Tutorial 8 — Transactions and Recovery

Richard Wong
rk2wong@edu.uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

March 19, 2018

Exercise 8-1

Is the following transaction schedule *recoverable*?

Can we make a conflict-equivalent schedule that is *cascadeless*?

T1	r_x			r_y							c
T2			w_y				r_x			c	
T3						w_x			r_x	c	

Exercise 8-1 Solution (1/2)

In the context of recoverability, a transaction T **depends on** a transaction S if T reads a value that S has written to previously.

For a schedule to be recoverable, for any pair of transactions S and T, if T depends on S, then S must commit *before* T does.

In the given schedule, that means:

- T3 must commit before T2 does, and
- T2 must commit before T1 does.

Exercise 8-1 Solution (2/2)

For a schedule to be recoverable, for any pair of transactions S and T, if T depends on S, then S must commit *before* T reads a dependent value.

The following schedule is conflict-equivalent to the original, and is also cascadeless.

T1	r_x							r_y	c
T2		w_y				r_x	c		
T3			w_x	r_x	c				

Exercise 8-2

What is the weakest isolation guarantee available in SQL?

When would an developer want to use a weaker isolation level in their application?

Exercise 8-2 Solution

The weakest isolation level provided by SQL is **read-uncommitted**.

This level guarantees *no dirty writes*: no writes on top of uncommitted writes by other transactions.

Weaker isolation affords a greater degree of concurrency, for a potential boost in throughput. Use if the possibly-resulting inconsistencies are irrelevant to the application, or they are otherwise worth dealing with.

Exercise 8-3

Some DMBSes use snapshot isolation to implement *serializable*-level isolation. It works most of the time, but has failure cases.

How can snapshot isolation fail to create a serializable schedule, and what should happen when it creates a non-serializable schedule?

Exercise 8-3 Solution

In **snapshot isolation**, transactions operate on a snapshot of the DB that looks as it did when the transaction started.

A consistency check (for dirty writes, essentially) is performed right before an attempted commit.

Snapshot isolation can result in non-serializable schedules because no additional work is done to ensure serializability as the schedule is generated. The consistency check at the end will abort non-serializable schedules.

Exercise 8-4

Show that 2PL can create schedules that result in deadlock.

What can we do to **prevent** deadlock?

Exercise 8-4 Solution (1/2)

Recall that deadlock requires four conditions to be simultaneously true:

1. mutual exclusion,
2. hold-and-wait,
3. no pre-emption, and
4. circular wait.

A 2PL schedule that results in deadlock, supposing T1 wants to read resources *a* and *b*, and T2 wants to write to those same resources:

T1	s_a			s_b
T2		x_b	x_a	

Exercise 8-4 Solution (2/2)

Note that in 2PL, transactions can run into deadlock only during their growing phase.

We can prevent deadlock by having a smarter lock manager. There are several ways to prevent deadlock. The following are some things that the lock manager can do:

- Use a deadlock detection algorithm (e.g. Banker's, graph algorithms) to determine whether granting a lock might lead to deadlock, and make the caller wait or abort if so. This can be expensive.
- Enforce a partial ordering (could be encoded with a tree) for lock acquisition to remove the possibility of circular wait.

Exercise 8-5

Show that 2PL can create recoverable schedules with cascading rollbacks.

What variant of 2PL creates cascadeless schedules?

Exercise 8-5 Solution

T1	x_a	w_a	u_a									abort
T2				s_a	r_a	x_a	w_a	u_a				
T3									s_a	r_a	u_a	

Exercise 8-6

Supposing that instead handling deadlock with an avoidance/prevention strategy, we try to detect and recover. How do we recover from a deadlock?

Exercise 8-6 Solution

Recovering from a deadlock involves rolling back transactions until the circular wait is removed.

We choose a transaction to roll back using some heuristic that takes starvation into account, then roll back some or all of the transaction.