>>

Properties of Schedules

- Schedule Properties
- Serializable Schedules
- Transaction Failure
- Recoverability
- Cascading Aborts
- Strictness
- Classes of Schedules

^ >>

❖ Schedule Properties

If a concurrent schedule on a set of tx's *TT* ...

- produces the same effect as a serial schedule on *TT*
- then we say that the schedule is serializable

A goal of isolation mechanisms (see later) is

- arrange execution of individual operations in tx's in *TT*
- to ensure that a serializable schedule is produced

Serializability is one property of a schedule, focusing on isolation

Other properties of schedules focus on recovering from failures

<< ^ >>

❖ Serializable Schedules

Producing a serializable schedule

- eliminates all update anomalies ✓
- may reduce opportunity for concurrency ✗
- may reduce overall throughput of system ✗

If DB programmers know update anomalies are unlikely/tolerable

- serializable schedules may not be neccessary
- some DBMSs offer less strict isolation levels (e.g. repeatable read)
- allowing more opportunity for concurrency ✓

<< ^ >>

❖ Transaction Failure

So far, have implicitly assumed that all transactions commit.

Additional problems can arise when transactions abort.

Consider the following schedule where transaction T1 fails:

```
T1: R(X) W(X) A
T2:             R(X) W(X) C
```

Abort *will* rollback the changes to **X**, but ...

Consider three places where the rollback might occur:

```
T1: R(X) W(X) A [1]     [2]         [3]
T2:                R(X)     W(X) C
```

<< ^ >>

❖ Transaction Failure (cont)

Abort / rollback scenarios:

```
T1: R(X) W(X) A [1]      [2]          [3]
T2:                 R(X)     W(X) C
```

Case [1] is ok

- all effects of T1 vanish; final effect is simply from T2

Case [2] is problematic

- some of T1's effects persist, even though T1 aborted

Case [3] is also problematic

- T2's effects are lost, even though T2 committed

<< ^ >>

❖ Recoverability

Consider the serializable schedule:

```
T1:          R(X)  W(Y)  C
T2:  W(X)                    A
```

(where the final value of **Y** is dependent on the **X** value)

Notes:

- the final value of X is valid (change from T_2 rolled back)
- T_1 reads/uses an X value that is eventually rolled-back
- even though T_2 is correctly aborted, it has produced an effect

Produces an invalid database state, even though serializable.

<< ^ >>

❖ Recoverability (cont)

Recoverable schedules avoid these kinds of problems.

For a schedule to be recoverable, we require additional constraints

- all tx's T_i that write values used by T_j must commit before T_j commits

and this property must hold for all transactions T_j

Note that recoverability does not prevent "dirty reads".

In order to make schedules recoverable in the presence of dirty reads and aborts, may need to abort multiple transactions.

<< ^ >>

❖ Cascading Aborts

Recall the earlier non-recoverable schedule:

```
T1:          R(X)  W(Y)  C
T2:  W(X)                  A
```

To make it recoverable requires:

- delaying T_1's commit until T_2 commits
- if T_2 aborts, cannot allow T_1 to commit

```
T1:          R(X)  W(Y) ...   C? A!
T2:  W(X)                  A
```

Known as cascading aborts (or cascading rollback).

<< ^ >>

❖ Cascading Aborts (cont)

Example: T_3 aborts, causing T_2 to abort, causing T_1 to abort

```
T1:                          R(Y)  W(Z)          A
T2:          R(X)  W(Y)                    A
T3:  W(X)                                A
```

Even though T_1 has no direct connection with T_3 (i.e. no shared data).

This kind of problem ...

- can potentially affect very many concurrent transactions
- could have a significant impact on system throughput

<< ^ >>

❖ Cascading Aborts (cont)

Cascading aborts can be avoided if

- transactions can only read values written by committed transactions

  (alternative formulation: no tx can read data items written by an uncommitted tx)

Effectively: eliminate the possibility of reading dirty data ✓

Downside: reduces opportunity for concurrency ✗

GUW call these ACR (avoid cascading rollback) schedules.

All ACR schedules are also recoverable.

<< ^ >>

❖ Strictness

Strict schedules also eliminate the chance of *writing* dirty data.

A schedule is strict if

- no tx can read values written by another uncommitted tx (ACR)
- no tx can write a data item written by another uncommitted tx

Strict schedules simplify the task of rolling back after aborts.

<< ^ >>

❖ Strictness (cont)

Example: non-strict schedule

```
T1:  W(X)          A
T2:        W(X)      A
```

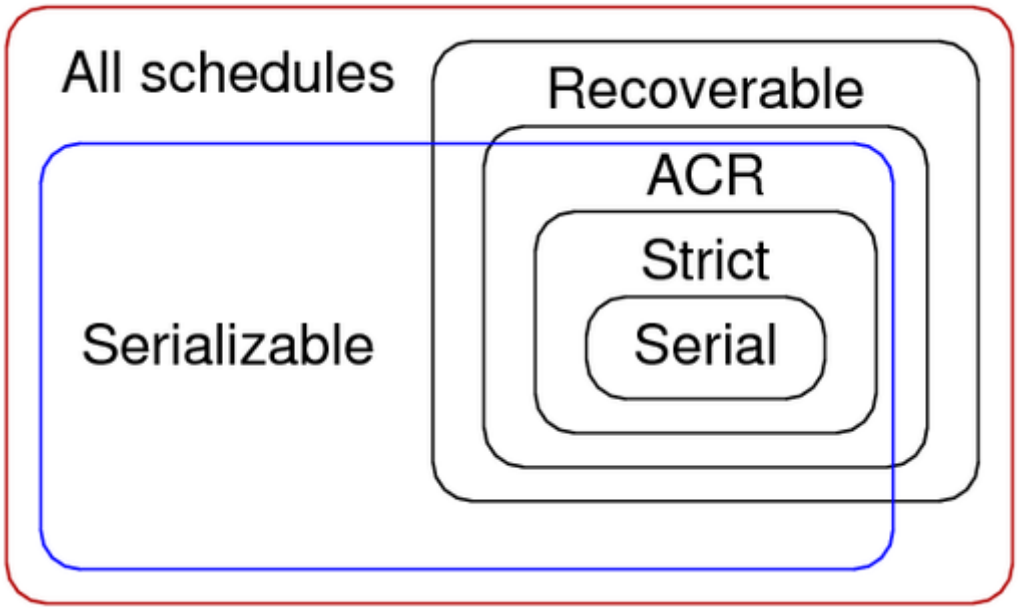
Problems with handling rollback after aborts:

- when T_1 aborts, don't rollback (need to retain value written by T_2)
- when T_2 aborts, need to rollback to pre-T_1 (not just pre-T_2)

<< ^

❖ Classes of Schedules

Relationship between various classes of schedules:



Schedules ought to be serializable and strict.

But more serializable/strict ⇒ less concurrency.

DBMSs allow users to trade off "safety" against performance.

COMP9315 21T1 ♢ Properties of Schedules ♢ [12/12]

Produced: 14 Apr 2021