UNIVERSITÄT DES SAARLANDES

Fachrichtung 6.2 – Informatik

Lehrstuhl Prof. Weikum

Transaktionsorientierte Informationssysteme SS 2000

Übung 4 Beispiellösung

Aufgabe 1:

Consider the following 2-level schedule consisting of two funds-transfer transactions and a balance-lookup transaction. Assume that the getbalance operations do not commute with withdraw operations not with deposits. 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:

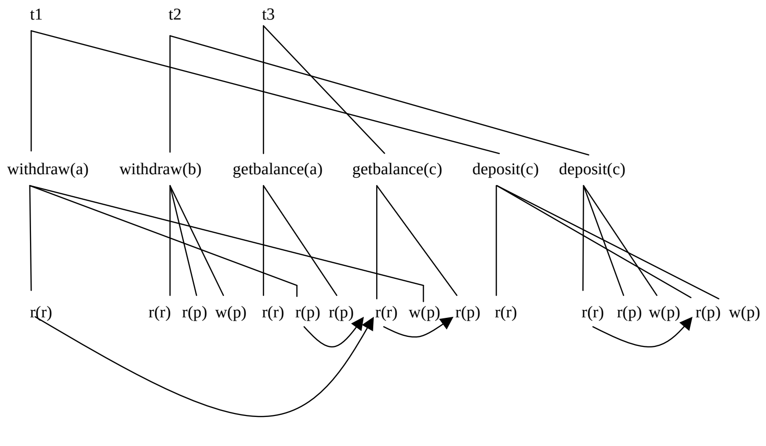


Abbildung 1:

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:

Now all subtrees are isolated, we can prune them and get the equivalent schedule $t_3t_2t_1$.

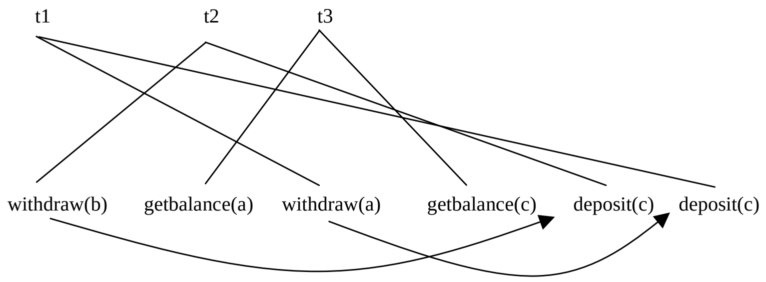


Abbildung 2:

Aufgabe 2:

Consider a data server with a 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 (*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 of these transactions as 3-level transactions, and discuss possible schedules for them. Give (non-trivial, e.g., non-serial) examples for 3-level schedules that are a) tree-reducible and b) not tree-reducible and thus not admissible.

We start with modelling the transaction that finds all persons, which is depicted in Figure 2:

Analogously, we model the second transaction in the Figure 2:

Aufgabe 3:

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.

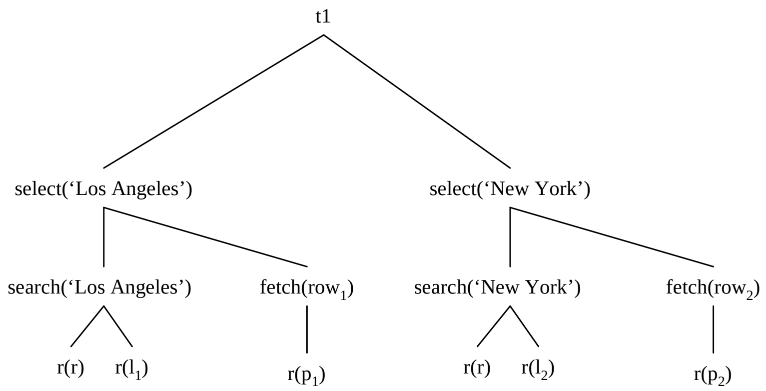


Abbildung 3:

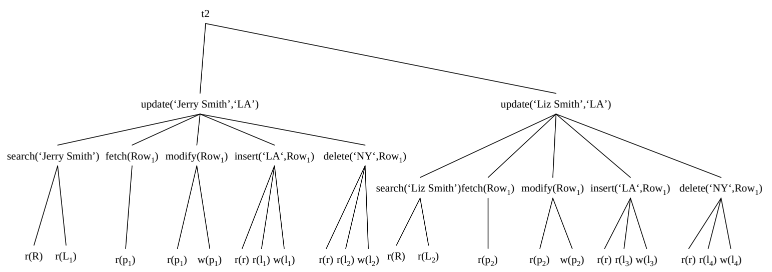


Abbildung 4:

Aufgabe 4:

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. Analogously, *decrement* subtracts a given value from the counter or returns 'no' if the lower bound became 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.

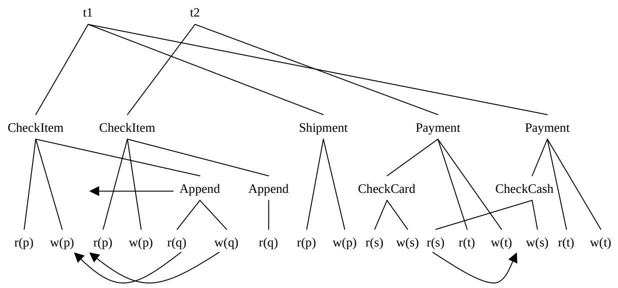
	$inc \uparrow ok$	$inc \uparrow no$	$dec \uparrow ok$	$dec \uparrow no$	$gv \uparrow res$
$inc \uparrow ok$	+	-	-	+	-
$inc \uparrow no$	+	+	-	+	+
$dec \uparrow ok$	-	+	+	-	-
$dec \uparrow no$	-	+	+	+	+
$gv \uparrow res$	-	+	-	+	+

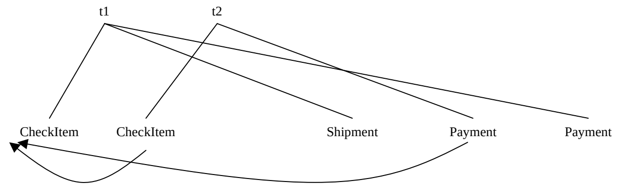
Aufgabe 5:

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

The schedule

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





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.

Aufgabe 6:

Construct the return-value commutativity table for a semiqueue. Show, by means of example schedules, that semiqueues allow higher concurrency than FIFO queues.

The key difference between regular queues and semiqueues is that order of enqueue and dequeue operations doesn't matter, unless the queue becomes empty. Therefore, enqueues and dequeues can always be executed if the queue is not empty, so semiqueues allow a much higher concurrency. The following return-value commutativity table for semiqueues reflects this:

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

Aufgabe 7:

Consider the following execution of operations on an initially empty queue q where a, b, and c are entries added into 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 serialzable if we assume general commutativity. For example, it is not possible to commute t_1's last enqueue operation outside 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 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.)

Aufgabe 8:

None of the participants presented a solution.

Aufgabe 9:

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. One could the 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. (*Code skipped here*). Give an example schedule with these operations that would be considered conflict-serializable relative to the compatibility of these operations, but is not conflict-serializable under a strict interpretation of commutativity. Discuss the acceptability of the resulting (worst-case) situation 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 may result 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 has a balance of $-10. While the bank would easily accept such an anomality, 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.)