UNIVERSITÄT DES SAARLANDES

Fachrichtung 6.2 – Informatik

Lehrstuhl Prof. Weikum

Transaktionsorientierte Informationssysteme SS 2000

Übung 5 Beispiellösung

Aufgabe 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?

When a transaction starts executing an operation on level l_i, it first acquires a corresponding lock on this level. At this time, the transaction does not hold any locks in lower levels. After that, the execution begins, submitting operations to level l_{i+1}. After the execution is finished, all locks in level l_{i+1} are freed, while the lock in level l_i is held until the operation in the level above is finished. Based on this observation and using an induction argument, we can conclude that when a transaction is blocked in level l_i, it doesn't hold any locks in lower levels, so it can't block any other transaction there.

Now assume that we have a 'global ' deadlock, such that t_1 waits for t_2 in l_{i1}, and so on, and t_n waits for t_1 in l_{in}, with $n \geq 2$. Then t_2 holds a lock in l_{i1} and requests a lock in l_{i2}. By what we said before, we have $i_1 \leq i_2$. By induction, we get $i_1 \leq i_2 \cdots \leq i_n$ and, since the same argument holds for t_n and t_1, $i_n \leq i_1$. But then all levels are indeed the same, so we showed that only level-internal deadlocks can happen.

When we allow non-layered schedules, we must consider global deadlocks. For example, consider a 2-level architecture with a record and a page level. Transaction t_1 first reads page p (and therefore locks it before). Then t_2 executes a record-level $op(x)$ operation that in turn reads and tries to write page p, but is blocked when requesting a write lock for p. After that, t_1 tries to execute a record-level $op(x)$ operation that is not compatible with t_2's previous operation, hence t_1 is blocked. But t_1 still holds its lock on p, so we have a global deadlock.

Aufgabe 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 7.5. Does it have performance advantages to have all three levels involved in locking rather than only two? If locking should be limited to two levels, are there arguments on which levels should be chosen?

It is clear that transactions hold locks shorter (in a specific level) if there are locks in another level above it. Locks can be released after an operation in this upper level is finished, and needn't be held until the end of the transaction. Obviously, this is an useful property, so it is better to have more levels.

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

In the example, we have the choice between query-level locks and record-level locks (page level locks are always required). If we chose to acquire record-level locks, those 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 those pages, but are not in conflict with those accessed by the first transaction, can therefore resume there work quickly. If we use query-level locks, pge-level locks are held much longer (especially for long queries like table scans), so concurrent transactions may be blocked longer.

As a rule of thumb, it is often more useful to use locks in lower levels than in higher levels. (Unless, of course, all page-level conflicts match exactly with conflicts in higher levels; if that is the case, the early release of page-level locks won't give any advantage at all.)

Aufgabe 3:

Trace the lock requests, conflict tests, lock acquisitions, and lock releases for the execution of the schedule in Figure 7.6 under the general object-model 2PL with a SS2PL variant. How would the example execution change if the append operations were dropped and t2's decrement operation were replaced by an incr(a) operation that commutes with other increments? How would the execution change if t2 started first and t1 were spawned after all of t2's operations but before the commit of t2?

Aufgabe 4:

Develop pseudocode for a restricted version of the object-model 2PL, where a generalized form of 2-level schedules is considered with some transactions bypassing the object level L1 and invoking L0 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.

For a quick introduction in the problems arising with concurrency control in transactional federations, you may consult our TDD'99 paper available from our website. You will find references to other, more general literature there.

The following solution is proposed in the manuscript: After an object-level operation of a global transaction (a so-called global subtransaction) is finished, its page-level locks are not released, but converted into a '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 in a system, we have to distinct between local and global subtransactions, and we must know about the commit (or abort) of a global transaction. Additionally, we introduce two new lock modes *retained read* and *retained write*. All transactions acquire 'read' and 'write' locks. Upon the commit of a global subtransaction, its locks are converted into retained locks. When its parent global transaction commits (or aborts), its locks are released.

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	-	-	-	-
ret. read	+	+	+	-
ret. write	+	+	-	-

The following pseudocode represents the necessary steps. (Note that the Begin operations can be easily optimized out.)

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

BeginGlobalSubtransaction(GlobalTransactionID, SubtransactionID)
{ add subtransaction to global transaction
      in list of global transactions;
}

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;
}
```

The implementation of multilevel transaction management in the *TraFIC* prototype provides this kind of interface to support local transactions in the federated database system.

Aufgabe 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.

Aufgabe 6:

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

Aufgabe 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 7.8.

	x.inf	x.sup	
	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	

Aufgabe 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	
	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	

Aufgabe 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, ϵ, δ):
if x ≥ ϵ then x := x - δ 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 round 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, too. 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 $\epsilon := x.low + \delta$, we get exactly the same semantics as with the operations without the extra condition.

Let's consider conditional_decr. In the escrow test, we have to compare the current infimum $x.inf$ of the counter with ϵ, the bound it is expected to hold. If it is larger than or equal to ϵ, we can execute the decrememt and update $x.inf$ and x. If $x.inf$ is less than ϵ, but $x.sup$ is greater or equal, 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.