Transactions and Schedules

Harvard Extension School

Cody Doucette, Ph.D.

Lecture designed by David G. Sullivan

Transactions: An Overview

- A *transaction* is a sequence of operations that is treated as a single logical operation. (abbreviation = txn)
- Example: a balance transfer

transaction T1

```
read balance1
write(balance1 - 500)
read balance2
write(balance2 + 500)
```

- Transactions are *all-or-nothing*: all of a transaction's changes take effect or none of them do.

Executing a Transaction

1. Issue a command indicating the start of the transaction.
2. Perform the operations in the transaction.
   - in SQL: `SELECT`, `UPDATE`, etc.
3. End the transaction in one of two ways:
   - *commit it:* make all of its results visible and persistent
     - *all* of the changes happen
   - *roll it back / abort it:* undo all of its changes, returning to the state before the transaction began
     - *none* of the changes happen

Why Do We Need Transactions?

- To prevent problems stemming from system failures.
  - example: a balance transfer

```
read balance1
write(balance1 - 500)
CRASH
read balance2
write(balance2 + 500)
```

Why Do We Need Transactions? (cont.)

- To ensure that operations performed by different users don't overlap in problematic ways.
  - example: this should *not* be allowed

user 1

```
read balance1
write(balance1 - 500)



read balance2
write(balance2 + 500)
```

user 2

```
read balance1
read balance2
if (balance1+balance2 < min)
    write(balance1 - fee)
```

ACID Properties

- A transaction has the following "ACID" properties:

  Atomicity: either all of its changes take effect or none do

  Consistency preservation: its operations take the database from one consistent state to another
  - consistent = satisfies the constraints from the schema, and any other expectations about the values in the database

  Isolation: it is not affected by and does not affect other concurrent transactions

  Durability: once it commits, its changes survive failures

- The user plays a role in consistency preservation.
  - ex: add to balance2 the same amnt subtracted from balance1
  - the DBMS helps by rejecting changes that violate constraints
  - guaranteeing the other properties also preserves consistency

Atomicity and Durability

- These properties are guaranteed by the part of the system that performs logging and recovery.
- After a crash, the recovery subsystem:
  - *redoes* as needed all changes by committed txns
  - *undoes* as needed all changes by uncommitted txns
    - restoring the old values of the changed data items
- We'll look more at logging and recovery later in the semester.

Isolation

- To guarantee isolation, the DBMS has to prevent problematic interleavings like the one we saw earlier:

transaction T1

```
read balance1
write(balance1 - 500)



read balance2
write(balance2 + 500)
```

transaction T2

```
read balance1
read balance2
if (balance1+balance2 < min)
    write(balance1 - fee)
```

- One possibility: enforce a *serial schedule* (no interleaving).

```
read balance1
write(balance1 - 500)
read balance2
write(balance2 + 500)
```

```
read balance1
read balance2
if (balance1+balance2 < min)
    write(balance1 - fee)
```

or

```
read balance1
read balance2
if (balance1+balance2 < min)
    write(balance1 - fee)
```

```
read balance1
write(balance1 - 500)
read balance2
write(balance2 + 500)
```

  - doesn't make sense for performance reasons. why?

Serializability

- A *serializable schedule* is one whose effects are equivalent to the effects of some serial schedule. For example:

schedule 1

transaction T1

```
read X
X = X + 5
write X

read Y
Y = Y + 6
write Y
```

transaction T2

```
read Y
Y = Y + 2
write Y

read X
X = X + 10
write X
```

- X is increased by 15
- Y is increased by 8

schedule 2 (a serial schedule)

transaction T1

```
read X
X = X + 5
write X
read Y
Y = Y + 6
write Y
```

transaction T2

```
read Y
Y = Y + 2
write Y
read X
X = X + 10
write X
```

- X is increased by 15
- Y is increased by 8

- Because the effects schedule 1 are equivalent to the effects of a serial schedule (schedule 2), schedule 1 is *serializable*.

Not All Schedules Are Serializable!

- Schedule 1 is a special case.

schedule 1

transaction T1

```
read X
X = X + 5
write X

read Y
Y = Y + 6
write Y
```

transaction T2

```
read Y
Y = Y + 2
write Y

read X
X = X + 10
write X
```

schedule 2 (a serial schedule)

transaction T1

```
read X
X = X + 5
write X
read Y
Y = Y + 6
write Y
```

transaction T2

```
read Y
Y = Y + 2
write Y
read X
X = X + 10
write X
```

- both T1 and T2 use addition to change the values of X and Y
- addition is commutative
- thus, the order in which T1 and T2 make their changes doesn't matter!

Not All Schedules Are Serializable! (cont.)

- If we change T2 so that it uses multiplication, the original interleaving is no longer serializable.

schedule 1B

transaction T1

```
read X
X = X + 5
write X

read Y
Y = Y + 6
write Y
```

transaction T2B

```
read Y
Y = Y * 2
write Y

read X
X = X * 10
write X
```

- X → **10(X + 5)**
- Y → **2Y + 6**

schedule 2B

transaction T1

```
read X
X = X + 5
write X
read Y
Y = Y + 6
write Y
```

transaction T2B

```
read Y
Y = Y * 2
write Y
read X
X = X * 10
write X
```

- X → 10(X + 5)
- Y → **2(Y + 6)**

schedule 3

transaction T2B

```
read Y
Y = Y * 2
write Y
read X
X = X * 10
write X
```

transaction T1

```
read X
X = X + 5
write X
read Y
Y = Y + 6
write Y
```

- X → **10X + 5**
- Y → 2Y + 6

- Because the effects schedule 1B are ***not*** equivalent to the effects of *any* serial schedule of T1 + T2B, schedule 1B is ***not*** serializable.

Conventions for Schedules

- We abstract all transactions into sequences of reads and writes.
  - example:

T2

```
read balance1
read balance2
if (balance1 + balance2 < min)
    write(balance1 – fee)
```

T2

```
read(A)
read(B)
write(A)
```

  - we use a different variable for each data item that is read or written
  - we ignore:
    - the actual meaning and values of the data items
    - the nature of the changes that are made to them
    - things like comparisons that a transaction does in its own address space

Conventions for Schedules (cont.)

- We can represent a schedule using a table.
  - one column for each transaction
  - operations are performed in the order given by reading from top to bottom

T_1	T_2
r(A)	
	r(B)
w(A)	
	r(A)
	w(A)

- We can also write a schedule on a single line using this notation:

  $r_i(A)$ = transaction T_i reads A
  $w_i(A)$ = transaction T_i writes A

  - example for the table above:

    $r_1(A)$; $r_2(B)$; $w_1(A)$; $r_2(A)$; **$w_2(A)$**

Serializability of Abstract Schedules

- How can we determine if an abstract schedule is serializable?
  - given that we don't know the exact nature of the changes made to the data
- We'll focus on the following:
  - which transaction is the last one to write each data item
    - that's the version that will be seen after the schedule
  - which version of a data item is read by each transaction
    - assume that if a transaction reads a different version, its subsequent behavior might be different

Conflicts in Schedules

- A *conflict* is a pair of actions that can't be swapped without potentially changing the behavior of one or more transactions.
- Examples in the schedule at right:
  - $w_1(A)$ and $r_2(A)$
    - swapping them leads T2 to read a different value of A
    - this may cause T2 to behave differently
  - $w_2(B)$ and $w_1(B)$
    - swapping them means later readers of B will see a different value of B
    - this may cause them to behave differently
- $r_1(B)$ and $r_2(B)$ do *not* conflict. why?

T_1	T_2
r(B)	
	r(B)
w(A)	
	r(A)
	w(A)
	w(B)
w(B)	
…	…

Which Actions Conflict?

- Actions in different transactions conflict if:
  1) they involve the same data item
  and 2) at least one of them is a write
- Pairs of actions that *do* conflict (assume i != j):
  - $w_i(A)$; $r_j(A)$ the value read by T_j may change if we swap them
  - $r_i(A)$; $w_j(A)$ the value read by T_i may change if we swap them
  - $w_i(A)$; $w_j(A)$ subsequent reads may change if we swap them
  - two actions from the same txn (their order is fixed by the client)
- Pairs of actions that *don't* conflict:
  - $r_i(A)$; $r_j(A)$ – two reads of the same item by different txns
  - $r_i(A)$; $r_j(B)$
  - $r_i(A)$; $w_j(B)$
  - $w_i(A)$; $r_j(B)$
  - $w_i(A)$; $w_j(B)$

  } operations on two *different* items by different txns

Conflict Serializability

- Rather than ensuring serializability, it's easier to ensure a stricter condition known as *conflict serializability*.
- A schedule is *conflict serializable* if we can turn it into a serial schedule by swapping pairs of *consecutive* actions that *don't* conflict.

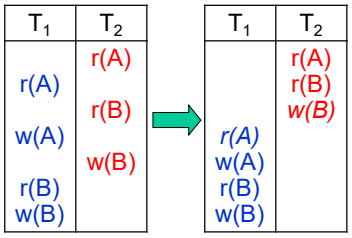
Example of a Conflict Serializable Schedule

$r_2(A); r_1(A); r_2(B); w_1(A); w_2(B); r_1(B); w_1(B)$

$r_2(A); r_2(B); r_1(A); w_1(A); w_2(B); r_1(B); w_1(B)$

$r_2(A); r_2(B); r_1(A); w_2(B); w_1(A); r_1(B); w_1(B)$

$r_2(A); r_2(B); w_2(B); r_1(A); w_1(A); r_1(B); w_1(B)$



- The final schedule is referred to as an *equivalent serial schedule.*
  - *serial* – all of T2, followed by all of T1
  - *equivalent* – it produces the same results as the original schedule

Testing for Conflict Serializability

- Because conflicting pairs of actions can't be swapped, they impose constraints on the order of the txns in an equivalent serial schedule.
  - example: if a schedule includes $w_1(A)$ … $r_2(A)$, T1 must come before T2 in any equivalent serial schedule
- To test for conflict serializability:
  - determine all such constraints
  - make sure they aren't contradictory
- Example: $r_2(A)$; $r_1(A)$; $r_2(B)$; $w_1(A)$; $w_2(B)$; $r_1(B)$; $w_1(B)$

  $r_2(A)$ … $w_1(A)$ means T2 must come before T1
  $r_2(B)$ … $w_1(B)$ means T2 must come before T1
  $w_2(B)$ … $r_1(B)$ means T2 must come before T1
  $w_2(B)$ … $w_1(B)$ means T2 must come before T1

  no contradictions, so this schedule is equivalent to the serial ordering T2;T1

  Thus, this schedule *is* conflict serializable.

Testing for Conflict Serializability (cont.)

- What about this schedule? $r_1(B)$; $w_1(B)$; $r_2(B)$; $r_2(A)$; $w_2(A)$; $r_1(A)$
- Which of the following pairs of actions from this schedule conflict? (choose all that apply)

A. $r_1(B)$; $r_2(B)$

B. $r_1(B)$; $w_2(A)$

C. $w_1(B)$; $r_2(B)$

D. $r_2(B)$; $r_2(A)$

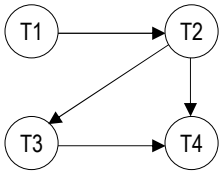
E. $w_2(A)$; $r_1(A)$

Using a Precedence Graph

- Tests for conflict serializability can use a *precedence graph*.
  - the vertices/nodes are the transactions
  - add an edge for each precedence constraint: T1 → T2 means T1 must come before T2 in an equivalent serial schedule

- Example: $r_2(A); r_3(A); r_1(B); w_4(A); w_2(B); r_3(B)$

  $r_2(A) \ldots w_4(A)$ means T2 → T4
  $r_3(A) \ldots w_4(A)$ means T3 → T4
  $r_1(B) \ldots w_2(B)$ means T1 → T2
  $w_2(B) \ldots r_3(B)$ means T2 → T3

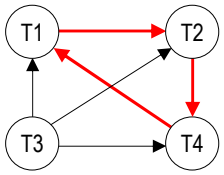


- After the graph is constructed, we test for *cycles* (i.e., paths of the form A → … → A).
  - if the graph is acyclic, the schedule is conflict serializable
    - use the constraints to determine an equivalent serial schedule (in this case: T1;T2;T3;T4)
  - if there's a cycle, the schedule is *not* conflict serializable

More Examples

- Determine if the following are conflict serializable:
  - $r_1(A); r_3(A); r_1(B); w_2(A); r_4(A); w_2(B); w_3(C); w_4(C); r_1(C)$

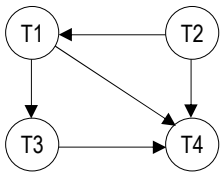
    $r_1(A) \ldots w_2(A)$ means T1 → T2
    $r_3(A) \ldots w_2(A)$ means T3 → T2
    $r_1(B) \ldots w_2(B)$ means T1 → T2
    $w_2(A) \ldots r_4(A)$ means T2 → T4
    $w_3(C) \ldots w_4(C)$ means T3 → T4
    $w_3(C) \ldots r_1(C)$ means T3 → T1
    $w_4(C) \ldots r_1(C)$ means T4 → T1



cycle: T1 → T2 → T4 → T1
not *conflict serializable*

  - $r_1(A); w_3(A); w_4(A); w_2(B); r_2(B); r_1(B); r_4(B)$

    $r_1(A) \ldots w_3(A)$ means T1 → T3
    $r_1(A) \ldots w_4(A)$ means T1 → T4
    $w_3(A) \ldots w_4(A)$ means T3 → T4
    $w_2(B) \ldots r_1(B)$ means T2 → T1
    $w_2(B) \ldots r_4(B)$ means T2 → T4



no cycles, so conflict serializable.
equivalent to T2; T1; T3; T4

Conflict Serializability vs. Serializability

- Conflict serializability is a *sufficient* condition for serializability, but it's not a *necessary* condition.
  - all conflict serializable schedules are serializable
  - not all serializable schedules are conflict serializable
- Consider the following schedule involving three txns:
- It is *not* conflict serializable, because:
  $r_2(A)$ … $w_1(A)$ means $T_2 \rightarrow T_1$
  $w_1(A)$ … $w_2(A)$ means $T_1 \rightarrow T_2$
- It *is* serializable because its effects are equivalent to either
  $T_1; T_2; T_3$ *or* $T_2; T_1; T_3$ why?

T_1	T_2	T_3
r(A)	r(A)	
w(A)	r(B)	
	w(A)	r(B)
		w(A)

Recoverability

- While serializability is important, it isn't enough for full isolation.
- Consider the serializable schedule at right.
  - includes "c" actions that indicate when the transactions commit
- Imagine that the system crashes:
  - *after* T1's commit
  - *before* T2's commit
- During recovery from the crash, the system:
  - *keeps* all of T1's changes, because it committed before the crash
  - *undoes* all of T2's changes, because it didn't commit before the crash

T_1	T_2
	r(A)
	w(B)
r(B)	
w(A)	
c	
CRASH	
	c

Recoverability (cont.)

- This is problematic!
  - T1 reads T2's write of B
  - it then performs actions that may be based on the new value of B
  - during recovery from the crash, T2 is rolled back
    ➔ B's old value is restored
  - it's possible T1 would have behaved differently if it had read B's old value
  - it's too late to roll back T1, because it has already committed!
- We say that this schedule is *unrecoverable.*
  - if a crash occurs between the two commits, the process of recovering from the crash could lead to problematic results

T_1	T_2
	r(A)
	w(B)
r(B)	
w(A)	
c	
CRASH	
	c

Recoverability (cont.)

- In a *recoverable* schedule, if T1 reads a value written by T2, T1 must commit *after* T2 commits.
- This allows us to safely recover from a crash at any point:

unrecoverable

T_1	T_2
	r(A)
	w(B)
r(B)	
w(A)	
c	
	c

recoverable

T_1	T_2
	r(A)
	w(B)
r(B)	
w(A)	
c	
	c

T_1	T_2
	r(A)
	w(B)
r(B)	
w(A)	
	c
c	
CRASH	

the reader of the changed value survives the crash, but so does the writer

T_1	T_2
	r(A)
	w(B)
r(B)	
w(A)	
CRASH	
	c
c	

the writer of the changed value is rolled back, but so is the reader

T_1	T_2
	r(A)
	w(B)
r(B)	
w(A)	
	c
CRASH	
c	

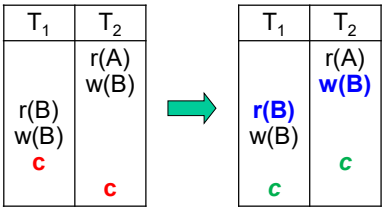
the reader is rolled back and the writer isn't, but that's okay since the writer didn't base its actions on what the reader did

Dirty Reads and Cascading Rollbacks

- *Dirty data* is data written by an uncommitted txn.
  - it remains dirty until the txn is either:
    - committed: in which case the data is no longer dirty and it is safe for other txns to read it
    - rolled back (either voluntarily or by the DBMS): in which case the write of the dirty data is undone
- A *dirty read* is a read of dirty data.
- Dirty reads can lead to *cascading rollbacks*.
  - if the writer of the dirty data is rolled back, the reader must be, too

Dirty Reads and Cascading Rollbacks (cont.)

- We made our earlier schedule recoverable by switching the order of the commits:



- Could the revised schedule lead to a cascading rollback?
- To get a *casecadeless* schedule, don't allow dirty reads.

Goals for Schedules

- We want to ensure that schedules of concurrent txns are:
  - *serializable*: equivalent to some serial schedule
  - *recoverable*: ordered so that the system can safely recover from a crash or undo an aborted transaction
  - *cascadeless*: ensure that rolling back one transaction does not produce a series of *cascading rollbacks*
- To achieve these goals, we use some type of *concurrency control mechanism*.
  - *controls* the actions of *concurrent* transactions
  - prevents problematic interleavings

Extra Practice

- Is the schedule at right:
  - conflict serializable?
  - serializable?
  - recoverable?
  - cascadeless?

T_1	T_2
	r(B)
r(B)	
	w(B)
w(A)	
	r(A)
c	
	c

Extra Practice

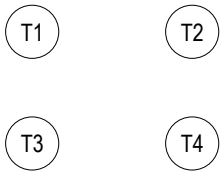
- What scenarios involving the schedule at right could produce cascading rollbacks?

T_1	T_2	T_3
	r(C)	w(C)
r(B)	w(B)	
w(A)		
		r(A)
…	…	…

Is This Schedule Conflict Serializable?

- Draw the precedence graph to find out!

$w_1(A); r_2(B); r_2(A); r_4(A); w_2(B); r_4(B); w_4(C); r_3(D); w_3(C)$



A. Yes. It is equivalent to the serial schedule T1;T2;T3;T4

B. Yes. It is equivalent to the serial schedule T1;T2;T4;T3

C. No. The graph includes the cycle T1 → T4 → T2 → T1

D. No. The graph includes the cycle T1 → T2 → T4 → T1

What If We Add This Write?

- Draw the precedence graph to find out!

$w_1(A); r_2(B); r_2(A); r_4(A); w_2(B); r_4(B); w_4(C); r_3(D); w_3(C);$ **$w_1(D)$**

$w_1(A) \ldots r_3(A)$ means T1 → T3
$w_1(A) \ldots r_2(A)$ means T1 → T2
$w_1(A) \ldots r_4(A)$ means T1 → T4
$w_2(B) \ldots r_4(B)$ means T2 → T4
$w_4(C) \ldots w_3(C)$ means T4 → T3