycle. That is, on the one hand we have an s edge between t_{ik} and t_{il}. On the other hand each directed edge corresponds to a c edge between transaction pieces containing conflict operations: $t_{ik} -_c \ldots -_c t_{mj} -_c t_{il}$. Thus, the chopping graph contains an *sc* cycle. This is a contradiction!

Exercise 8.10 :

Show that if a set of chopped transactions contains an sc cycle, any further chopping of any of the transactions will not render the graph acyclic.

An *sc* cycle contains at least one c edge and one s edge. Any two nodes in a chopping graph can be connected either by an s edge or by a c edge but not by both at the same time. This implies there are at least three nodes involved in the cycle: say t_1, t_2 and t_3. Without loss of generality the cycle looks as follows:

$$t_1 -_c t_2 -_{[s|c]} \ldots -_{[s|c]} t_3 -_s t_1$$

The following cases may occur by further transaction chopping:

(a) A transaction t_j involved in the cycle by two or more s and only s edges is chopped into $t_{j_1}, \ldots, t_{j_n}$. So this situation looks as follows: $t_i -_s t_j -_s t_k$. Then we obtain a path $t_i -_s t_{j_1} -_s t_k$ which still connects t_i and t_k. Thus, this does not break the cycle.

(b) A transaction t_j involved in the cycle by two or more c and only c edges is chopped into $t_{j_1}, \ldots, t_{j_n}$. So this situation looks as follows: $t_i -_c t_j -_c t_k$. Thus, there are $p_{j_1},\ p_{j_2} \in op(t_j)$ and $q_i \in op(t_i),\ q_k \in op(t_k)$, such that p_{j_1} is in conflict with q_i, and p_{j_2} is in conflict with q_k. Without loss of generality assume $p_{j_1} \in t_{j_1}$ and $p_{j_2} \in t_{j_2}$. Then we obtain a path $t_i -_c t_{j_1} -_s t_{j_2} -_c t_k$ which still connects t_i and t_k. Thus, this does not break the cycle either. Note that t_{j_1} and t_{j_2} may be one and the same transaction.

(c) A transaction t_j involved in the cycle by exactly one c edge and exactly one s edge is chopped into $t_{j_1}, \ldots, t_{j_n}$. So this situation looks as follows: $t_i -_c t_j -_s t_k$. Some $q_i \in op(t_i)$ and some $p_{j_1} \in op(t_j)$ are in conflict. Without loss of generality assume $p_{j_1} \in t_{j_1}$. Then we obtain a path $t_i -_c t_{j_1} -_s t_k$ which still connects t_i and t_k. Thus, this does not break the cycle either.

(d) A transaction t_j connected to the cycle by more than one s or c edges is chopped. In this situation we can immediately apply the argumentation as in cases (a) or (b).

Thus, we showed that in any case which might occur in the process of chopping, an *sc* cycle cannot be broken once it is there.

Exercise 8.11 :

Suppose that a *purchase* program processes the purchases made by a company by adding the value of an item purchased to the *inventory* and by subtracting the money paid for it from *cash.* The application specification requires that the value of *cash* never becomes negative, so that a transaction recording a purchase has to abort if subtracting the money from *cash* would make that value negative. The structure of such a program is given by a parameterized procedure *purchase(i, p)* recording the purchase of item i for price p, whose body can be written as follows:

```
if (p > cash)       then rollback
                    else inventory[i] := inventory[i] + p;
cash := cash – p;
```

Discuss under which conditions and to what extent chopping can be applied to this scenario.

First we need some kind of semantic analysis of the operations in this scenario. Note that two increments on an *inventory* object are commutable. This holds also for two decrements on a *cash* object. But a read operation on a *cash* object like in the test `p > cash` does not commute with a *decrement* on the same object.

It is intuitively clear that checking feasibility of a cash update and the cash update itself have to be executed atomically, so that no chopping is possible, when these two operations are the fist and the last step of a transaction. But we can show this result even formally. Any non-trivial chopping of the purchase procedure into *two* pieces implies that the first piece includes the read operation on *cash*, and the last one contains the cash update. Now consider the chopping graph for two instances, t_1 and t_2, of the chopped *purchase* procedure depicted in Figure 2. We observe an *sc* cycle $t_{11} -_s t_{12} -_c t_{21} -_s t_{22} -_c t_{11}$, and as we have already shown in Exercise 8.10, there is no further chopping which leads to an acyclic chopping graph.

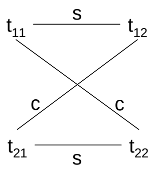


Figure 2: Chopping graph for the purchase scenario

But we can do better, if we slightly rewrite the procedure without changing its overall semantics as you can see below.

```
if (p > cash)                   then rollback
                                else cash := cash – p;
inventory[i] := inventory[i] + p;
```

So we obtain the only possible and non-trivial correct chopping, i.e., where the *inventory* update is detached into a separate transaction. The acyclic chopping graph from Figure 3 illustrates the correctness of such a chopping. Note that the requirement for a program-initiated transaction abort being part of first transaction piece is also satisfied here.

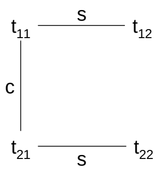


Figure 3: Chopping graph for the modified purchase procedure

Chapter 9 - Solutions to Exercises

Exercise 9.1 :

Discuss how the incremental key-range locking protocol on index keys has to be changed for a unique index. Discuss also how unique and non-unique indexes may be treated differently by the various optimizations sketched in Section 9.5.

When we consider unique indices, key-range locking has to be changed in the following way:

- No combination of insert and delete operations is commutative. For example, if transaction t_1 inserts a new key into the index, a subsequent insert of the same key will fail due to the uniqueness property of the index; if the two inserts were commuted, the latter would be successful, while the other would fail. If we considered return values, we could allow commuting two failed inserts, allowing a higher degree of concurrency.
- A successful delete operation always removes the key from the index, because there is only one associated tuple for this key. It is therefore necessary to lock the previous key, which could be omitted with a non-unique index. However, when the delete operation were implemented using the "deferred" delete technique, such that it removes only the rowid leaving the key in the index, no changes would be required.
- The optimization from Section 9.5.2 concerning a special insert lock mode is not applicable anymore, because two insert operations are no longer commutative as we already mentioned above.
- The optimization from Section 9.5.3 that suggests locking pairs of a key and a rowid (instead of locking only the key) does not allow any additional concurrency anymore, because there is at most one rowid for each key in the index.

Exercise 9.2 :

Discuss which lock(s) need to be acquired by an unsuccessful exact match query for a key that is smaller than the smallest key that exists in the index. How can you ensure that a subsequent insertion of this particular key "steps" on a lock conflict?

In order to properly handle such a situation we need to introduce a special virtual ("dummy") key representing $-\infty$ which is considered to be smaller than any physically existing key. Thus, an unsuccessful search for a key k that is smaller than the smallest key k_{min} results in locking the key $-\infty$, which in turn represents the interval $[-\infty,\ k_{min})$. When another transaction tries to execute $insert(k)$, it will have to acquire locks on k itself and on $-\infty$ according to the *incremental key range locking rules*, so that this type of conflicts is correctly detected as well.

The locking rules require no further changes.

Exercise 9.3 :

Incremental key-range locking based on read and write locks on index keys, as explained in Section 9.3, does not allow two insert operations in the same, previously unoccupied key interval to proceed concurrently. For example, in the index of Figure 9.5, an insert(27) would block a later insert(28) operation, although none of these keys are initially present and the two operations should be considered as commutative. Discuss whether the resulting blocking of one of the two operations can be safely avoided, and how this could be accomplished. Also discuss the analogous situation for delete operations, and the combination of insert and delete operations.

Section 9.5.2 of the book explicitly addresses these issues. We intuitively consider two inserts to be commutative operations, which leads us to the special *insert lock mode*. Thus, by locking the same previous key, 25 in this concrete example, the two transactions do not block each other. A lock in *insert mode*, however, commutes neither with read nor with write key locks. As we already mentioned in this chapter, insert and delete operations do not commute in general. But two deletes can be considered to be commutable. Thus, we can analogously introduce an additional self-compatible *delete lock mode*.

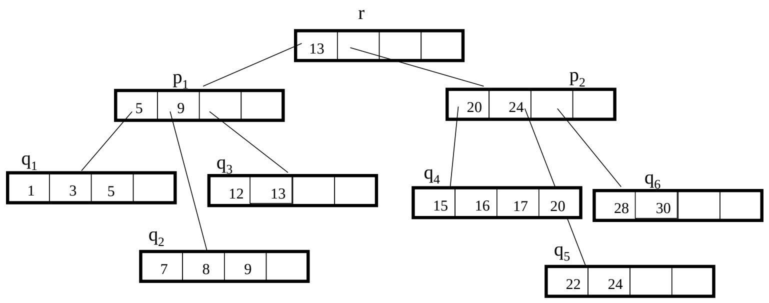
Exercise 9.4 :

Discuss the locking requirements for an unsuccessful insert operation on a unique index, i.e., when the key to be inserted already exists in the index.

As we already showed in Exercise 9.1 two unsuccessful inserts perfectly commute. This would be a motivation to use return value commutativity. We also notice that an unsuccessful insert commutes with all operations other than deletion of the same key.

Exercise 9.6 :

Consider the following B^+-tree index on the attribute AccountNumber of an Accounts table. Assume that all tree nodes have a space capacity to hold up to four entries. Which locks need to be requested for the execution of the following transaction, assuming the incremental key-range locking at the access layer, lock coupling at the page layer?



```
begin of transaction;
   select count(*) from Accounts where AccountNumber
          between 11 and 25;
   insert into Accounts (AccountNumber, ...) values (27, ...);
commit transaction;
```

search(11):
read lock r
read lock p_1
unlock r
read lock q_3
read lock q_2 // determine previous key
key-range lock 9 // previous key
unlock p_1
unlock q_3

unlock q_2
return 12, q_3

next(12, q_3*, 25):*
read lock q_3
key range lock 12
unlock q_3
return 13, q_3

next(13, q_3*, 25):*
read lock q_3
key range lock 13
read lock q_4
unlock q_3
unlock q_4
return 15, q_4

next(15, q_4*, 25):*
read lock q_4
key range lock 15
unlock q_4
return 16, q_4

next(16, q_4*, 25):*
read lock q_4
key range lock 16
unlock q_4
return 17, q_4

next(17, q_4*, 25):*
read lock q_4
key range lock 17
unlock q_4
return 20, q_4

next(20, q_4*, 25):*
read lock q_4
key range lock 20
read lock q_5
unlock q_4
unlock q_5
return 22, q_5

next(22, q_5*, 25):*
read lock q_5
key range lock 22
unlock q_5
return 24, q_4

next(24, q_5*, 25):*
read lock q_5
key range lock 24
read lock q_6
unlock q_5
unlock q_6
return no_more_keys, nil

insert(27):
write lock r
write lock p_2
unlock r

write lock q_6
read lock q_5 // get previous key
key range lock 24
key range lock 27
unlock q_5
unlock q_6
unlock p_2

commit:
unlock all key range locks

Exercise 9.7 :

Consider a situation where a node p in a B^+-tree is not split safe, but has a split safe descendant q, with both p and q on the traversal path of an insert operation. Discuss how the lock coupling technique should handle this situation to minimize lock contention.

Since the node q is split safe, this insert operation is not going to split node q. Therefore, we can safely release the locks on all of its predecessors.

Exercise 9.8 :

State explicit rules for lock requests and lock releases for the link technique (in a form similar to our presentation of locking rules for the lock coupling technique).

We assume that *Delete* is implemented similarly to *Search*, i.e., when the key is not found in a node where it is supposed to be, *Delete* follows the link to the sibling node until it finds the key or detects another key larger than that it searches for.

Link technique locking rules:

1. *Search* operations need to request a read lock on a node before the node can be accessed; *Insert* operations need to request a write lock. *Delete* operations need to acquire a write lock for a leaf node only, otherwise a read lock has to be acquired.
2. For *Insert* operations a lock on a node can be granted only if no conflicting lock is currently held and the requesting operation holds a lock on the node's parent. For *Delete* and *Search* operations the latter condition is dropped.
3. *Search* and *Delete* operations can release the lock on a node once it is processed.
4. *Insert* operations can release the lock on a node if (a) the node is split safe and (b) they have acquired a lock on a child of that node.
5. A *next*(*currentkey*, *currentpage*, *highkey*) operation needs to acquire a read lock on *currentpage*. If it needs to follow the link to the *currentpage*'s sibling it has to acquire a read lock for the sibling, but it does not have to hold the read lock on the *currentpage* in order to obtain this lock.

Chapter 10 - Solutions to Exercises

Exercise 10.1 :

Discuss to what extent lock escalation may incur deadlocks or livelocks.

In the process of lock escalation the fine-grained locks are converted into coarser-grained locks which increases the likelihood of additional conflicts between already admitted transactions. Consider the following scenario. Initially record level locking is used. A transaction t_1 updates records x_1 and y_1 in relations R_x and R_y, respectively. Another transaction t_2 reads records, say x_2 and y_2. This may result in the following interleaved execution of the two transactions on the server: $w_1(x_1)r_2(x_2)r_2(y_2)w_1(y_1)$. With all these records being pairwise distinct the transactions do not block each other. As the number of parallel transactions reaches some critical threshold, the data server might wish to start lock escalation and thereby convert all record level locks into table locks. Before releasing the record locks the transactions need to acquire the table locks. Thus, t_1 acquires an exclusive lock on R_x, and blocks t_2, whereas t_2 acquires a shared lock on R_y and prevents t_2 from obtaining an exclusive lock on R_y. In order to resolve this deadlock, one of the transactions has to be aborted and restarted.

As we already emphasized the number of conflicts between concurrent transactions increases significantly when the scheduler can grant coarse-grained locks only. So does the probability for a transaction to get involved in a deadlock. This can potentially lead to transaction starvation (*livelock*) as described in Section 4.3.3, such that a restarted transaction is chosen as deadlock victim over and over again.

Exercise 10.2 :

Discuss if and under which conditions a form of lock de-escalation could make sense. De-escalation would essentially be the opposite of lock escalation, converting one (or a few) coarse-grained lock(s) into several fine-grained locks while a transaction is executing.

When the number of deadlocks increases dramatically, it results in a major amount of wasted work, i.e., executing and rolling back the deadlock victim transactions. As a solution to this problem it could make sense to check, whether a deadlock persists after the lock de-escalation. If the deadlock can be resolved this way, the scheduler should carry out lock de-escalation only for transactions involved in the deadlock.

Exercise 10.3 :

Give examples for schedules that fall into the following five isolation-level classes (but not into the next larger, surrounding class): not even read uncommitted, read uncommitted, read committed, repeatable read, serializability.

(a) *not even read uncommitted*

In the schedule $s_1 = w_1(x)r_2(x)w_2(x)c_2c_1$ the transaction t_2 not only reads x from t_1, t_2 also overwrites x before t_1 finishes.

(b) *read uncommitted*

By putting c_1 between $r_2(x)$ and $w_2(x)$ in s_1 we obtain the schedule $s_2 = w_1(x)r_2(x)c_1w_2(x)c_2$ which can be produced under the read uncommitted isolation level.

(c) *read committed*

Consider the schedule $s_3 = r_1(x)r_2(x)w_2(x)c_2w_1(x)c_1$ which can be produced under the read committed isolation level. Since this schedule represents the lost-update anomaly, it does not satisfy the definition of the repeatable read isolation level.

(d) *repeatable read*

Let P be a predicate and x is not in P. The operation $w_2(x)$ changes x in a way that x satisfies P. Consider the schedule $s_4 = r_1(P)w_2(x)c_2r_1(P)c_1$. All locks are acquired properly according to S2PL but it does not prevent phantoms.

(e) *serializability*

Let P be a predicate with $x, y \notin P$. The operation $w_2(x)$ changes x in a way that x satisfies P. The operation $w_2(y)$ does not touch P. Consider the schedule $s_5 = r_1(P)w_2(y)r_1(P)c_1w_2(x)c_2$. Here the scheduler tests not only the compatibility of the locks on data items, but it also takes the predicate locks into account.

Exercise 10.4 :

Give an example for a snapshot-isolated history violating the consistency of the persistent data. Explain what kinds of inconsistencies can arise.

We revise the schedule $s = r_1(x_0)r_1(y_0)r_2(y_0)w_1(x_1)c_1w_2(y_2)c_2$ which we have already investigated in Exercise 5.9. This schedule satisfies the criterion of *snapshot isolation.*

As we have seen, the following data inconsistency may arise. Assume, that x and y are two counters (e.g., stock investments) with the constraint $x + y \leq 100$ (e.g., for risk limitation). The initial values are $x = 40$ and $y = 40$. Then, t_1 could set x to 60, and t_2 could set y to 60, thus arriving in a state that violates the constraint.

Exercise 10.5 :

Design a concurrency control protocol that can guarantee snapshot isolation (and can indeed generate all snapshot-isolated histories).

We can derive the following rules from the definition of the snapshot isolated schedules:

1. If there is no t_k with $x \in WS(t_k)$ and $begin(t_i) < c_k$, a step $w_i(x)$ is transformed into $w_i(x_i)$ (i.e., a write operation creates a new version of object x without overwriting the previous one), and x is added to $WS(t_i)$. Otherwise t_i has to be aborted. WS is a special data structure by whose means we keep track of write-accessed data items for each transaction.
2. A step $r_i(x)$ is transformed into $r_i(x_k)$ where t_k is the most recent committed transaction that updated x, and finished before t_i has started.
3. At the point where each transaction t_i with $begin(t_i) < c_k$ is finished (i.e., either committed or aborted), $WS(t_k)$ can be garbage-collected.

Exercise 10.6 :

Give examples for showing that MVSR and snapshot isolation are incomparable classes of histories, i.e., neither one of them is included in the other.

First consider the following schedule: $s_1 = r_1(x_0)w_1(x_1)r_2(y_0)c_1r_2(x_1)c_2$. This schedule is equivalent to the serial schedule t_1t_2. However, s_1 is not in SI, which you can verify by observing that t_2 reads x_1 instead of x_0.

Again reconsider the schedule $s_2 = r_1(x_0)r_1(y_0)r_2(y_0)w_1(x_1)c_1w_2(y_2)c_2$ which we have already investigated in Exercise 5.9. This schedule satisfies the criterion of *snapshot isolation*. However, we have shown, that s_2 is not in MVSR, since neither t_2 reads x from t_1 nor t_1 reads y from t_2, and thus, none of the two possible serialization is possible.

Chapter 11 - Solutions to Exercises

Exercise 11.1 :

Let $s_1 = w_1(x)w_1(y)r_2(u)w_2(x)r_2(y)w_2(y)a_2w_1(z)c_1$ and
$s_2 = w_1(x)w_1(y)r_2(u)w_2(x)r_2(y)w_2(y)w_1(z)a_1c_2$.

Determine exp(s_1) and exp(s_2) as well as the corresponding reductions.

$exp(s_1) = w_1(x)w_1(y)r_2(u)w_2(x)r_2(y)w_2(y)w_2^{-1}(y)w_2^{-1}(x)c_2w_1(z)c_1$
Then we obtain:

by UR: $w_1(x)w_1(y)r_2(u)w_2(x)r_2(y)w_2^{-1}(x)c_2w_1(z)c_1$
by CR: $w_1(x)w_1(y)r_2(u)w_2(x)w_2^{-1}(x)r_2(y)c_2w_1(z)c_1$
by UR: $w_1(x)w_1(y)r_2(u)r_2(y)c_2w_1(z)c_1$
by NR: $w_1(x)w_1(y)c_2w_1(z)c_1$

As you see s_1 is a reducible schedule.
$exp(s_2) = w_1(x)w_1(y)r_2(u)w_2(x)r_2(y)w_2(y)w_1(z)w_1^{-1}(z)w_1^{-1}(y)w_1^{-1}(x)c_1c_2$
Then we obtain:

by UR: $w_1(x)w_1(y)r_2(u)w_2(x)r_2(y)w_2(y)w_1^{-1}(y)w_1^{-1}(x)c_1c_2$

There is no way to further reduce s_2's expansion because we cannot move $w_1(x)$ and $w_1^{-1}(x)$ together, since these two operations are totally ordered by $w_1(x) <_{exp(s_2)} w_2(x) <_{exp(s_2)} w_1^{-1}(x)$. A similar argument holds for the writes on y because of: $w_1(y) <_{exp(s_2)} r_2(y) <_{exp(s_2)} w_2(y) <_{exp(s_2)} w_1^{-1}(y)$.

Exercise 11.2 :

Which of the properties RC, ST, RG, PRED, and LRC are satisfied by the following schedules:

$s_1 = r_1(a)r_2(a)w_1(a)c_1c_2$
$s_2 = r_1(a)w_1(a)r_2(b)w_2(b)w_2(a)c_2c_1$
$s_3 = r_1(a)incr_2(a)incr_2(b)incr_3(b)c_3a_2c_1$

Some of the above properties cannot be applied to a schedule with operations other than read and write, such as schedule s_3 above. Try to define generalizations of these properties to flat object schedules with arbitrary operations.

Consider s_1 first. $s_1 \in CSR$ because there is only one conflict pair in this schedule. $s_1 \in RC$, because both transactions do not read from each other. s_1 is not only recoverable but also log recoverable because there is no write/write conflict so that the second criterion for membership in LRC is automatically met. With $s_1 \in CSR$ and $s_1 \in LRC$ we obtain by Theorem 11.5 $s_1 \in PRED$. $s_1 \notin RG$ because we have a read/write conflict between t_2 and t_1 when they are both uncommitted, but the schedule is strict, i.e., $s_1 \in ST$ because neither write/read nor write/write conflicts appear in the schedule.

Let us turn to schedule s_2. This schedule is recoverable (i.e., $s_2 \in RC$), because neither transaction reads from the other. However, $s_2 \notin LRC$ because $c_2 <_{s_2} c_1$ whereas $w_1(a) <_{s_2} w_2(a)$. $s_2 \notin LRC$ implies that $s_2 \notin PRED$ which follows from Theorem 11.5. $s_2 \notin ST$ because of the write/write conflict mentioned above between the two uncommitted transactions and thus, the schedule is not in RG either.

In order to be able to apply the syntactic recoverability criteria that have been introduced for the page model to flat objects, we first have to classify L_1 operations into read-only, mixed read/update and update-only operation classes. The updates will play the same role as the writes in the page model and the read-only operations can be handled the same way as the page model reads. The mixed operations have to be considered as both, reads and writes, when we will adopt the page model criteria. In this particular example the *incr* operation is clearly a mixed read/update operation, since you have to read the current counter value first prior to incrementing and writing it back to the stable storage. Next, we have to define the inverse operation for each update(-only) operation. Not surprisingly, *decr* is the inverse operation for *incr* and vice versa; *decr* is also a mixed read/update operation. With the considerations above the adoption of the definitions of the expanded schedule, RED and PRED to the flat object schedules is quite straightforward.

Finally, we can consider schedule s_3. This schedule is not in RC because it exposes a typical dirty read situation. t_3 reads from t_2 by $incr_3(b)$, and t_2 is aborted after t_3 is already committed. Consequently s_3 is not in LRC and therefore not in PRED either. $s_3 \notin ST$ because the conflict between $incr_2(a)$ and $incr_3(a)$ (*incr* implies a write operation) occurs when both corresponding transactions are active, and thus, this schedule is not in RG either.

Exercise 11.3 :

For each of the following histories determine to which of the classes RC, ACA, or ST it belongs:

$$s_1 = w_1(x)r_2(y)r_1(x)c_1r_2(x)w_2(y)c_2$$
$$s_2 = w_1(x)r_2(y)r_1(x)r_2(x)c_1w_2(y)c_2$$
$$s_3 = w_1(x)r_2(y)r_2(x)r_1(x)c_2w_1(y)c_1$$

The schedule s_1 is in RC because for t_2 reading from t_1 we have also $c_1 <_{s_1} c_2$. s_1 is in ACA because $w_1(x) <_{s_1} c_1 <_{s_1} r_2(x)$. This is the only conflict occurring in the schedule s_1, therefore the criteria for $s_1 \in ST$ are satisfied as well.

The schedule s_2 is a modification of s_1 in a way that now t_2 reads an uncommitted data item x from t_1. Thus, s_2 is still recoverable (i.e., $s_2 \in RC$), but it is neither in ACA nor in ST.

The schedule s_3 is not in RC because in contrast to s_1 we obtain for t_2 reading from t_1 $c_2 <_{s_3} c_1$. Therefore s_3 is automatically neither in ACA nor in ST (see Theorem 11.2).

Exercise 11.6 :

A history s is *prefix expanded conflict serializable* if each of its prefixes is expanded conflict serializable. Let PXCSR denote the class of all prefix expanded conflict serializable histories. Show the following:
(a) PXCSR $\subset$ XCSR
(b) PXCSR $\subset$ PRED
(c) RG $\subset$ PXCSR
(d) PXCSR = ST $\cap$ XCSR
(e) Based on results (a)–(d), complete Figure 11.3.

(a) $PXCSR \subseteq XCSR$ is trivially true since each schedule is a prefix of itself. We have to show that the inclusion is strict. Consider the following schedule: $s = w_1(x)r_2(y)w_2(x)c_2c_1$. The expansion of this schedule is the schedule itself and it is in XCSR because it contains only one conflict pair. However, if we consider the longest proper prefix of the schedule (i.e., $s' = w_1(x)r_2(y)w_2(x)c_2$), t_1 appears in the set of active transactions, and thus, must be aborted. So

$exp(s') = w_1(x)r_2(y)w_2(x)c_2w_1^{-1}(x)c_1$ is not in XCSR because a cycle arises $w_1(x) <_{exp(s')} w_2(x) <_{exp(s')} w_1^{-1}(x)$. So we showed $PXCSR \subset XCSR$.

(b) $XCSR \subset RED$ according to Theorem 11.1. This implies that a schedule which is in PXCSR is automatically in PRED as well (i.e., $PXCSR \subseteq PRED$) . Now consider the schedule $s = r_1(x)w_2(x)w_2(y)c_2r_1(y)$. This schedule is clearly not in XCSR and thus not in PXCSR either. However, s is a prefix reducible schedule as you may convince yourself by applying the Null Rule, which simply removes t_1 from s (and all of its prefixes). Therefore, we obtain $PXCSR \subset PRED$.

(c) Consider an arbitrary schedule $s \in RG$. By Theorem 11.3 we already showed that $CP(s) \in COCSR \subset CSR$. Convince yourself that if we replace all *abort* by *commit* operations the schedule remains rigorous. Now we are going to show that no edges are added to the conflict graph in the process of the schedule expansion, where all inverse operations are inserted immediately before the abort replaced by a commit operation. For each conflict pair $(p_i,\ q_j)$ with $p_i <_s q_j$ we obtain: $p_i <_{exp(s)} p_i^{-1} <_{exp(s)} c_i <_{exp(s)} q_j <_{exp(s)} q_j^{-1}$, but these conflicts are already reflected by the edge $t_i \rightarrow t_j$ in the original conflict graph. So we showed $RG \subseteq XCSR$. It can be easily verified that each prefix of a rigorous schedule is also rigorous and thus we obtain $RG \subseteq PXCSR$. With the schedule $r_1(x)w_2(x)a_2c_1$ which is in PXCSR but not in RG you may verify that the inclusion is strict, i.e., $RG \subset PXCSR$.

(d)
" $\subseteq$: " Consider an arbitrary schedule $s \in PXCSR$. Since $PXCSR \subset XCSR$, we only have to show that $s \in ST$. Assume, this is not the case. Then there are some $w_j(x)$ and $p_i(x)$ such that $w_j(x) <_s p_i(x) <_s f_j$ with $p_i \in \{r_i, w_i\}$ and $f_j \in \{a_j,\ c_j\}$. Now consider the prefix of s ending with $p_i(x)$. Let us denote it s'. For its expansion we obtain $w_j(x) <_{exp(s')} p_i(x) <_{exp(s')} w_j^{-1}(x)$, which clearly results in a cycle. $s' \notin XCSR$ which incurs $s \notin PXCSR$. This is a contradiction. Thus, we obtain $PXCSR \subseteq ST \cap XCSR$.

" $\supseteq$: " Consider a schedule $s \in ST \cap XCSR$. Assume, $s \notin PXCSR$. Consider the longest prefix s' which is not in XCSR. Some transaction t_j, which is committed in the original schedule, becomes now active and has to be aborted and this causes a cycle. So we have $w_j(x) <_{exp(s')} p_k(x) <_{exp(s')} w_j^{-1}(x) <_{exp(s')} c_j$ with $p_k \in \{r_k,\ w_k\}$, $k \neq j$. But this implies that in the original schedule s we must have had $w_j(x) <_s p_k(x) <_s c_j$ which is a contradiction to the fact $s \in ST$.

So we showed $PXCSR = ST \cap XCSR$

(e)

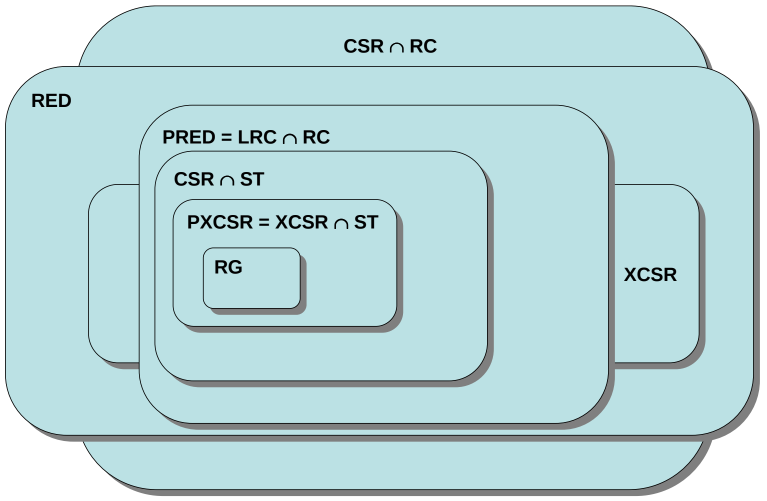


Figure 1: Completed Figure 11.3

Exercise 11.7 :

Prove Theorem 11.6 stating that *Gen*(SS2PL) = RG.

" $\subseteq$: " With SS2PL write and read locks are held until the end (i.e, commit or abort) of the transaction. Thus, if there is a conflict between two operations, say p_i and q_j, then we obtain the following relationship:

$$pl_i <_s p_i <_s pu_i <_s \{c_i, a_i\} <_s ql_j <_s q_j$$

Thus, the rigorousness criteria are satisfied.

" $\supseteq$: " Let us examine an arbitrary rigorous schedule s (i.e., $s \in RG$). For each conflict pair (p_i, q_j) we obtain $p_i <_s \{c_i, a_i\} <_s q_j$. Without changing the schedule semantics we can add a lock acquisition step immediately before the corresponding data operation and insert a lock release step after all data operations but before *commit* or *abort* respectively: $pl_i <_s p_i <_s pu_i <_s \{c_i, a_i\} <_s ql_j <_s q_j$. Obviously such a schedule can be produced by an SS2PL scheduler.

So we showed *Gen*(SS2PL) = RG.

Exercise 11.8 :

Consider a database consisting of positive integers with the following operations defined on them:

- $incr(x)$: increments x if $x > 0$ and returns 1; otherwise does nothing and returns 0.
- $incr^{-1}(x, y)$: decrements x if y is the return value of the corresponding forward operation and $y \neq 0$; otherwise does nothing, always return 0.
- $reset(x)$: reset x to 1 and returns the old value of x.
- $reset^{-1}(x)$ set x to value y, where y is the return value of the corresponding forward operation; always returns 0.
- $retrieve(x)$: returns the current value of x.
- $retrieve^{-1}(x)$: is a null operation and returns an empty sequence.
- $decr(x)$: decrements x and return 0.
- $decr^{-1}(x)$: increments x and returns 0.

Determine the commutativity relation for these operations, and find out whether it is perfect or normal. If one of the latter properties does not hold, try to restrict the relation in such a way that this respective property is achieved.

	$incr(x)$	$reset(x)$	$retrieve(x)$	$decr(x)$	$incr^{-1}(x,y)$	$reset^{-1}(x)$	$decr^{-1}(x)$
$incr(x)$	+	−	−	−	−	−	−
$reset(x)$	−	−	−	−	−	−	−
$retrieve(x)$	−	−	+	−	−	−	−
$decr(x)$	−	−	−	+	+	−	+
$incr^{-1}(x,y)$	−	−	−	+	+	−	+
$reset^{-1}(x)$	−	−	−	−	−	+	−
$decr^{-1}(x)$	−	−	−	+	+	−	+

Figure 2: Commutativity relation

The table in Figure 2 represents the commutativity relation for the operations. $retrieve^{-1}$ is omitted in the table, since it is a *null* operation which trivially commutes with each other operation. The commutativity relation is not perfect because, for instance, two *incr* operations do commute, whereas $incr(x)$ and $incr^{-1}(x)$ do

	$incr(x)$	$reset(x)$	$retrieve(x)$	$decr(x)$	$incr^{-1}(x,y)$	$reset^{-1}(x)$	$decr^{-1}(x)$
$incr(x)$	$-$	$-$	$-$	$-$	$-$	$-$	$-$
$reset(x)$	$-$	$-$	$-$	$-$	$-$	$-$	$-$
$retrieve(x)$	$-$	$-$	$+$	$-$	$-$	$-$	$-$
$decr(x)$	$-$	$-$	$-$	$+$	$-$	$-$	$+$
$incr^{-1}(x,y)$	$-$	$-$	$-$	$-$	$-$	$-$	$-$
$reset^{-1}(x)$	$-$	$-$	$-$	$-$	$-$	$-$	$-$
$decr^{-1}(x)$	$-$	$-$	$-$	$+$	$-$	$-$	$+$

Figure 3: Perfect closure of the lock compatibility table

not. The commutativity relation is not normal either, because of the fact that, for instance, *decr* and *incr* do not commute whereas *decr* and $incr^{-1}$ are commutable.

The table in Figure 3 depicts the perfect closure of the commutativity relation, which is $perfect$ and *normal* as well.

Chapter 12 - Solutions to Exercises

Exercise 12.1 :

Assume that both the database cache and the log buffer are, to a large extent, crash resilient (i.e., survive a system failure and are thus accessible with their pre-crash contents during restart) by using battery backed, non-volatile RAM (also known as "RAM disk" or "flash memory"). Discuss which types of failures still need recovery measures? How safe is such a safe-RAM approach?

If the data structures kept in RAM can survive the crash, we can consider these as equally safe as those on disk. Thus, there is no need to redo updates executed on behalf of committed transactions, but we will still have to undo uncommitted updates, in order to satisfy the correctness criterion given in Definition 12.6.

However, if it is Heisenbugs in the software that cause the crash then we cannot trust the contents of the memory-resident data structures (for the software could have misaddressed and thus corrupted data structures). In this case, memory should be re-initiated anyway. A possible approach to guard critical data structures against software errors would be to enable critical memory protection on these structures and minimize the, then implicitly trusted, code for which the protection is temporarily disabled.

Exercise 12.2 :

Assume again that both the database and the log buffer reside in safe RAM. In addition, assume that virtual-memory page protection bits are used to carefully control the software access to these data structures during normal operation. Discuss again which types of failures still need explicit recovery measures. For the class of system failures (i.e., soft crashes), which of the following statements are true, which ones are false?

(a) Undo recovery is no longer needed at all.

(b) Redo recovery is no longer needed at all.

(c) When a page is flushed from the database cache during normal operation, the log buffer must be forced beforehand.

(d) When a page is flushed from the database cache during restart, the log buffer must be forced beforehand.

(e) When a transaction commits (during normal operation), the log buffer must be forced beforehand.

(a) is false (see Exercise 12.1).

(b) is true (see Exercise 12.1).

(c) No, sufficient undo information is provided in the log buffer and it will be available after the crash.

(d) No, for the same reason as (c).

(e) No, since we only have to guarantee that the cached database will reflect committed updates. With safe RAM the cached database persists over crashes.

Chapter 13 - Solutions to Exercises

Exercise 13.1 :

Consider the the action history given in Figure 13.20, including checkpoints and flush actions. Assume that there is a system crash right after the last action. Determine the necessary logging actions during normal operation and the recovery actions during restart, by completing the table. First consider the case where heavy-weight checkpoints are used and flush actions are not logged, then consider lightweight checkpoints, and finally discuss the additional effect of keeping log entries for flush actions. In all cases, assume that the redo-history paradigm is employed.

Now consider the extended scenario given in Figure 13.21. In contrast to the previous scenario, this action history contains two transaction rollbacks during normal operation one of which is completed before the crash whereas the second one is interrupted by the crash. Determine again the necessary logging and restart actions for this scenario by completing Figure 13.21.

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action]	Log entries added to stable log [LogSeqNo's]
1:begin(t_1)			1: begin(t_1)	
2: write(p, t_1)	p:2		2: write(p, t_1)	
3: write(q, t_1)	q:3		3: write(q, t_1)	
4: commit(t_1)			4: commit(t_1)	1,2,3,4
5: flush(p)		p:2		
6: begin(t_2)			6: begin(t_2)	
7: write(p, t_2)	p:7		7: write(p, t_2)	
8: write(r, t_2)	r:8		8: write(r, t_2)	
9: checkpoint		p:7,q:3,r:8	9: CP ActiveTrans={t_2}	6,7,8,9
10: commit(t_2)			10: commit(t_2)	10
11: begin(t_3)			11: begin(t_3)	
12: flush(p)				11
13: write(p, t_3)	p:13		13: write(p, t_3)	
14: write(q, t_3)	q:14		14: write(q, t_3)	
15: flush(q)		q:14		13,14
16: write(r, t_3)	r:16		16: write(r, t_3)	
Crash				
Analysis pass: losers={t_3}				
redo(13)	p:13			
consider-redo(14)	q:14			
16: compensate(14)	q:16		16: compensate(14)	
17: compensate(13)	p:17		17: compensate(13)	
18: rollback(t_3)			18: rollback(t_3)	16, 17, 18

Figure 1: *Action history from Figure 13.20 with heavyweight checkpoints*

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action]	Log entries added to stable log [LogSeqNo's]
1:begin(t_1)			1: begin(t_1)	
2: write(p, t_1)	p:2		2: write(p, t_1)	
3: write(q, t_1)	q:3		3: write(q, t_1)	
4: commit(t_1)			4: commit(t_1)	1, 2, 3, 4
5: flush(p)		p:2		
6: begin(t_2)			6: begin(t_2)	
7: write(p, t_2)	p:7		7: write(p, t_2)	
8: write(r, t_2)	r:8		8: write(r, t_2)	
9: checkpoint			9: CP DirtyPages={p, q, r} RedoLSNs={p:7, q:3, r:8} ActiveTrans={t_2}	6, 7, 8, 9
10: commit(t_2)			10: commit(t_2)	10
11: begin(t_3)			11: begin(t_3)	
12: flush(p)		p:7		11
13: write(p, t_3)	p:13		13: write(p, t_3)	
14: write(q, t_3)	q:14		14: write(q, t_3)	
15: flush(q)		q:14		13, 14
16: write(r, t_3)	r:16		16: write(r, t_3)	
Crash				
Analysis pass: loser={t_3} DirtyPages={p, q, r} RedoLSNs={p:7, q:3, r:8}				
consider-redo(3)	q:14			
consider-redo(7)	p:7			
redo(8)	r:8			
redo(13)	p:13			
skip-redo(14)				
16: compensate(14)	q:16		16: compensate(14)	
17: compensate(13)	p:17		17: compensate(13)	
18: rollback(t_3)			18: rollback(t_3)	16, 17, 18

Figure 2: *Action history from Figure 13.20 with lightweight checkpoints*

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action]	Log entries added to stable log [LogSeqNo's]
1:begin(t_1)			1: begin(t_1)	
2: write(p, t_1)	p:2		2: write(p, t_1)	
3: write(q, t_1)	q:3		3: write(q, t_1)	
4: commit(t_1)			4: commit(t_1)	1, 2, 3, 4
5:flush(p)		p:2	5:flush(p)	5
6: begin(t_2)			6: begin(t_2)	
7: write(p, t_2)	p:7		7: write(p, t_2)	
8: write(r, t_2)	r:8		8: write(r, t_2)	
9: checkpoint			9: CP DirtyPages={p, q, r} RedoLSNs={p:7, q:3, r:8} ActiveTrans={t_2}	6, 7, 8, 9
10: commit(t_2)			10: commit(t_2)	10
11: begin(t_3)			11: begin(t_3)	
12: flush(p)		p:7	12:flush(p)	11, 12
13: write(p, t_3)	p:13		13: write(p, t_3)	
14: write(q, t_3)	q:14		14: write(q, t_3)	
15: flush(q)		q:14	15: flush(q)	13, 14, 15
16: write(r, t_3)	r:16		16: write(r, t_3)	
Crash				
Analysis pass: losers={t_3}, DirtyPages={p,r} RedoLSNs={p:13, r:8}				
redo(8)	r:8			
redo(13)	p:13			
skip-redo(14)				
16: compensate(14)	q:16		16: compensate(14)	
17: compensate(13)	p:17		17: compensate(13)	
18: rollback(t_3)			18: rollback(t_3)	16, 17, 18

Figure 3: *Action history from Figure 13.20 with flush logentries and lightweight checkpoints*

Sequence number: action	Change of cached database [PageNo: SeqNo]	Change of stable database [PageNo: SeqNo]	Log entry added to log buffer [LogSeqNo: action]	Log entries added to stable log [LogSeqNo's]
1:begin(t_1)			1:begin(t_1)	
2: write(p, t_1)	p:2		2: write(p, t_1)	
3: write(q, t_1)				
4: commit(t_1)			4: commit(t_1)	1,2,3,4
5: flush(p)		p:5	5: flush(p)	
6: begin(t_2)			6: begin(t_2)	
7: write(p, t_2)	p:7		7: write(p, t_2)	
8: write(r, t_2)	r:8		8: write(r, t_2)	
9: checkpoint			9: CP DirtyPages={p, q, r} RedoLSNs={p:7, q:3, r:8} ActiveTrans={t_2}	5, 6, 7, 8, 9
10: abort(t_2)				
11: compensate(8)	r:11		11: commpensate(8: r, t_2) NextUndoSeqNo:7	
12: compensate(7)	p:12		12: compensate(7: p, t_2) NextUndoSeqNo:6	
13: rollback(t_2)			13: rollback(t_2)	10, 11, 12, 13
14: begin(t_3)			14: begin(t_3)	
15: flush(p)		p:12	15: flush(p)	
16: write(p, t_3)	p:16		16: write(p, t_3)	
17: write(q, t_3)	q:17		17: write(q, t_3)	
18: flush(q)		q:17	18: flush(q)	14, 15, 16, 17, 18
19: write(r, t_3)	r:19		19: write(r, t_3)	
20: abort(t_3)				
21: compensate(19)	r:21		21: compensate(19: r, t_3) NextUndoSeqNo:17	
22: begin(t_4)			22: begin(t_4)	
23: compensate(17)	q:23		compensate(17: q, t_3) NextUndoSeqNo:16	
24: write(s, t_4)	s:24		24: write(s, t_4)	
25: flush(q)		q:23	25: flush(q)	19, 20, 21, 22, 23, 24, 25
26: commit(t_4)			26: commit(t_4)	26
Crash				
Analysis pass: losers={t_3} DirtyPages={p, r, s} RedoLSNs={16, 8, 24}				
redo(8)	r:8			
redo(11)	r:11			
skip-redo(12)				
redo(16)	p:16			
skip-redo(17)				
redo(19)	r:19			
redo(21)	r:21			
skip-redo(23)				
redo(24)	s:24			
27: compensate(16)	p:27		27: compensate(16: p, t_3) NextUndoSeqNo:14	
28: rollback(t_3)			28: rollback(t_3)	27, 28

Figure 4: *Action history from Figure 13.21*

Exercise 13.3 :

Consider the following specialization of the redo-winners algorithm. Assume that all updates are logged as full-writes, so that log entries have the form of page before images or after images. Further assume that the database cache is large enough to have the "no-steal" property, which guarantees that a page that has been modified by an active (i.e., incomplete) transaction is never flushed. In addition, assume that the log buffer is also large enough to contain all before images of all active transactions. Design a special recovery algorithm under this premise that (i) should be simpler than the general purpose redo-winners algorithm and (ii) aims to shorten the restart duration as much as possible. In particular address the following issues:

Are checkpoints still needed at all (under the no-steal premise)? How can you handle transaction aborts? Is there a way of gracefully degrading the special algorithm if the no-steal property cannot be guaranteed but the database cache is almost always large enough to avoid flushing pages whose last modification belongs to an active transaction?

Hint: The DB Cache algorithm of Bayer and Elhardt (1984) is such an algorithm (see bibliographical remarks).

With "no-steal" guarantee provided, the undo pass is no longer needed, because active transactions do not leave any traces in the stable databases. This simplifies and speeds up the restart procedure after a system failure. As a consequence, undo information has not to be logged during normal operation, which results in significant disk space saving when physical logging is used. The only limitation for the log-truncation procedure is the oldest redo LSN. And if the data server deploys a write-behind daemon, (which flushes pages with committed updates in the background starting with the lowest page sequence number whenever it discovers low I/O activity on the disk) all committed updates will eventually be permanently stored on the disk and log-truncation will always be possible. So we can essentially abandon periodic checkpointing, which accelerates transaction processing during normal operation.

Now consider the case, in which the data server has to deal with several long running transactions so that the overall number of concurrent transactions increases in a way that from time to time the "no-steal" property can no longer be guaranteed. Bayer and Elhardt propose to handle some transactions in such a situation in a similar way as in the original redo-winners algorithm. When a cache overflow arises cache manager selects active transactions with the largest amount of allocated cache memory. For each of these transactions a special before image log is created on the disk. Note that before images can be obtained from the stable DB, since the corresponding changed copies of the pages have not been flushed yet. After the logs are created, the pages can be replaced by the cache manager. Upon a commit of a "long" transaction all pages by this transaction are flushed and the before image log is deleted (this is the commit point). So when the data server is restarted and the redo pass is completed, the server looks for any before image logs and if present the before images are written back to the disk. Note that this method as described here works only with exclusive write locks on pages, where at any point in time only one transaction may access a page in write-mode.

Exercise 13.4 :

Reconsider the redo pass of the redo-history algorithm. Show that writes for a page that originate from loser transactions and are not followed by any winner writes on the same page do not need to be redone during the redo pass. Design a variant of the redo-history algorithm, both the redo and the undo pass, that avoids redoing writes in the above category, thus reducing the overall work of the redo pass. Note that this reduction is usually not a major gain as it affects only a small fraction of log entries; this variant is interesting in that it consolidates and deepens the understanding of the relationships between the redo and the undo pass.
Hint: The ARIES/RRH algorithm by Mohan and Pirahesh (1991) is such a variant of the redo-history algorithm (see bibliographical remarks).

The effect of a complete recovery process after a failure is that each page in the cached database along with all other pages on the stable database reflect the most recent winner writes. This means that all the loser updates following the most recent winner one will be erased anyway and thus do not have to be redone. This is feasible because we apply page state testing and will not erroneously undo an update, which has not been redone. As we handle completely aborted transactions as winners (no matter, whether in the context of transaction recovery or complete crash recovery) we can state that a winner update never follows a loser update with regard to each particular page (since we still consider page-level locking). This property immediately follows from the LRC criterion, which is fundamental for Part III of this book.

With the previous considerations we can simply ignore the log entries of forward operations belonging to loser transactions during the redo pass. In contrast to the redo-winners paradigm we will still need to redo loser's inverse steps (if needed according to the result of page state testing) because otherwise we might erroneously "forget" to undo a loser update which was already undone during prior transaction/crash-recovery but was not written back to the stable database (note that this issue arises only in the context of NextUndoSeqNo chains avoiding multiple-times inverse actions). Consider the following sample history:

$$1 : w_1(x) \;\; 2 : c_1 \;\; 3 : w_2(x) \;\; 4 : w_2(y) \;\; flush(y) \;\; 5 : w_2^{-1}(y) \;\; logforce \;\; crash$$

Assume that log buffer had to be forced to the disk and then a system crash happened. In this scenario changes on y made by t_2 are already part of the stable database. Since in the undo pass we will immediately skip to `StableLog[5].NextUndoSeqNo` (which is 3) thus leaving $w_2(y)$ undone, the loser's CLEs need special handling during the redo pass. So we repeat a loser's CLE only if the page sequence number is greater than or equal to the LSN of the forward operation (which we now additionally include in each CLE, denoted `UndoneLSN`) and less than the CLE's LSN. The redo pass algorithm with this enhancement is given in Figure 5. In the sample scenario, we would thus redo the operations with LSNs 1 and 5, but would skip 3 (a loser update not yet undone) and 4 (an update already flushed to disk).

The undo pass of the algorithm in its final presentation in this chapter does not require any changes.

```
redo pass ( ):
  SystemRedoLSN := min DirtyPages[p].RedoSeqNo;
  max := LogSeqNo of most recent log entry in StableLog;
  for i := SystemRedoLSN to max do
    if (StableLog[i].TransId not in losers) and
       (StableLog[i].ActionType = write or full-write or compensate)
    then
      pageno = StableLog[i].PageNo;
      if pageno in DirtyPages and
         DirtyPages[pageno].RedoSeqNo < i
      then
        fetch (pageno);
        if DatabaseCache[pageno].PageSeqNo < i
          then
            read and write (pageno) according to StableLog[i].RedoInfo;
            DatabaseCache[pageno].PageSeqNo := i;
          end /*if*/;
      end /*if*/;
    end /*if*/;
    if (StableLog[i].TransId is in losers) and
       (StableLog[i].ActionType = compensate)
    then
      pageno = StableLog[i].PageNo;
      if pageno in DirtyPages and
         DirtyPages[pageno].RedoSeqNo < i
      then
        fetch (pageno);
        if StableLog[i].UndoneLSN ≤ DatabaseCache[pageno].PageSeqNo and
           DatabaseCache[pageno].PageSeqNo < i
        then
           read and write (pageno) according to StableLog[i].RedoInfo;
           DatabaseCache[pageno].PageSeqNo := i;
        end /*if*/;
      end /*if*/;
    end /*if*/;
  end /*for*/;
```

Figure 5: *Redo-history with avoidance of forward operation redo for losers*

Chapter 14 - Solutions to Exercises

Exercise 14.1 :

Consider the algorithm for object model recovery. Assume that the undo information and the redo information for an action are recorded in separate log entries, rather than always combining them into one log entry. What is the impact on the algorithm? In particular, what are the implications for the order in which log entries must be created and the forcing of the log buffer? Sketch the modified algorithm.

The goal of each database recovery is to preserve committed updates and to erase all loser traces from the stable database. Since each transaction which is currently active is a potential loser we have to ensure that we can undo its updates after the redo pass is completed. Now it is clear that the undo log entry has to be forced before the redo log entry, because otherwise we may lose the undo information of the very last forward operation detected in the stable log. Note that these considerations concern only L_0 operations, since we create a separate log entry with undo information for object level operations anyway (i.e., subcommit).

The algorithm as it is presented in the book does not require significant changes: in the redo pass we simply ignore the log entries with undo information and vice versa. Backward chaining is needed only for the log entries with undo information.

Exercise 14.2 :

Consider the DB Cache method (see Exercise 13.3), applied to subtransactions, for implementing the L_0 recovery of the simple two-level recovery algorithm. When adopted in a straightforward way, this method would force after-images to the stable log, coined the "safe", upon each subcommit. Discuss under which conditions these forced log I/Os can be avoided. As a hint, assume that a set of after-images can be written to the safe atomically and analyze what happens when the forcing of a subtransaction's after-images is deferred until the transaction's commit. In particular, analyze how to deal with subtransactions, belonging to the same or different transactions, whose write sets have pages in common. How do such "dependencies" between subtransactions affect the deferral of forced log I/Os?

These optimizations have been examined by Weikum and Hasse (1993) (see Bibliographic Notes and References in the book).

The fact that subtransactions need not be made persistent at the subcommit point is the main motivation for deferred L_0 log forcing. The ideal solution would be to postpone forcing after-images until the commit point of the transaction, which does not work as you will see below. An issue which needs careful handling is that now a particular page can be simultaneously accessed by multiple transactions (see Chapters 6 and 7) and thus even with physical logging by means of after images we have to take into account that an after-image reflects not only the updates made on behalf of the transaction to be committed but it also reflects a number of potentially uncommitted updates made on behalf of high-level commutative operations belonging to some other active transactions. Consider a two-level history shown in Figure 1. When t_2 commits and we subsequently flush the after images produced by t_2, the changes made by t_1 on the page p become also part of the stable database (or of the safe). Imagine that the data server fails immediately after that. In this case we cannot redo $Withdraw_1(a)$ completely since we do not have the after image for the page q and therefore the subsequent undo by means of L_1 undo information is not possible either. Note that undo of partially executed subtransactions as in the

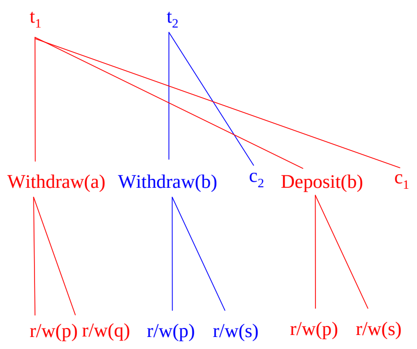


Figure 1: Sample two-level history

original simple recovery algorithm is not feasible because with DB Cache we don't store before images in the stable log. Even if $Withdraw_2(a)$ only read the page p the subsequent write on the page s would most likely semantically depend on the content of p. Therefore losing t_1's update on p together with making t_2's update on s persistent is not acceptable either. These considerations motivate a "forces" relationship between two **completed** subtransactions $t_{ij} \rightarrow t_{kl}$ (pronounced: t_{ij} forces t_{kl}). $t_{ij} \rightarrow t_{kl}$ holds if t_{ij} modifies or reads a page that has previously been modified by t_{kl} but had not been made persistent yet (i.e., forced to the safe, the stable log)). This is a transitive, asymmetric relationship. The *Persistence Sphere* of t_{ij} denoted $PS(t_{ij})$ contains all pages modified by t_{ij} but not yet made persistent and it is transitively closed with respect to the $\rightarrow$ relationship.

Now the solution is obvious: upon commit of a transaction t_i we have to force not only the after images from the transaction's write-set but also all other pages from $PS(t_i)$ which is the union of the persistence spheres of its subtransactions. Additionally, if some dirty page p is chosen as a cache replacement victim, we need to force the persistence sphere of the last completed subtransaction which modified p. Note that writing a persistence sphere to the safe can be easily made atomic, and indeed needs to be made atomic, as this write is a sequential disk I/O.

In the example from Figure 1 we now obtain a recoverable execution because, as $Withdraw_2(b) \rightarrow Withdraw_1(a)$ holds, the after image set we are going to force consists not only of p and s but also of q and thus, the atomicity of $Withdraw_1(a)$ is preserved as well. So it can be undone in the undo pass by using L_1 undo information, which must always be made persistent immediately before forcing L_0 log or writing back a page.

Exercise 14.3 :

Consider the two-level action history given in Figure 14.9, with operations on records ($store$, $modify$) and index keys ($insert$, $delete$). For the latter operations, the first parameter denotes a key and the second parameter the RID (i.e., address) of a stored record. Assume that there is a system crash right after the last action. Determine the necessary logging actions during normal operation and the recovery actions during restart, by completing the table. Use the enhanced 2-level recovery algorithm with a single log from Section 14.4.

Sequence number: action	Cached changes [PageNo:SeqNo]	Stable changes [PageNo:SeqNo]	Log entry added [LogSeqNo: action] [NextUndoSeqNo]
1: begin(t_1)			1: begin(t_1), next=nil
2: modify(x, t_1)			
3: subbegin(t_{11})			
4: write(p, t_{11})	p:4		4: write(p, t_{11}), next=nil
5: begin(t_2)			5: begin(t_2), next=nil
6: store(y, t_2)			
7: subbegin(t_{21})			
8: write (q, t_{21})	q:8		8: write(q, t_{21}), next=nil
9: write(r, t_{21})	r:9		9: write(r, t_{21}), next=8
10: subcommit(t_{21})			10: store^{-1}(y, t_2), next=nil
11: write(r, t_{11})	r:11		11: write(r, t_{11}), next=4
12: subcommit(t_{11})			12: modify^{-1}(x, t_1), next=nil
13: delete(a,@x, t_1)			
14: subbegin(t_{12})			
15: write(l, t_{12})	l:15		15: write(l, t_{12}), next=12
18: subcommit(t_{12})			16: delete^{-1}(a,@x, t_1), next=12
19: insert(f,@x, t_1)			
20: subbegin(t_{13})			
21: write(l, t_{13})	l:21		21: write(l, t_{13}), next=18
22: write(k, t_{13})	k:22		22: write(k, t_{13}), next=21
23: write(n, t_{13})	n:23		23: write(n, t_{13}), next=22
24: subcommit(t_{13})			24: insert^{-1}(f,@x, t_1), next=18
25: begin(t_3)			25: begin(t_3), next=nil
26: store(z, t_3)			
27: subbegin(t_{31})			
28: write(q, t_{31})	q:28		28: write(q, t_{31}), next=nil
29: write(r, t_{31})	r:29		29: write(r, t_{31}), next=28
30: subcommit(t_{31})			30: store^{-1}(z, t_3), next=nil
31: insert(h,@z, t_3)			
32: subbegin(t_{32})			
33: write(k, t_{32})	k:33		33: write(k, t_{32}), next=30
34: subcommit(t_{32})			34: insert^{-1}(h, @z, t_3), next=30
35: commit(t_3)			35: commit(t_3)
36: insert(b,@y, t_2)			
37: subbegin(t_{22})			
38: write(l, t_{22})	l:38		38: write(l, t_{22}), next=10
Crash			
Analysis pass: loser$\{t_1, t_2\}$			
redo(4)	p:4		
redo(8)	q:8		
redo(9)	r:9		
redo(11)	r:11		
redo(15)	l:15		
redo(21)	l:21		
redo(22)	k:22		
redo(23)	n:23		
redo(28)	q:28		
redo(29)	r:29		
redo(33)	k:33		
redo(38)	l:38		
39: compensate(38)	l:39		39: CLE(38), next=10
40: compensate(24, t_{13}) $\uparrow t_{14}$			
41: subbegin(t_{14})			
42: write(l, t_{14})	l:42		42: write(l, t_{14}), next=24
43: write(k, t_{14})	k:43		43: write(k, t_{14}), next=42
44: write(n, t_{14})	n:44		44: write(n, t_{14}), next=43
45: subcommit(t_{14})			45: CLE(24, t_{13}, t_{14}), next=18
46: compensate(18, t_{12}) $\uparrow t_{15}$			
47: subbegin(t_{15})			
48: write(l, t_{15})	l:48		48: write(l, t_{15}), next=18
49: subcommit(t_{15})			49: CLE(18, t_{12}, t_{15}), next=12
50: compensate(12, t_{11}) $\uparrow t_{16}$			
51: subbegin(t_{16})			
52: write(p, t_{16})	l:52		52: write(p, t_{16}), next=12
53: write(r, t_{16})	k:53		53: write(r, t_{16}), next=52
54: subcommit(t_{16})			54: CLE(12, t_{11}, t_{16}), next=nil
55: rollback(t_1) complete			55: rollback(t_1)
56: compensate(10, t_{21}) $\uparrow t_{23}$			
57: subbegin(t_{23})			
58: write(q, t_{23})	l:58		58: write(q, t_{23}), next=10
59: write(r, t_{23})	k:59		59: write(r, t_{23}), next=59
60: subcommit(t_{23})			60: CLE(10, t_{21}, t_{23}), next=nil
61: rollback(t_2) complete			61: rollback(t_2)

Figure 2: Completed Figure 14.9

Chapter 15 - Solutions to Exercises

Exercise 15.1 :

Reconsider the two-level action history in Figure 15.7, which was already discussed in Exercise 14.3. It contains operations on records (*store*, $modify$) and index keys (*insert*, *delete*); for the latter operations, the first parameter denotes a key and the second parameter the RID (i.e., address) of a stored record. Assume that the $insert(f, @x, t1)$ operation initiates a split of leaf page l, creating the new leaf page k and posting the split to the parent node n. Discuss to what extent logical log entries for the higher-level operations are feasible for redo purposes. (For undo, such log entries are needed anyway.) Can we avoid creating physiological log entries altogether? Which flush-order dependencies need to be observed for the execution?

As we learned from this chapter logical logging is feasible as long as arising flush-order dependencies are rare, so that only a small amount of memory has to be dedicated to the flush-order graph data, and cyclic flush-order dependencies are rather exceptional. In the case of a cycle we need to take special actions like atomically writing after images for pages involved in the cycle. This would boil down to the same amount of logged information as for physiological logging. So logical logging is beneficial only in specific scenarios like node splitting or copying large objects where these properties are known to hold. Having to dissolve cyclic dependencies from time to time prevents us from completely abandoning physical or physiological log entries.

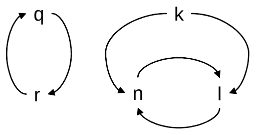


Figure 1: Flush-order graph

Now let us turn to the flush-order dependencies for the example. As the log history contains no read actions we assume that none of the subtransactions is a *blind writer* (see Exercise 3.8), and for simplicity we further assume that each subtransaction reads a page immediately before updating its content, and for the ease of notation we assign the same identifier to both operations in such a pair (e.g., $r_{ijk}(p)$ and $w_{ijk}(p)$). We obtain for each operation f $writeset(f) = readset(f)$. The only exception is the *insert* operation mentioned above where a completely new page k has been created. Thus, we observe:

- $writeset(modify_{11}(x)) = \{p, r\}$ with $r_{111}(p) < w_{111}(p)$
  $r_{112}(r) < w_{112}(r) < w_{312}(r)$
- $writeset(store_{21}(x)) = \{q, r\}$ with $r_{211}(q) < w_{211}(q) < w_{311}(q)$
  $r_{212}(r) < w_{212}(r) < w_{112}(r) < w_{312}(r)$
- $writeset(delete_{12}(a, @x)) = \{l\}$ with $r_{121}(l) < w_{121}(l) < w_{131}(l) < w_{221}(l)$
- $writeset(insert_{13}(f, @x)) = \{k, l, n\}$ with $r_{131}(l) < w_{131}(l) < w_{221}(l)$
  $r_{133}(n) < w_{133}(n)$
- $writeset(store_{31}(z)) = \{q, r\}$ with $r_{311}(q) < w_{311}(q)$
  $r_{312}(r) < w_{312}(r)$

- $writeset(insert_{32}(h, @z)) = \{k\}$ with $r_{321}(k) < w_{321}(k)$
- $writeset(insert_{22}(b, @y)) = \{l\}$ with $r_{221}(l) < w_{221}(l)$

Then we can derive the flush-order graph shown in Figure 1. In this scenario we have to deal with two set of pages involved in cycles; for each of these two sets which we need to create after images and write them to the log atomically.

Exercise 15.2 :

Give all log entries and their proper $NextUndoSeqNo$ backward chains for the nested-transaction scenario of Figure 15.5. Assume that all of t_{11} is executed sequentially in the sense that a new subtransaction begins only after its previously initiated siblings are terminated, and that t_{12} is spawned after the termination of t_{11}. Within t_{12} assume that all subtransactions are spawned asynchronously in separate threads; so at the end of the scenario all subtransactions of t_{12} are simultaneously active. Describe the necessary steps to abort subtransaction t_{12}.

Sequence number: action	Cached changes [PageNo:SeqNo]	Stable changes [PageNo:SeqNo]	Log entry added [LogSeqNo: action] [NextUndoSeqNo]
1: begin(t_1)			1: begin(t_1), next=nil
2: subbegin(t_{11})			2: subbegin(t_{11}), next=1
3: subbegin(t_{111})			3: subbegin(t_{111}), next=2
4: write(a,t_{111})	a:4		4: write(a,t_{111}), next=3
5: write(b,t_{111})	b:5		5: write(b,t_{111}), next=4
6: subcommit(t_{111})			6: subcommit(t_{111}), next=2
7: subbegin(t_{112})			7: subbegin(t_{112}),next=6
8: subbegin(t_{1121})			8: subbegin(t_{1121}), next=7
9: write(c,t_{1121})	c:9		9: write(c,t_{1121}), next=8
10: write(d,t_{1121})	d:10		10: write(d,t_{1121}), next=9
11: subcommit(t_{1121})			11: subcommit(t_{1121}), next=7
12: subbegin(t_{1122})			12: subbegin(t_{1122}), next=11
13: write(e,t_{1122})	e:13		13: write(e,t_{1122}), next=12
14: subcommit(t_{1122})			14: subcommit(t_{1122}), next=8
15: subcommit(t_{112})			15: subcommit(t_{112}), next=7
16: subcommit(t_{11})			16: subcommit(t_{11}), next=2
17: subbegin(t_{12})			17: subbegin(t_{12}), next=16
18: subbegin(t_{121})			18: subbegin(t_{121}), next=17
19: write(f,t_{121})	f:19		19: write(f,t_{121}), next=18
20: subbegin(t_{122})			20: subbegin(t_{122}), next=17
21: subbegin(t_{1221})			21: subbegin(t_{1221}), next=20
22: write(g,t_{1221})	g:22		22: write(g,t_{1221}), next=21
23: write(h,t_{1221})	h:23		23: write(h,t_{1221}), next=22
24: subbegin(t_{1222})			24: subbegin(t_{1222}), next=20
25: write(j,t_{1222})	j:25		25: write(j,t_{1222}), next=24
26: rollback(t_{12})			
27: compensate(25)	j:27		27: compensate(25), next=24
28: compensate(23)	h:28		28: compensate(23), next=22
29: compensate(22)	g:29		29: compensate(22), next=21
30: compensate(19)	f:30		30: compensate(19), next=18
31: rollback(t_{12}) complete			31: rollback(t_{12}) complete

Exercise 15.4 :

Develop pseudocode for the lock conflict test of new transactions that are admitted during the redo pass of a restart, assuming that the `OldestUndoLSN` and the `DirtyPages` data structure has been determined by the analysis pass and serves as an approximative test for "uncritical" pages that require neither redo nor undo steps. Assume that page locking is used, but no explicit locks are reacquired during the restart to keep the overhead low.

With the enhanced redo algorithm the following can be observed: pages that are not among `DirtyPages` are not accessed during the redo pass, and if the page sequence number of such a page is smaller then `OldestUndoLSN` it is not touched by the undo pass either. Thus, a lock for this page can be granted, if no conflicting lock is already held by some other transaction.

```
is-in-conflict(pageno, lock-mode):
  if pageno in DirtyPages or
    there is a conflict lock
  then
    return yes;
  end; /*if*/
  fetch(pageno);
  if DataBaseCache[pageno].PageSeqNo < OldestUndoLSN
  then
    return no;
  else
    return yes;
  end; /*if*/
```

Exercise 15.6 :

Design a scenario, with concrete page numbers, transaction identifiers, log sequence numbers, etc., that shows the need for synchronizing the local log sequence numbers created at different servers of a data-sharing cluster. Construct anomalies that would arise if the global sequence numbers merely were local numbers padded with server identifiers.

Consider the following log history before one of the cluster servers crashes.

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

It consists of two transactions, t_1 and t_2. Assume that they are run by server 1 and 2 respectively. Let n denote the number of individual servers in the cluster, i be a local counter incremented by one upon each local log entry. Assume that unique totally ordered LSNs are generated according to the formula: $lsn := n * i + server\_id - 1$. Hence, server 1 will generate even-numbered LSNs 0, 2, 4, ... whereas server 2 will produce odd-numbered LSNs 1, 3, 5, ...

Then server 1's log looks like: $0: \ begin(t_1) \ \ 2: \ w_1(y) \ \ 4: \ w_1(x) \ \ 6: \ c_1$
And server 2's log is the following: $1: \ begin(t_2) \ \ 3: \ w_2(x) \ \ 5: \ c_2$

Now assume that server 1 fails. During the analysis pass on the merged log the surviving node (server 2) will fetch page x, finding that its page sequence number is 4. The recovery would then errorneously infer that the action with LSN 3 must not be redone. This is incorrect.

Chapter 16 - Solutions to Exercises

Exercise 16.1 :

Reconsider the `MediaRecoveryLSN` for the method based on backups and archive logging. Give concrete examples, with histories referring to concrete page numbers etc., to show each of the following points:

a. It is insufficient to start the redo pass of the media recovery at the most recent begin-backup log entry if this most recent backup was not completed before the media failure occurred.

b. It is insufficient to start the redo pass of the media recovery at the begin-backup log entry of the most recent completed backup if the copying procedure for a backup bypasses the server's page cache.

c. It may be necessary (albeit extremely unlikely) to start the redo pass of the media recovery at the `OldestUndoLSN` as of the time when the most recent complete backup was initiated.

Consider the following sample log history and assume that physiological logging is applied.

$$1\text{:begin}(t_1)\ 2\text{:}w_1(x)\ 3\text{:begin-backup}\ 4\text{:}c_1\ 5\text{:begin}(t_2)\ 5\text{:}w_2(x)$$

Now imagine that the backup process was interrupted by a crash (case a.) and page x is not yet copied. If we start the redo pass with LSN 3 we would erroneously apply the operation with LSN 5 to some out-of-date backup copy of x without repeating the operation with LSN 2, which results in incorrect execution.

But even if the backup procedure were completed (case b.), but the page cache is bypassed, starting the redo pass with LSN 3 could lead to incorrect execution. Imagine that t_1's update on x has not been flushed at the time x is copied to the backup database. Thus, applying operation 5 to the stale version of x is still incorrect.

With fine-grained locking a loser update on a page can be followed by a winner update after an operation's subcommit (sc). Hence, the following scenario is possible:

$$1\text{:}w_{111}(x)\ 2\text{:}sc_{11}\ 3\text{:}w_{211}(x)\ 4\text{:begin-backup}\ 5\text{:}sc_{21}\ 6\text{:}c_2\ 7\text{:end-backup}\ 8\text{:flush(x)}\ 9\text{:}w_{121}(x)$$

where `OldestUndoLSN` is 1 and `SystemRedoLSN` is 9. Now assume that a disk failure occurrs and page x must be recovered. Since the backup copy of x is as of the time immediately before `OldestUndoLSN` we have to start the redo from this log entry.

Exercise 16.2 :

Investigate the mean time to data loss (MTTDL) for mirrored disks as well as for "triple mirroring" where each block is replicated on three different disks. To this end, design a stochastic state-transition model and derive from it an exact formula for the MTTDL, in close analogy to the derivation that we carried out in Section 16.2.

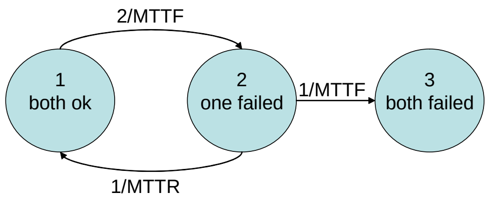


Figure 1: Stochastic state-transition model for "simple mirroring"

The stochastic state-transition model for "simple mirroring" given in Figure 1 leads to the following system of linear equations.

$$\begin{aligned} E_{13} &= H_1 + p_{12}E_{23}, \quad H_1 = \frac{MTTF}{2}, \quad p_{12} = 1 \\ E_{23} &= H_2 + p_{21}E_{13}, \quad H_2 = \frac{MTTF \cdot MTTR}{MTTR + MTTF}, \quad p_{21} = \frac{MTTF}{MTTR + MTTF} \end{aligned}$$

E_{13} is MTTDL. For solving such a system of linear equations you can use, for example, the advanced online solver on *math.com*. So we obtain:

$$MTTDL = \frac{H_1 + H_2 p_{21}}{1 - p_{12}p_{21}} = \frac{MTTF^2 + 3 \cdot MTTF \cdot MTTR}{2 \cdot MTTR} \approx \frac{MTTF^2}{2 \cdot MTTR}$$

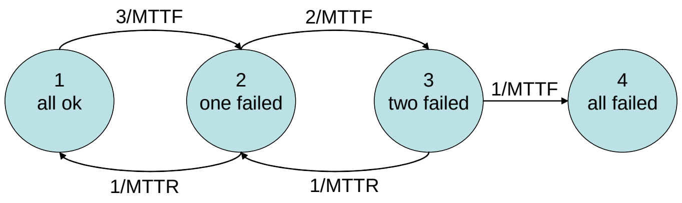


Figure 2: Stochastic state-transition model for "triple mirroring"

Now consider the case of "triple mirroring". The corresponding state-transition diagram is shown in Figure 2. We can derive the following system of linear equations:

$$\begin{aligned} E_{14} &= H_1 + p_{12}E_{24}, \quad H_1 = \frac{MTTF}{3}, \quad p_{12} = 1 \\ E_{24} &= H_2 + p_{21}E_{14} + p_{23}E_{34}, \quad H_2 = \frac{MTTR \cdot MTTF}{MTTF + 2 \cdot MTTR} \\ p_{21} &= \frac{MTTF}{MTTF + 2 \cdot MTTR}, \quad p_{23} = \frac{2 \cdot MTTR}{MTTF + 2 \cdot MTTR} \\ E_{34} &= H_3 + p_{32}E_{24}, \quad H_3 = \frac{MTTR \cdot MTTF}{MTTF + MTTR} \\ p_{32} &= \frac{MTTF}{MTTF + MTTR} \\ MTTDL = E_{14} &= \frac{H_1(1 - p_{23}p_{32}) + H_2 p_{12} + H_3 p_{12} p_{23}}{1 - p_{12}p_{21} - p_{23}p_{32}} = \\ &= \frac{MTTF^3 + 4 \cdot MTTR \cdot MTTF^2 + 11 \cdot MTTR^2 \cdot MTTF}{6 \cdot MTTR^2} \approx \\ &\approx \frac{MTTF^3}{6 \cdot MTTR^2} \end{aligned}$$

Chapter 17 - Solutions to Exercises

Exercise 17.1 :

Extend the pseudocode for stateless queued transactions in a two-tier architecture, given in Section 17.2, to include the case of testable output devices such as automatic teller machines. The extended pseudocode should have the value-added property that, once a user input is successfully enqueued, the corresponding output message is delivered exactly once.

We assume that the client processes output transactions sequentially because we consider the client as a single-user system. We assume further that we have special hardware support for counting messages or dispensed cash in the case of an ATM. Such a counter (let us denote it *hc* for *hardware counter*) is incremented if and only if the corresponding output device really delivered a message (cash). An algorithm for processing output exactly once in this setting is presented in Figure 1. Note that we consider the updating *hc* on the stable storage (as proposed in Section 17.2) as a part of an output transaction.

```
user-output processing by client:
  wait until reply queue is not empty;
  begin transaction;
     dequeue (reply);
     compute output message from reply;
     read (lasthc);
     hc := read current hardware counter from output device;
     while lasthc = hc do
        send output to output device;
        hc := read current hardware counter from output device;
     end /*while*/;
     lasthc := hc;
     write (lasthc);
  commit transaction;
client restart:
  check request queue;
  if not empty then
     initiate processing of requests like during normal operation;
  end /*if*/;
```

Figure 1: Exactly-once output processing

Analysis of various cases (regarding client failures):

1. no failure at all: client dequeues reply, reads the current *hc* as of the last output message, finds that the current *hc* equals the last remembered one, thus sends the output once (and then exits the loop), and writes the, now incremented, *hc*, to stable storage
2. client fails right before dequeue: restart will find the original reply queue and retries the output processing procedure
3. client fails right after dequeue: transactional recovery will restore the reply queue to its original state; so this case is identical to case 2
4. client fails right after sending the output message: transactional recovery will restore the reply queue, then the client will retry the entire output processing, after again reading the current *hc*, there are sub-cases:

   (a) the *hc* is now larger than the remembered counter, so the output has been received by the output device; the output is suppressed, and the new *hc* is written to stable storage (and we obtain the same behavior if the client fails again during this restart processing)
   (b) the *hc* is stil the same as the remembered counter, so the output has not been received by the output device; the output is sent again, the *hc* is read again (now it has been incremented, or the loop body is iterated), and the new *hc* is written to stable storage

5. client fails right after writing the new *hc* to stable storage, but before transaction commit: the *hc* on stable storage is restored to the original state; and when retrying the entire output processing the client will compare this old value of the *hc* to the newly read one; the client will realize that the new *hc* is larger than the remembered one, it will thus suppress the output message, but will try again to save the new *hc* on stable storage (and will eventually succeed once the transaction commits)

Exercise 17.3 :

Consider a conversational client application such as the travel reservation scenario of Section 17.3. Compare the number and data volume of forced log I/Os for running this application as a pseudo-conversational queued transaction chain versus using the general server reply logging method of Section 17.5, to provide application recovery. Assume that the queue manager resides on the data server, and forced logging is required only upon the commit of a transaction.

client	server
1. begin transaction;	
2. enqueue *flight reservation* request;	
3. commit;	
	4. begin transaction;
	5. dequeue *flight reservation* request;
	6. execute *flight reservation* request;
	7. enqueue *flight reservation* reply;
	8. commit;
9. begin transaction;	
10. dequeue *flight reservation* reply;	
11. enqueue *hotel reservation* request;	
12. commit;	
	13. begin transaction;
	14. dequeue *hotel reservation* request;
	15. execute *hotel reservation* request;
	16. enqueue *hotel reservation* reply;
	17. commit;
18. begin transaction;	
19. dequeue *hotel reservation* reply;	
20. enqueue *rental car reservation* request;	
21. commit;	
	22. begin transaction;
	23. dequeue *rental car reservation* request;
	24. execute *rental car reservation* request;
	25. enqueue *rental car reservation* reply;
	26. commit;
27. begin transaction;	
28. dequeue *rental car reservation* reply;	
29. commit;	

Figure 1: Travel reservation with queued transactions.

First consider the case of running the application when the queued transaction paradigm is used. Travel reservation comprises three business transactions initiated by the server (flight reservation, hotel reservation, and rental car reservation) and four request/reply message delivery transactions initiated by the client. Thus, we have at least **7 forced log I/Os**. Note that all forced log I/Os take place on the server because it hosts the queue manager (and the business data). Additionally to conventional server logging which is needed for transaction/crash/media recovery the server now has to store the contents of request and reply messages along with application state for redo in the case of an enqueue operation and for undo in the case of an dequeue operation. Altogether we have six message interchanges (i.e., $6 \times$ *enqueue* and $6 \times$ *dequeue*). With an average message size m and application state size s we obtain a total amount of logging volume of **12(m+s)**.

client	server
1. force-log user input	
2. send *flight reservation* request;	
	3. execute *flight reservation* request;
	4. create *flight reservation* reply log;
	5. force log;
	6. send reply
7. present output;	
8. force-log user input	
9. send *hotel reservation* request;	
	10. execute *hotel reservation* request;
	11. create *hotel reservation* reply log;
	12. force log;
	13. send reply;
14. present output	
15. force-log user input	
16. send *rental car reservation* request;	
	17. execute *rental car reservation* request;
	18. create *rental car reservation* reply log;
	19. force log;
	20. send reply;
21. present output;	

Figure 2: Travel reservation with server reply logging.

When the server reply logging algorithm is used, forced logging is needed only upon server reply on the server side and upon each user input on the client side. As the log information for read operations is rather small or can be derived from undo information if physical logging is applied, the amount of logged information is not that different from the standard data logging. Further we assume that the message size is approximately the same as the user input. As the client can be replayed we need not force application state to the disk. Thus, we obtain **3 forced I/Os** on the client for user input (with client logging volume **3m**) and **3 forced I/Os** on the server for each server reply (sever logging volume **3m**). It is clear that an interactive client can easily sustain three forced I/Os. And the server benefits from significant reduction of logging costs by more than **50%** with regard to the number of forced I/Os and by more than **75%** with regard to the amount of logged information under the assumptions made above.

Exercise 17.4 :

Construct a concrete example, with concrete transaction identifiers, page numbers, etc., that shows the necessity for the data server to log data read operations on behalf of client requests for the general client-server application recovery based on the server reply logging method of Section 17.5.
Hint: Consider, for example, a client request that invokes a stored procedure on the data server whose execution consists of a sequence of transactions.

When a client issues a request invoking a multi-transaction stored procedure then its effects are not isolated from the other concurrent requests running on the server at the same time. This means, that even if the request execution is not complete at the time a server failure occurred, partial effects of the request might be already propagated to the outside world as far as they are reflected in reply messages of some concurrent requests. This implies that the server committed its state including the incomplete request and it must be able to deterministically replay and to complete the request.

$$\begin{array}{ll} \textit{client } 1: & \searrow \\ \textit{server}\ : & r_1(x)w_1(x)c_1r_2(x)r_2(y)w_2(s)c_2r_1(y)w_1(y)c_1\ldots\dagger \\ \textit{client } 2: & \qquad\qquad\qquad\nearrow\qquad\qquad\qquad\quad\searrow \end{array}$$

Figure 3: Problem scenario

Consider the example scenario depicted in Figure 3 with two clients concurrently issuing requests to the server. Assume that data items x, y, z, etc. are items in an online store, and data item s is a shopping cart filled on behalf of client 2. The session of client 1 starts a batch of transactions in order to decide upon a special sales promotion: each item is looked up, and based on its current price, current quantity in stock, recent sales, and pending orders it is decided whether the item is included in the sales promo or not (i.e., the price of the qualifying items is reduced). This decision process may even be context-dependent in the sense that the decision about an item z depends on the previously made decisions about items x and y. For the concrete example, assume that both item x and item y are included in the promo. Concurrently, client 2 places orders for various items. Seeing the reduced price of item x, it decides to purchase an exceptionally large quantity of this item. In addition, it decides to purchase some of item y at the standard price (as this step precedes the price reduction for item y).

Note that this kind of concurrent behavior is, of course, not serializable in terms of the entire client sessions. However, it is serializable in terms of the clients' transactions. This kind of application structuring is not uncommon in practice and is often unavoidable for long-lived workflows. So the application run by client 1 has been intentionally organized as a batch of transactions rather than a single transaction.

Now assume that the server crashes after the last action shown in Figure 3 (the commit of client 1's second transaction); note that this happens before sending the reply message to client 1, but after a reply has been sent to client 2. Upon restart, the server must not undo the actions on behalf of client 2, as it had already sent the reply message and thus committed its state to the external world. Even worse, the server must not undo the actions of the first transaction issued by client 1 as the price reduction for item x affected the session of client 2. So the only solution is to deterministically replay the pre-crash behavior of both sessions. For client 2 this means recovering the database state (if necessary) and recreating the reply message if needed. For client 1 this means recovering the database state (as of the time of the crash) and, in addition, recreating the session state of the interrupted application (i.e., the variables, cursor positions, etc. of a stored procedure or servlet). Note that this is crucial for ensuring correct decision-making regarding the sales promotion while being consistent with the decisions that were already made before the crash and visible to other clients.

Exercise 17.5 :

Discuss possible generalizations of the server reply logging method for general client-server applications towards the following settings:

1. client applications that interact with more than one data server,
2. a data server that does itself invoke requests on another data server, for example, to resolve a higher-level data view or wrapped ADT-style object interface by collecting some of the underlying data from a remote site, or to propagate

high-level updates onto the underlying base data (e.g., by firing remote triggers) including the important case of maintaining replicated data at multiple sites,

3. a three-tier architecture where the applications run on the middle-tier application server.

Hint: Lomet and Weikum (1998) provide some high-level ideas about possible generalizations and also point out difficulties in specific kinds of extensions.

Meanwhile, an algorithm that addresses these issues is presented in:

R. Barga, D. Lomet, G. Weikum, *Recovery Guarantees for General Multi-Tier Applications*, Int. Conference on Data Engineering 2002, San Jose, CA, USA.

Chapter 18 - Solutions to Exercises

Exercise 18.2 :

Give a sample execution of a distributed history s s.t. (i) s itself is not conflict-serializable, but (ii) each local projection of the execution obeys the distributed 2PL protocol. In particular, state the exact order in which a scheduler acquires and releases locks and executes read and write steps.

Example 18.2 in the book demonstrates that applying 2PL to local projections without synchronization of "lock points" may lead to globally non-serializable schedules.

Server 1:	$rl_1(x)r_1(x)ru_1(x)$		$wl_2(x)w_2(x)$	
Server 2:		$rl_2(y)r_2(y)ru_2(y)$		$wl_1(y)w_1(y)$

Although both transactions in the example above locally acquire locks according to the 2PL protocol, the conflict graph of the corresponding global schedule has a cycle $t_1 \rightarrow t_2 \rightarrow t_1$.

Exercise 18.3 :

Consider the distributed wait-for graph for transactions t_1 through t_6 running on servers A, B, and C shown in Figure 18.10. Assume that transaction t_1 now requests a lock on server A for which transaction t_2 already holds an incompatible lock. Simulate the path pushing algorithm for deadlock detection at this point, and give the resulting messages.

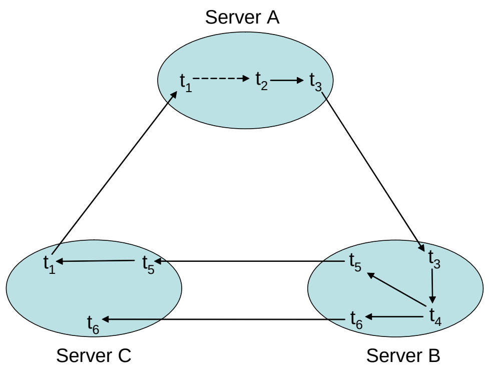


Figure 1: Distributed WFG

When the edge $t_1 \rightarrow t_2$ is added on server A, a new waits-for path (from t_1 to t_3) arises with both incoming and outgoing waits-for message edges. Thus, this path is pushed to server B. On server B, in turn, we obtain two paths which have to be forwarded to server C: $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow t_4 \rightarrow t_5$ and $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow t_4 \rightarrow t_6$. Finally on server C we detect the cycle $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow t_4 \rightarrow t_5 \rightarrow t_1$. Note that the path forwarding terminates on server C, because t_6 does not have an outgoing waits-for message edge, and $ts(t_5) > ts(t_1)$ holds for the edge $t_5 \rightarrow t_1$.

Exercise 18.4 :

Apply the optimistic ticket method (OTM) to the local histories from Examples 18.8, 18.11, and 18.12 and show how global serializability can be obtained in each of these cases.

In each case we assume that the initial ticket value is zero ($I_j[0]$).

- **Example 18.8** with OTM
  $s_1 = r_1(I_1[0])w_1(I_1[1])r_1(a)r_3(a)r_3(b)w_3(a)w_3(b)r_2(I_1[1])w_2(I_1[2])r_2(b)$
  $s_2 = r_2(I_2[0])w_2(I_2[1])r_2(c)r_4(c)r_4(d)w_4(c)w_4(d)r_1(I_2[1])w_1(I_2[2])r_1(d)$

  When one of the two global transactions issues a commit request the GTM will check whether the committing transactio is involved in a cycle. Let $v_i(I_j)$ denote a ticket value obtained by the global transaction t_i on the site j. For this example we obtain a global cycle indicated by $v_1(I_1) < v_2(I_1)$ and $v_2(I_2) < v_1(I_2)$, and one of the global transactions must be aborted.

- **Example 18.11** with OTM
  $s_1 = w_1(a)r_1(I_1[0])w_1(I_1[1])c_1r_3(a)r_3(b)c_3r_2(I_1[1])w_2(I_1[2])w_2(b)c_2$
  $s_2 = r_2(I_2[0])w_2(I_2[1])w_2(c)c_2r_4(c)r_4(d)c_4r_1(I_2[1])w_1(I_2[2])w_1(d)c_1$

  Here the GTM has to deal with a similar situation with regard to ticket values as in Example 18.8, so that a global cycle in the conflict graph is detected again.

- **Example 18.12** with OTM
  $s_1 = r_1(I_1[0])w_1(I_1[1])r_1(a)w_3(a)w_3(b)r_2(I_1[1])w_2(I_1[2])r_2(b)c_1c_3c_2$
  $s_2 = w_4(c)r_1(I_2[0])w_1(I_2[1])r_1(c)r_2(I_2[1])w_2(I_2[2])r_2(d)w_4(d)c_2c_4c_1$

  In this example a non-CSR execution cannot be detected by the GTM, because the ticket values on both sites are now in the same order. But, fortunately, the local scheduler on site 2 is able to detect this pathological situation by observing a local cycle $t_4 \to t_1 \to t_2 \to t_4$. Then the local scheduler decides to abort one of the transactions. If the local scheduler were aware of the presence of global transactions, it would rather abort a local transaction because it is much easier than having to abort the local subtransactions of a global transaction at all sites where it is executed.

Exercise 18.7 :

Find a sample history proving that the containment COCSR $\subset$ ECOCSR is proper.

Consider the local databases $D_1 = \{a\}$ and $D_2 = \{b\}$ with the corresponding local histories $s_1 = r_1(a)c_1r_3(a)w_4(a)c_4c_3w_2(a)c_2$ and $s_2 = w_1(b)r_2(b)c_1c_2$. The global transactions (t_1 and t_2) obey the commit ordering rule, whereas for the local transactions (t_3 and t_4) on site 1 we obtain $r_3(a) <_{s_1} w_4(a)$ but $c_4 <_{s_1} c_3$. So the global schedule $s_1 \cup s_2$ is in ECOCSR but not in COCSR.

Exercise 18.8 :

Discuss appropriate points for the take-a-ticket operation during the execution of a global transaction's subtransaction under the assumption that the underlying server uses a) 2PL, b) strict 2PL, c) strong 2PL, d) BOCC, e) FOCC, f) ROMV (see Chapters 4 and 5) for its local schedule. Consider the possibility that other servers on which the global transaction executes may use different protocols.

a) With 2PL protocol the GTM should issue *take-a-ticket* requests at the transaction's lock point, i.e., right before releasing the first lock on one of the underlying servers. This way, local lock on the ticket objects are held for a minimum duration but still serve to "check" and "manifest" the local serialization order and ensure global serialization.

b) Strict 2PL (S2PL) generates only strict schedules as it postpones releasing write locks until the end of transaction. Since ST is a proper subset of ACA and $Gen(S2PL)$ is a proper subset CSR, we obtain $Gen(S2PL) \subset ACA \cap CSR$. Now we can apply Theorem 18.7 and use the implicit ticket method.

c) Strong 2PL (SS2PL) produces only rigorous schedules as both, read and write locks are not released until the end of transaction, and as has been proven by Theorem 18.4, with commit-deferred transactions the global serilazability is guaranteed even without the ticket-method.

d) BOCC generates only rigorous schedules as well, which can be observed by the following: consider the committed projection of a schedule s produced by a BOCC scheduler. If there is a conflict of the form $w_i(x) \rightarrow r_j(x)$, then $c_i <_s r_j(x)$ because otherwise t_j could not be validated (i.e., committed). If there is a conflict of the form $w_i(x) \rightarrow w_j(x)$, then the val-write phase of t_i precedes the val-write phase of t_j (i.e., $c_i <_s w_j(x)$). If there is a conflict of the form $r_i(x) \rightarrow w_j(x)$, then t_i must have been validated first, and thus $c_i <_s w_j(x)$. This way we showed that there is no need for *take-a-ticket* request, when BOCC is used.

e) As already noticed in Chapter 4 FOCC generates COCSR schedules as it validates transactions against running transactions and the validation order corresponds to the commit order. Thus, no *take-a-ticket* operation is necessary with this protocol either.

When servers use different protocols, the GTM should issue the *take-a-ticket* request at the most appropriate point for each server separately depending on the particular protocol of the corresponding server.

Exercise 18.10 :

Discuss the use of the optimistic ticket method for servers that merely guarantee local snapshot isolation, as opposed to local conflict serializability (see Chapter 10 for the notion of snapshot isolation). What global correctness criteria can be guaranteed this way? What are the performance implications?

The following paper contains a detailed discussion of these questions:

R. Schenkel, G. Weikum, *Integrating Snapshot Isolation Into Transactional Federations*, 5th IFCIS International Conference on Cooperative Information Systems (CoopIS), Eilat, Israel, September 2000.

Exercise 18.11 :

Consider a data sharing system with three servers A, B, and C. Suppose that server C is the home of pages a, b, c, and d, and that these pages are dynamically accessed during the execution of transactions on servers A and B. Give the necessary messages between these servers under a page oriented callback locking protocol for the following distributed history:

A: $r_1(a)w_1(a)r_1(c)$ $c_1r_3(a)w_3(a)c_3$ $r_6(b)r_6(d)c_6$
B: $r_2(c)c_2$ $r_4(a)r_4(b)r_4(d)c_4r_5(b)w_5(b)r_5(d)c_5$

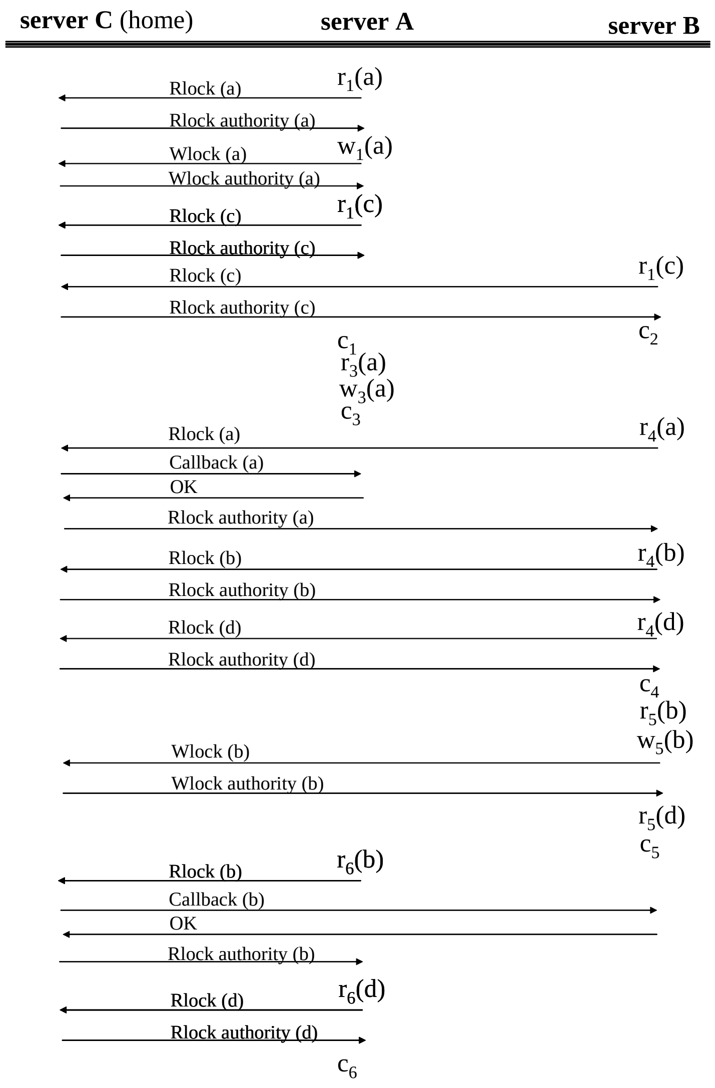


Figure 2: Message sequence diagram for callback locking

Note that Figure 2 shows lock requests only when the requesting site does not have the corresponding lock authority at the time of the request.

Chapter 19 - Solutions to Exercises

Exercise 19.2 :

Consider the following scenario. A transaction is initiated from a PC to make reservations for a vacation trip. The initiator communicates directly with a travel agency and a rental car company. The travel agency communicates with a travel wholesaler, which books two of the necessary flights (or flight legs) on the servers of the corresponding airlines, and also with two other airlines to book the rest of the necessary flights. All these steps belong to one distributed transaction, and all involved participants are able and willing to commit the transaction. The communication structure of this transaction is illustrated in the following figure.

(a) Give the message flow and log entries for the hierarchical 2PC protocol for this scenario. Indicate which log entries need to be immediately forced to the stable log.

(b) Which of the various optimizations that we discussed in this chapter are applicable and promising for the given scenario? What are the implications for the protocol's message and logging costs?

(c) How does the protocol and its execution cost change if the coordinator role were transferred to the server of the travel agency (at the time when the initiator issues its commit request)?

(d) Give the message flow and log entries for the case that the server of airline 3 is unable to commit and votes "no" in the hierarchical 2PC protocol.

(e) Apply the presumed-abort protocol to the case where the server of airline 3 votes "no". Give the resulting message flow and log entries.

(a) Figure 1 on page 4 depicts the message flow and the log entries for this scenario. We use the following abbreviations: RCC for rental car company, TA for travel agency, TW for travel wholesaler and AL for airline. The total number of messages is 28, the total number of log entries is 18, and all of them must be forced to disk immediately.

(b) Due to the fact that all sites participating in the distributed transaction are going to commit and execute update operations, the most promising optimization is the *presumed commit* 2PC protocol. Under this protocol only the top-level coordinator (i.e., the Client) must force-write the commit log entry. Hence we save *seven* forced writes. Moreover, since winner transactions do not have to send *ack* messages, we also save *seven* messages.

We could also consider the *presumed-abort* 2PC protocol with omission of the *begin* log entry on each (sub)coordinator. But in this case we would save only *three* forced writes, and there is no savings in terms of communication costs.

(c) When the coordinator role is with the travel agency, the corresponding node becomes the root of the communication tree. And the client node is converted to a child node of the travel agency node. This way the tree decreases in height from *four* nodes to *three* nodes, but the number of (sub)coordinators and the number of participants do not change. Thus the execution cost in terms of messages and forced writes does not change, but due to the lower tree height the execution is more parallelized and is slightly faster in real time.

(d) Figure 2 on page 5 depicts the necessary log entries and messages for this scenario.

(e) The execution under the *presumed-abort* 2PC protocol is shown in Figure 3 on page 6. So *begin* log entries and *ack* messages are abandoned. And *rollback* log entries are written in lazy manner. We can actually also avoid creating *rollback* entries at all, since it fits perfectly with the abort presumption.

Exercise 19.3 :

Consider a binary, perfectly balanced tree of processes of height n where all leaf nodes have the same distance from the root; so there is a total number of $m = 2^n - 1$ nodes in the tree. Assume that the root is the coordinator of the commit protocol. Determine the number of messages and forced log writes for the presumed-nothing (i.e., basic 2PC), presumed-abort, and presumed-commit protocols for the following situations:

(a) all processes have performed updates and the transaction commits,

(b) all processes have performed updates and the transaction aborts,

(c) all nonroot nodes are read-only and the transaction commits,

(d) all leaf nodes are read-only and the transaction commits.

With the **basic 2PC** protocol the nonleaf nodes ($m - 2^{n-1} = 2^{n-1} - 1$) force-write a *begin* log entry. All participants, i.e., all nodes except the root, ($m - 1 = 2^n - 2$) force-write a *prepared* log entry. And finally all m nodes force-write a *commit* log entry. Thus, we obtain in total $5 \cdot 2^{n-1} - 4$ forced writes. And each participant (i.e., each nonroot node) exchanges 4 messages with its coordinator. So the total number of messages is $4 \cdot 2^n - 8$. Since basic 2PC does not include any optimizations, the calculations above are valid for all of the four scenarios.

(a) In this scenario we cannot benefit from using the **presumed-abort 2PC** protocol, because it is geared only for loser transactions. Thus, we obtain identical costs as with the basic 2PC protocol. But if the begin log entry is written in a non-forced manner or completely dropped, then we safe as many forced writes as there are (sub)coordinators, i.e., the number of nonleaf nodes. So the total number of forced writes then is $(5 \cdot 2^{n-1} - 4) - (2^{n-1} - 1) = 3 \cdot 2^{n-1} - 3$

Now consider the **presumed-commit 2PC** protocol. Here we save in comparison with basic 2PC forced writes of the *commit* log entry on all participating sites. Thus we obtain $(5 \cdot 2^{n-1} - 4) - (2^n - 2) = 3 \cdot 2^{n-1} - 2$ forced writes. We also save an *ack* message per participant. So the overall number of messages is $(4 \cdot 2^n - 8) - (2^n - 2) = 3 \cdot 2^n - 6$

(b) In this scenario the costs of **presumed-commit 2PC** are identical to the costs of **basic 2PC**.

The following costs arise with **presumed-abort 2PC**. Only participants force-write a *prepared* log entry. Hence, the total number of forced writes is $2^n - 2$. Again we can abandon *ack* messages. $3 \cdot 2^n - 6$ messages are still required.

(c) Consider **presumed-commit 2PC** first. We still need one forced write for the *begin* log entry on each (sub)coordinator node (i.e., $2^{n-1} - 1$ forced writes). And there are only two messages to exchange for each participant (i.e., altogether $2 \cdot 2^n - 4$).

With **presumed-abort 2PC** we obtain zero logging costs and the same amount of messaging as when presumed-commit 2PC is used.

(d) Let us assume that all inner nodes perform local updates. Otherwise this scenario boils down to case (c).

With **presumed-commit 2PC**, additionally to (a) we save one forced write for the *prepared* log entry on each leaf node. Thus, the overall number of forced writes is $(3 \cdot 2^{n-1} - 2) - 2^{n-1} = 2^n - 2$. We also save one commit message for each leaf node in comparison to (a) (i.e., the total number of messages is $(3 \cdot 2^n - 6) - 2^{n-1} = 5 \cdot 2^{n-1} - 6$).

With **presumed-abort 2PC**, additionally to (a) we save the *prepared* and the *commit* log entry on each leaf node. So the overall number of forced writes amounts to $(3 \cdot 2^{n-1} - 3) - 2 \cdot 2^{n-1} = 2^{n-1} - 3$. And we also save one *commit* message and one *ack* message for each leaf node. So the total number of messages is $(4 \cdot 2^n - 8) - 2 \cdot 2^{n-1} = 3 \cdot 2^n - 8$

The table in Figure 1 summarizes the results presented above. With PN, PA and PC we abbreviate *presumed-nothing*, *presumed-abort* and *presumed-commit*, respectively.

	forced log writes				messages			
	a	b	c	d	a	b	c	d
PN	$5 \cdot 2^{n-1} - 4$	$5 \cdot 2^{n-1} - 4$	$5 \cdot 2^{n-1} - 4$	$5 \cdot 2^{n-1} - 4$	$4 \cdot 2^n - 8$	$4 \cdot 2^n - 8$	$4 \cdot 2^n - 8$	$4 \cdot 2^n - 8$
PA	$3 \cdot 2^{n-1} - 3$	$2^n - 2$	0	$2^{n-1} - 3$	$4 \cdot 2^n - 8$	$3 \cdot 2^n - 6$	$2 \cdot 2^n - 4$	$3 \cdot 2^n - 8$
PC	$3 \cdot 2^{n-1} - 2$	$5 \cdot 2^{n-1} - 4$	$2^{n-1} - 1$	$2^n - 2$	$3 \cdot 2^n - 6$	$4 \cdot 2^n - 8$	$2 \cdot 2^n - 4$	$5 \cdot 2^{n-1} - 6$

Figure 1: Logging and messaging costs for 2PC variants.

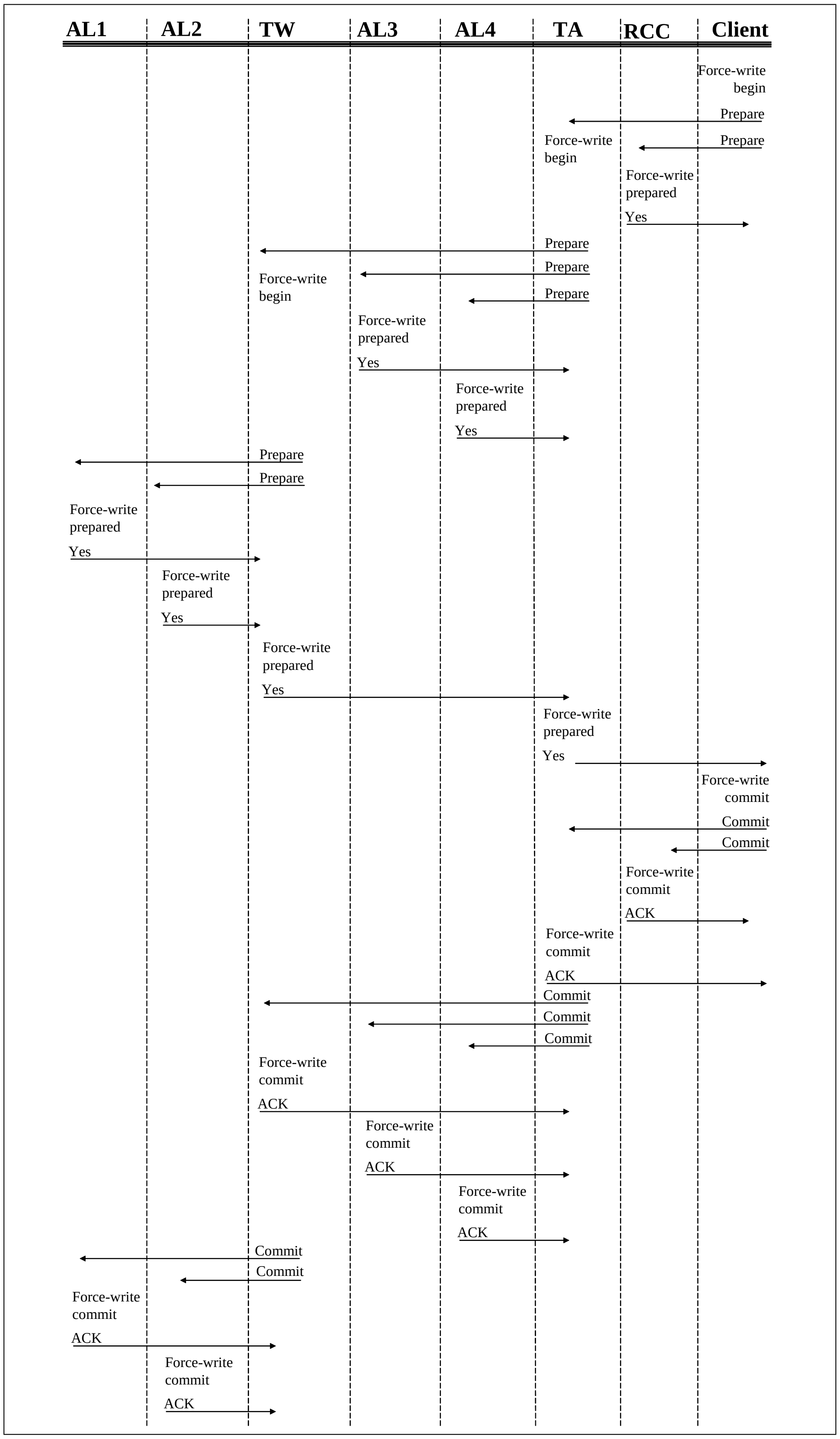


Figure 1: Message flow and log entries for case (a) of Exercise 19.2

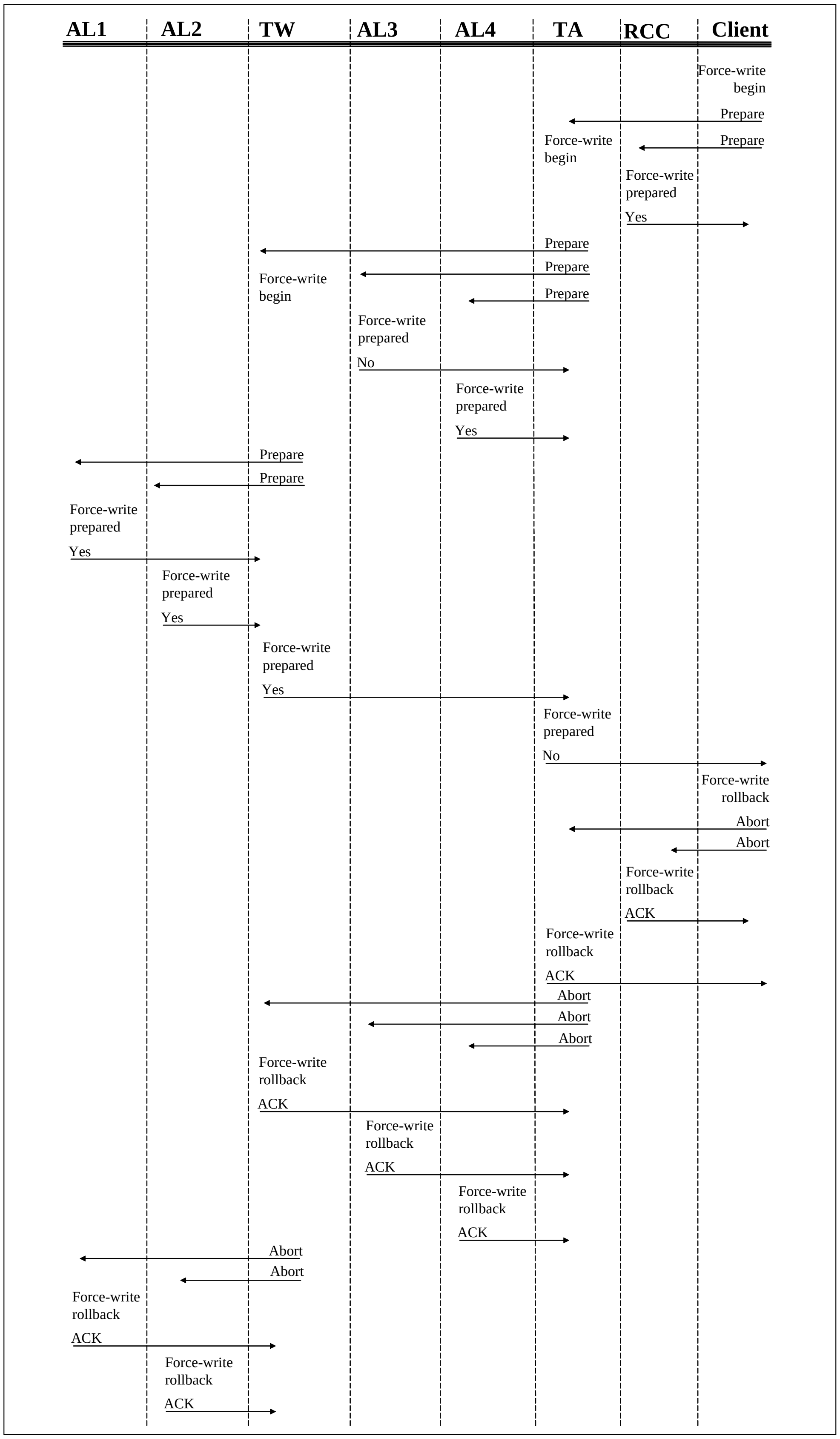


Figure 2: Message flow and log entries for case (d) of Exercise 19.2

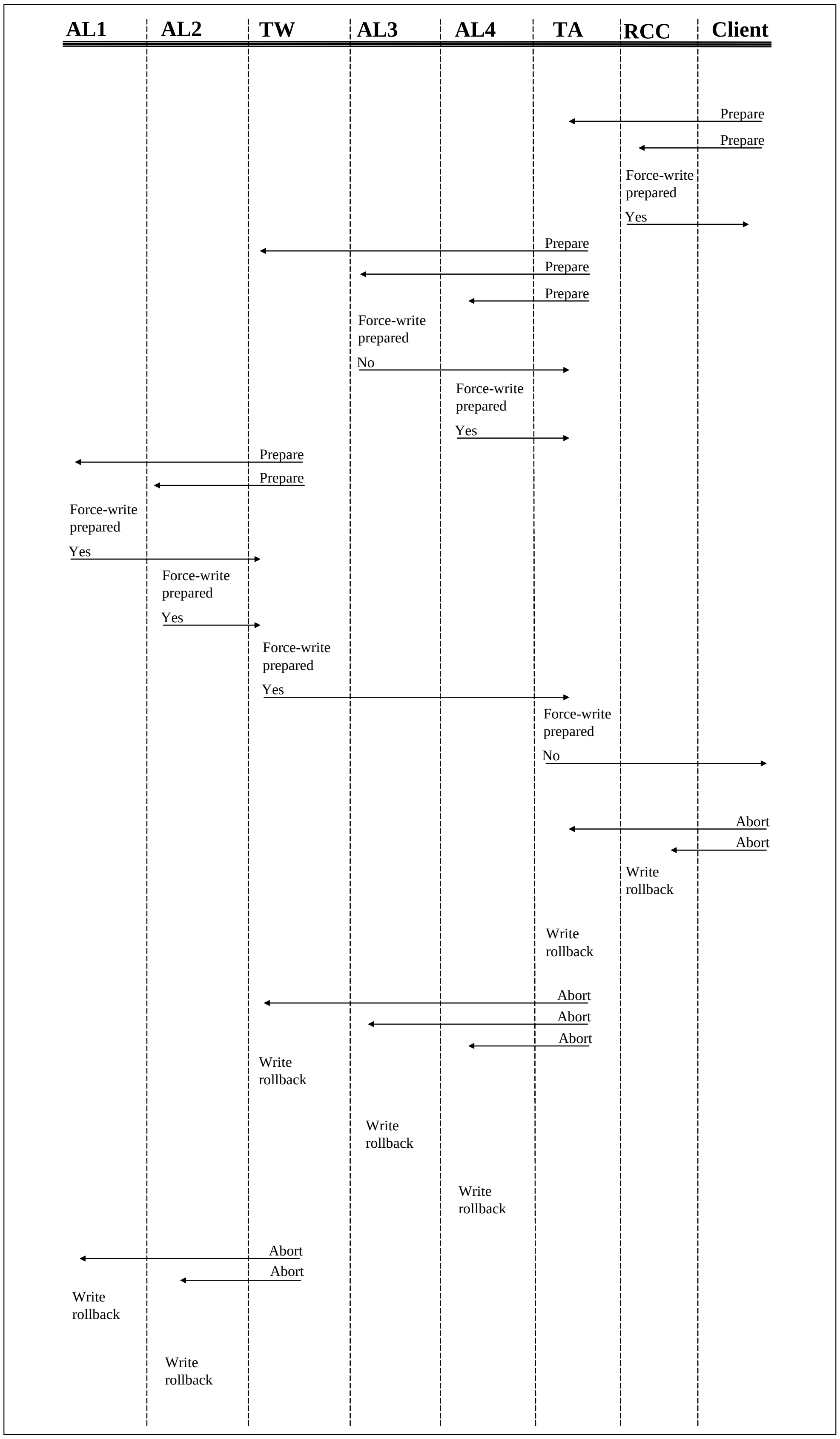


Figure 3: Message flow and log entries for case (e) of Exercise 19.2