Transaktionsorientierte Informationssysteme SS 2000

Übung 2 Beispiellösung

Aufgabe 1:

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

$s_1 = w_1(x)r_2(y)r_1(x)c_1r_2(x)w_2(y)c_2$
$s_4 = r_1(x)r_2(x)w_3(x)w_4(x)w_1(x)c_1w_2(x)c_2c_3c_4$

a) Lösung für den Schedule s_1:

2PL: $wl_1(x)w_1(x)rl_2(y)r_2(y)r_1(y)wu_1(x)c_1rl_2(x)r_2(x)wl_2(y)w_2(y)wu_2(y)ru_2(x)c_2$

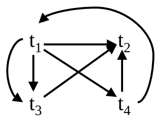
S2PL, SS2PL: identisch

TO: Man sieht, dass bei allen Operationen der TO-Test funktioniert, der Ausgabeschedule sieht also genauso aus wie der Eingabeschedule.

SGT : Die Ausführung kann laufen, bis t_1 sein Commit ausgeführt hat. Jetzt wird t_1 aus dem Serialisierungsgraphen entfernt, weil es committed ist und keine eingehende Kante hat. Anschließend ist nur noch t_2 im Graphen und aktiv, also kann es keinen Zyklus mehr geben. Unter dem SGT-Protokoll sieht der Ausgabeschedule also genauso aus wie der Eingabeschedule.

b) Lösung für den Schedule s_4:

2PL: Zunächst können die folgenden Operationen ablaufen:
$rl_1(x)r_1(x)rl_2(x)r_2(x)$
Nun fordern nacheinander t_3, t_4 und t_1 eine Schreibsperre auf x an, die nicht gewährt werden kann. Der Wartegraph sieht danach so aus:



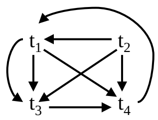
Wegen des Zyklus im Wartegraph wird z.B. t_1 abgebrochen, die Sperranforderungen von t_3 und t_4 können aber wegen der Lesesperre, die t_2 auf x hält, nicht gewährt werden. Anschließend fordert t_2

eine Schreibsperre auf x an, die gewährt werden kann. Der folgende Schedule sieht wiefolgt aus:
$wl_2(x)w_2(x)wu_2(x)c_2wl_3(x)w_3(x)wu_3(x)c_3wl_4(x)w_4(x)wu_4(x)c_4$

S2PL, SS2PL: identisch

TO: Bis zum Abschluss der Ausführung von $w_4(x)$ ergibt sich kein Problem. Wenn $w_1(x)$ ausgeführt werden soll, stellt der Scheduler fest, dass t_1 "zu alt"ist, um x zu schreiben, da bereits vorher die "jüngere" t_4 x geschrieben hatte, also wird t_1 abgebrochen. Das gleiche passiert, wenn anschließend t_2 x schreiben will. Der Gesamtschedule sieht also folgendermaßen aus:
$r_1(x)r_2(x)w_3(x)w_4(x)a_1a_2c_3c_4$

SGT: Bis zum Abschluss der Ausführung von $w_4(x)$ ergibt sich kein Zyklus im Serialisierungsgraphen. Wenn $w_1(x)$ ausgeführt werden soll, schließen sich mehrere Zyklen auf einmal:



Transaktion t_1 wird abgebrochen. Anschließend kann die Ausführung bis zum Ende durchlaufen, ohne dass ein weiterer Zyklus geschlossen wird. Der Gesamtschedule sieht also wiefolgt aus:
$r_1(x)r_2(x)w_3(x)w_4(x)a_1w_2(x)c_2c_3c_4$

Aufgabe 2:

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 gets again involved in a deadlock, is chosen as the victim and aborted. and so on. Provide a concrete example for such a situation, and describe how 2PL could be extended in order to avoid starvation of transactions.

Wir hatten das Problem bereits in der Vorlesung diskutiert. Eine der möglichen Lösungen besteht darin, bei der Opferauswahl das Alter einer Transaktion zu berücksichtigen und die jüngste Transaktion zurückzusetzen. Dabei beginnt man mit dem Zählen des Alters zu dem Zeitpunkt, wo die Transaktion zum ersten Mal gestartet wurde. Nachfolgende Starts nach Abbrüchen wegen der Auswahl als Opfer zählen nicht mit. Irgendwann ist die Transaktion dann einmal älter als alle anderen, die am Deadlock beteiligt sind, und kann zum Ende kommen.

Aufgabe 3:

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

Wir betrachten beispielhaft *wound-wait*, *wait-die* folgt analog. Man sieht zunächst einfach, dass eine Transaktion immer nur von einer älteren Transaktion blockiert werden kann. Falls also t_i von t_j blockiert wird, dann gilt $ts(t_i) > ts(t_j)$. Nehmen wir also an, es gebe einen Zyklus $t_1, \cdots, t_n = t_1$ im Wartegraphen. Dann sieht man sofort, dass dann auch $ts(t_1) > ts(t_1)$ gelten muss, was offenbar nicht sein kann.

Aufgabe 4:

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

a) Wir zeigen zunächst, dass $Gen(BTO) \not\subset Gen(2PL)$. Dazu betrachten wir den folgenden Schedule, der zwar von 2PL, aber nicht von BTO erzeugt werden kann:

$r_1(z)r_2(y)w_2(x)c_2w_1(x)c_1$

Unter dem BTO-Protokoll wird t_1 abgebrochen, wenn es versucht, x zu schreiben, da vorher bereits t_2 mit der größeren Zeitmarke x geschrieben hat. Unter dem 2PL-Protokoll kann dieser Schedule dagegen problemlos erzeugt werden.

b) Nun zeigen wir, dass $Gen(2PL) \not\subset Gen(BTO)$. Dazu betrachten wir den folgenden Schedule, der zwar von BTO, aber nicht von 2PL erzeugt werden kann:

$r_1(z)r_2(z)w_2(x)r_3(x)r_1(y)w_2(y)c_1c_2c_3$

Dieser Schedule kann von einem BTO-Scheduler erzeugt werden, da die Zugriffe auf die Objekte mit der Reihenfolge der ersten Operationen der Transaktionen verträglich sind. Man kann aber zeigen, dass dieser Schedule nicht von einem 2PL-Scheduler erzeugt worden sein kann: Es gilt

$$wl_2(x) < w_2(x) < wu_2(x) < rl_3(x) < r_3(x)$$

und ausserdem

$$rl_1(y) < r_1(y) < ru_1(y) < wl_2(y) < w_2(y)$$

sowie

$$r_3(x) < r_1(y)$$

und damit transitiv auch $wu_2(x) < wl_2(y)$, was ein Widerspruch zur Zweiphasigkeit ist.

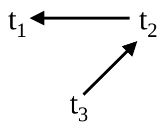
Aufgabe 5:

Reconsider the following condition for removing nodes (and edges) from the serialization graph in the SGT protocol: Remove t_i when it's finished and non 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 behaviour of the SGT protocol.

Wir betrachten den folgenden Präfix eines Schedules:

$w_1(z)r_2(x)w_1(x)c_1r_3(y)w_2(y)$

Der zugehörige Konfliktgraph sieht zu diesem Zeitpunkt folgendermaßen aus:



Wenn nun t_2 mit Commit beendet wird, würde man nach dem genannten Kriterium t_1 aus dem Graphen löschen. Damit riskiert man aber, dass der Schedule nicht konfliktserialisierbar werden kann. Nehmen wir an, dass t_3 als nächste Operation $r_3(z)$ ausführen will, so würde mit der Kante von t_1 nach t_3 ein Zyklus geschlossen, der aber nicht mehr erkannt werden kann, da t_1 bereits entfernt ist.

Aufgabe 6:

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.

See Figure 1 for the output schedule under BOCC, and Figure 2 for the output schedule under the FOCC protocol.

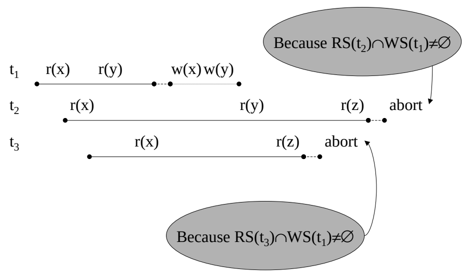


Abbildung 1: Resulting schedule under the BOCC protocol

Aufgabe 7:

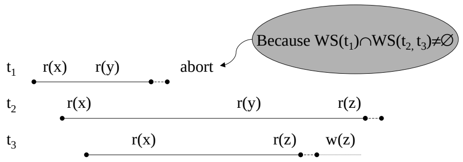


Abbildung 2: Resulting schedule under the FOCC protocol

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.

Aufgabe 8:

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

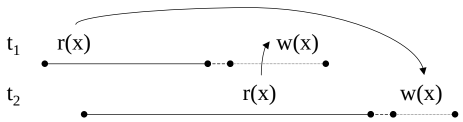


Abbildung 3: Incorrect FOCC execution without the critical section

See Figure 3 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 the write phase and the commit.

Aufgabe 9:

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 solve 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 an 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.