Advanced Database Management Systems

Lecture 20
Transactions - Chapter 17

Transactions

- ***Transaction***:
  an indivisible unit of data processing

- All transactions must have the ***ACID*** properties:

  - **Atomicity**: all or nothing
  - **Consistency**: no constraint violations
  - **Isolation**: no interference from other concurrent transactions
  - **Durability**: committed changes must not be lost
    due to any kind of failure

Atomic Transactions

- Fred wants to move $200 from his savings account to his checking account.

1) Money must be subtracted from savings account.
2) Money must be added to checking count.

If both happen, Fred and the bank are both happy.
If neither happens, Fred and the bank are both happy.

If only one happens, either Fred or the bank will be unhappy.

Fred's transfer must be *all or nothing*.

Transactions are Atomic

- Transactions must be ***atomic*** (indivisible)
  - the DBMS must ***ensure*** atomicity
  - everything happens, or nothing happens
  - boundaries of transaction (in time) are generally set by the application …

    the DBMS has no means of determining the intention of a transaction

Correct Transactions

- Wilma tries to withdraw $1000 from account 387.

Accounts

No	PIN	Balance
101	8965	10965.78
387	6643	**652.55**
543	4287	8720.12

constraint:
account.Balance must be non-negative

any transaction withdrawing
more than $652.55 from acct 387
will violate this constraint

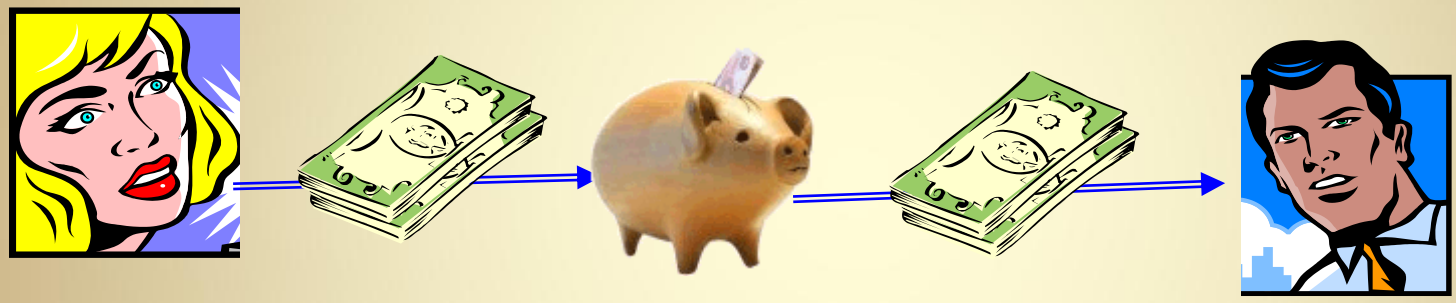
Wilma's transaction cannot be accepted.

Transactions are Consistent

- A transaction must leave the database in an valid or consistent state
  - valid state == no constraint violations
- A *constraint* is a declared rule defining specifying database states
- Constraints may be violated temporarily … but must be corrected before the transaction completes

Concurrent Transactions

- Fred is withdrawing $500 from account 543.
- Wilma's employer is depositing $1983.23 to account 543.
- These transactions are happening *at the same time*.



Accounts

No	PIN	Balance
101	8965	10965.78
387	6643	652.55
543	4287	8720.12

Accounts

No	PIN	Balance
101	8965	10965.78
387	6643	652.55
543	4287	10233.35

Combined result of both transactions must be correct

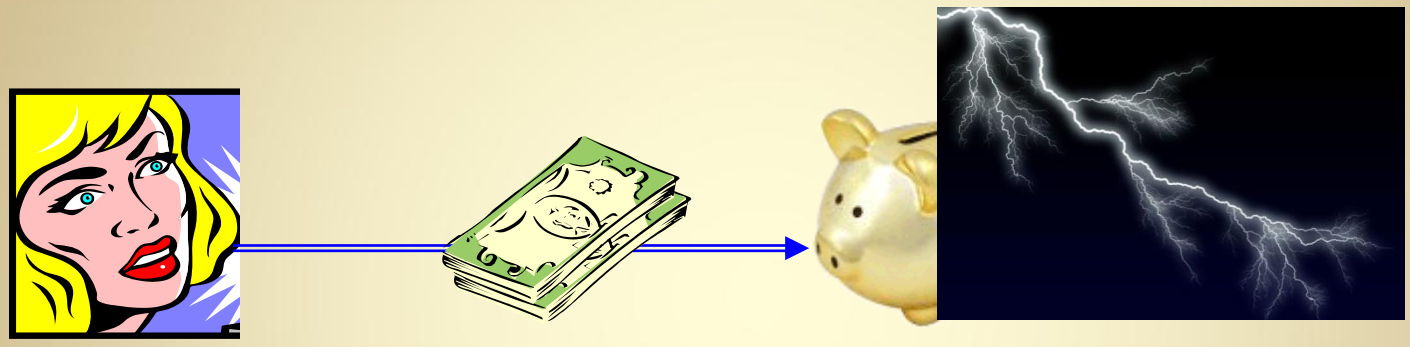
Transactions are Isolated

- If two transactions occur at the same time, the cumulative effect must be the same as if they had been done in *isolation*
  - ($8720.12 - $500) + $1983.23 = $10233.35
  - ($8720.12 + $1983.23) - $500 = $10233.35
  - $$\$8720.12 + \left[\begin{array}{l} -\$500.00 \\ +\$1983.23 \end{array}\right] = \$10233.35$$

    happen concurrently
- Ensuring isolation is the task of *concurrency control*

Durable Transactions

- Wilma deposits $50,000 to account 387.
- Later, the bank's computer crashes due to a lightning storm.



Accounts

No	PIN	Balance
101	8965	10965.78
387	6643	652.55
543	4287	8720.12

Accounts

No	PIN	Balance
101	8965	10965.78
387	6643	50652.55
543	4287	8720.12

Wilma's deposit cannot be lost.

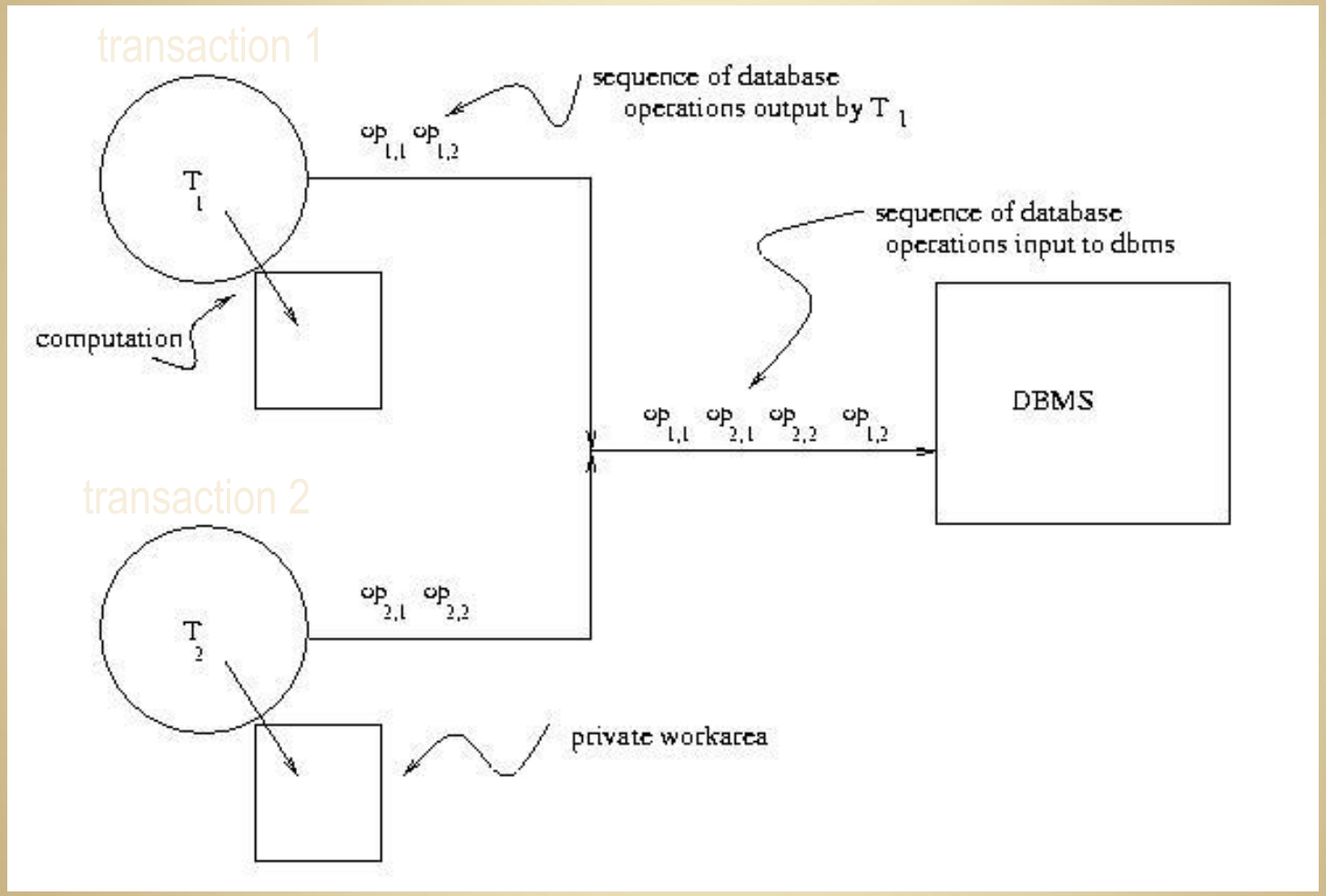
Transactions are Durable

- Once a transaction's effect on the database state has been *committed*, it must be *permanent*
- The DBMS must ensure *persistence*, even in the event of system failures
- Sources of failure:
  - computer or operating system crash
  - disk failure
  - fire, theft, power outage, earthquake, operator errors, …

Transaction Analysis

- A transaction is composed of a time-ordered sequence of operations on specific data items
- The operations are specific database accesses
  - insertion, deletion, modification or retrieval
  - we actually only need to consider two kinds of operations: writes and reads
- The data item are portions of the database state:
  - the database state is viewed as a collection of named data items
  - the size of the data items (granularity) that are used could be: an attribute, a record, a relation, a page/block

Interleaved Execution



from *Database Applications: an Application-Oriented Approach*
by Kifer, Bernstein and Lewis

Transaction Analysis

- To ensure transaction properties, it is sufficient to define transactions as sequences of operations of two types:
  - **read_item(X)**: access the current value of item X
  - **write_item(X)**: modify the value of item X

- Example Transaction:

read_item(acct_387_balance)
acct_387_balance := acct_387_balance - 300
write_item(acct_387_balance)

Concurrency Problem

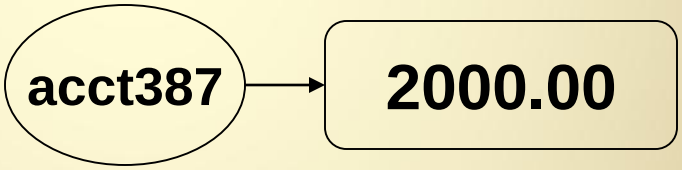
Fred's Transaction:

read_item(acct387)
acct387 := acct387 - 300
write_item(acct387)

Wilma's Transaction:

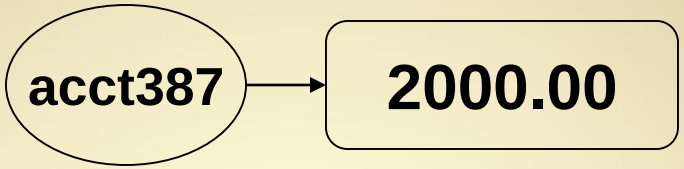
read_item(acct387)
acct387 := acct387 + 900
write_item(acct387)

(portion of) current database state



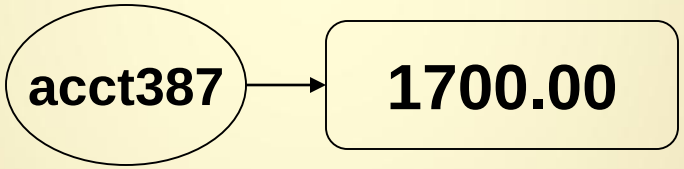
Is it possible that the execution of these two transactions, on the given database state, can cause an incorrect result?

Possible Execution #1



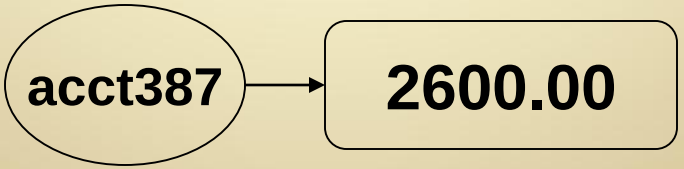
Fred's Transaction

f1: read_item(acct387)
f2: acct387 := acct387 - 300
f3: write_item(acct387)



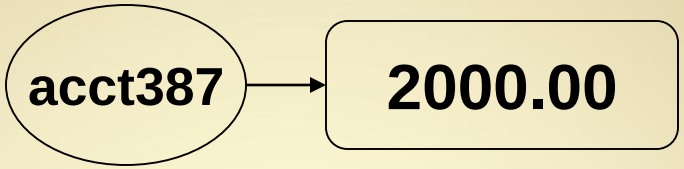
Wilma's Transaction

w1: read_item(acct387)
w2: acct387 := acct387 + 900
w3: write_item(acct387)



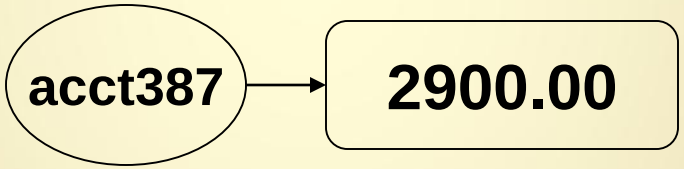
correct result

Possible Execution #2



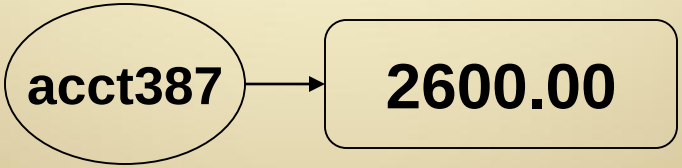
Wilma's Transaction

w1: read_item(acct387)
w2: acct387 := acct387 + 900
w3: write_item(acct387)



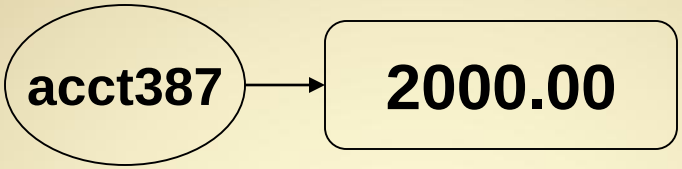
Fred's Transaction

f1: read_item(acct387)
f2: acct387 := acct387 - 300
f3: write_item(acct387)



correct result

Possible Execution #3



w1: read_item(acct387)

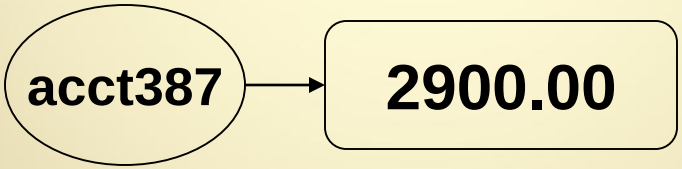
f1: read_item(acct387)

f2: acct387 := acct387 - 300

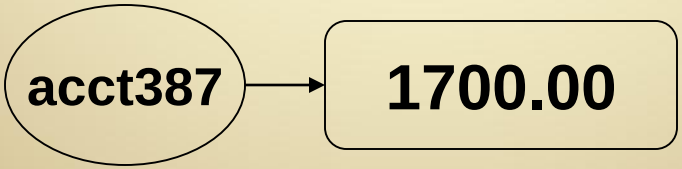
w2: acct387 := acct387 + 900

w3: write_item(acct387)

the two transactions overlap



f3: write_item(acct387)



incorrect result!

Theory and Implementation

- Transaction Theory (Chap 17)
  - interprets transactions as schedules
  - defines classes of schedules that obey ACID
- Concurrency Control (Chap 18)
  - implementation techniques that ensure *isolation* of concurrent transactions
  - transaction theory used to prove that implementation will work
- Recovery (Chap 19)
  - techniques for ensuring *durability*

Transaction Theory

Serializability and Equivalence of Transactions

Transaction Termination

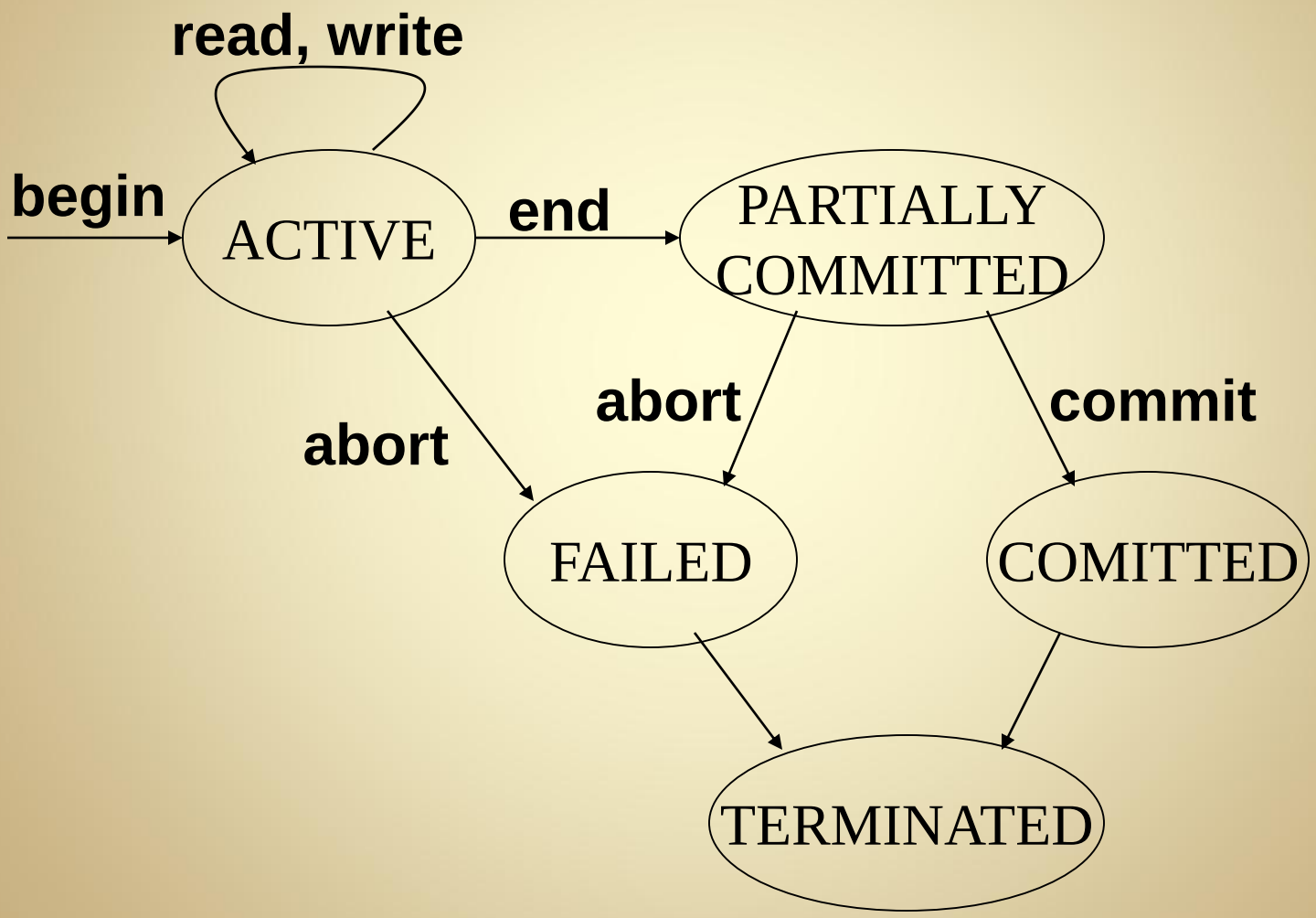
- Transactions are atomic:
  there are only two possible ways
  they can terminate:
  - **commit:** all operations are permanently applied to the database state
  - **abort:** no operations are applied to the database state

SQL calls an *abort* a *rollback*

Transaction Theory

- A transaction is modeled as a time-ordered sequence of operations on specific data items
- Operations:
  - **begin**: transaction start point
  - **read(X)**: data item X was read from the database
  - **write(X)**: data item X was written to the database
  - **end**: (logical) transaction end point,
    reads and writes have completed
  - **commit**: point at which all operations become permanent
  - **abort**: point at which all operations are removed

Transaction States



Transaction Data Items

- Read and write operations also record the *data item* that was involved
  - for simplicity, the same name is used for a data item in the database, and for a copy of the data item in some application's memory
- granularity = the size of the data items
  - could be: an attribute, a record, a relation, a page/block
  - granularity doesn't affect correctness of theory
  - in practice, larger granularity leads to less concurrency, since more conflict is (incorrectly) detected more often

Complete Schedules

- Schedule: A particular ordering of the operations from a set of concurrent transactions
- Only *complete schedules* are considered, others are not valid (atomicity)
- Complete schedule:
  - Includes all operations from every transaction
  - commit or abort is the last operation in each transaction
  - operations from the same transaction appear in the same order in the schedule

Example Transactions

- Three concurrent transactions:

T1	T2	T3
begin	begin	begin
read(X)	read(X)	read(Y)
X = X-2	X = X+3	Y = Y+1
write(X)	write(X)	write(Y)
read(Y)	end	end
Y = Y+2	commit	commit
write(Y)		
end		
commit		

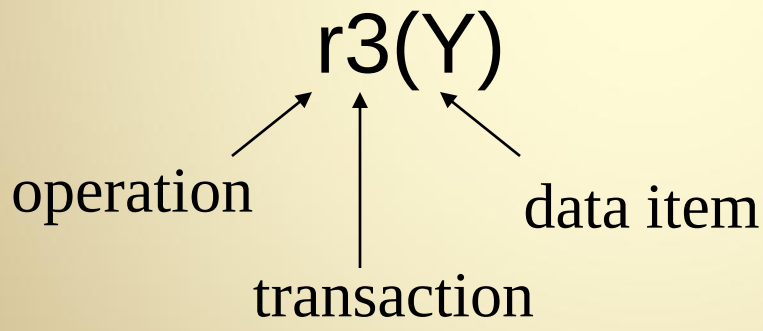
begin/end enclose all reads and writes

all end with a commit or an abort

Schedule Notation

- a more compact notation for schedules:

b3, r3(Y), w3(Y), e3, c3



T3
begin
read(Y)
Y = Y+1
write(Y)
end
commit

note: we ignore the computations on the local copies of the data when considering schedules (they're not interesting)

Serial Schedules

- A ***serial schedule*** is one in which the transactions do not overlap (in time)

b1,r1(X),w1(X),r1(Y),w1(Y),e1,c1,
b2,r2(X),w2(X),e2,c2,
b3,r3(Y),w3(Y),e3,c3

b2,r2(X),w2(X),e2,c2,
b1,r1(X),w1(X),r1(Y),w1(Y),e1,c1,
b3,r3(Y),w3(Y),e3,c3

b2,r2(X),w2(X),e2,c2,
b3,r3(Y),w3(Y),e3,c3,
b1,r1(X),w1(X),r1(Y),w1(Y),e1,c1

These are all serial schedules for the three example transactions

There are six possible serial schedules for three transactions

n! possible serial schedules for n transactions

Serial Schedules are Correct

- serial schedules are *correct schedules*
  - correct = transactions produce correct database states
- since each transaction is *consistency preserving*, they each must produce correct DB states when executed in *isolation*
- in a serial schedule, transactions do not overlap, therefore *isolation* must hold
- thus, a serial schedule is a correct schedule

Serializability

- If a schedule can be proven to be *equivalent* to ***some*** serial schedule, then that schedule must be correct
  - *serializable* = equivalent to some serial schedule
- There are different concepts of equivalence, each leading to different concepts of serializability
- We'll only consider the most conservative definition of equivalence: *conflict equivalence*

conservative = detects more potential conflicts, or identifies more schedules as incorrect. Rejecting correct schedules is not desirable, but accepting incorrect schedules in intolerable (for a DBMS).

Transaction Theory and Concurrency Control

- *Transaction theory* determines whether some particular schedule is correct
  - uses serializability and equivalence of schedules
  - in theory, we can manipulate the order of operations to find equivalent schedules.
    This is only a *test* of the correctness of some schedule.
- *Concurrency control* uses transaction theory to enforce *policies*, such as only allowing correct schedules to execute.
  - In practice, the flexibility to reorder operations is limited. Generally, we can only *delay* operations in time.

Transaction Theory and Concurrency Control

- Concurrency control can test a set of active or partially committed transactions to determine if the actual schedule is correct (or potentially correct).
  - CC cannot go back in time and reorder operations, other means for dealing with incorrect schedules are required.
- Basic concurrency control policies:
  - *pessimistic*: Prevent incorrect schedules from occurring by delaying conflicting operations as they happen using techniques such as *locking*.
  - *optimistic*: Allow any schedule to occur, then abort or roll-back transactions that lead to incorrect schedules.

Conflict Equivalence

- Two schedules are conflict equivalent if the order of any two conflicting operations is the same in both schedules.
- Two operations conflict if
  - they belong to different transactions
  - they access the same data item (read or write)
  - at least one is a write

T1: b1,r1(X),w1(X),r1(Y),w1(Y),e1,c1,

T2: b2,r2(X),w2(X),e2,c2

conflicting operations:
r1(X),w2(X)
w1(X), r2(X)
w1(X), w2(X)

Conflict Equivalence

- The term "conflicting operations" can be misleading
  - The operations do not conflict in any *particular* schedule, rather they will cause two schedules to be non-equivalent if their order is different in the two schedules
  - A better term might be "conflict causing operations"
- Two operations from the same transaction cannot conflict, since their relative order must be the same in all complete schedules.

Conflict Equivalence Example

schedule 1:

b1,r1(X),w1(X),r1(Y),w1(Y),e1,c1,b2,r2(X),w2(X),e2,c2

r1(X) < w2(X), w1(X) < r2(X), w1(X) < w2(X)

schedule 2:

b2,r2(X),w2(X),b1,r1(X),w1(X),r1(Y),w1(Y),e1,c1,e2,c2

w2(X) < r1(X), r2(X) < w1(X), w2(X) < w1(X)

schedule 3:

b1,r1(X),w1(X),b2,r2(X),w2(X),e2,c2,r1(Y),w1(Y),e1,c1,

r1(X) < w2(X), w1(X) < r2(X), w1(X) < w2(X)

schedule 1 and schedule 3 are conflict equivalent

schedule 2 is not conflict equivalent to either schedule 1 or 3

Conflict Serializability

- A schedule is *conflict serializable* if it is conflict equivalent to some serial schedule
  - a conflict serializable schedule is a correct schedule
- Previous example:
  - schedule 1 is a serial schedule
    schedule 1 is correct
  - schedule 3 is conflict equivalent to schedule 1,
    therefore it is serializable
    schedule 3 is correct
  - schedule 2 is not conflict equivalent to schedule 1,
    therefore it is not serializable
    schedule 2 may be incorrect

Testing for Conflict Serializability

- We could test a schedule against every possible serial schedule for the same transactions
  - this is intractable for large numbers of transactions
- Precedence graphs are a more efficient test
  - graph indicates a partial order on the transactions required by the order of the conflicting operations
  - the partial order must hold in any conflict equivalent serial schedule
  - if there is a loop in the graph, the partial order is not possible in any serial schedule
  - if the graph has no loops, the schedule is conflict serializable

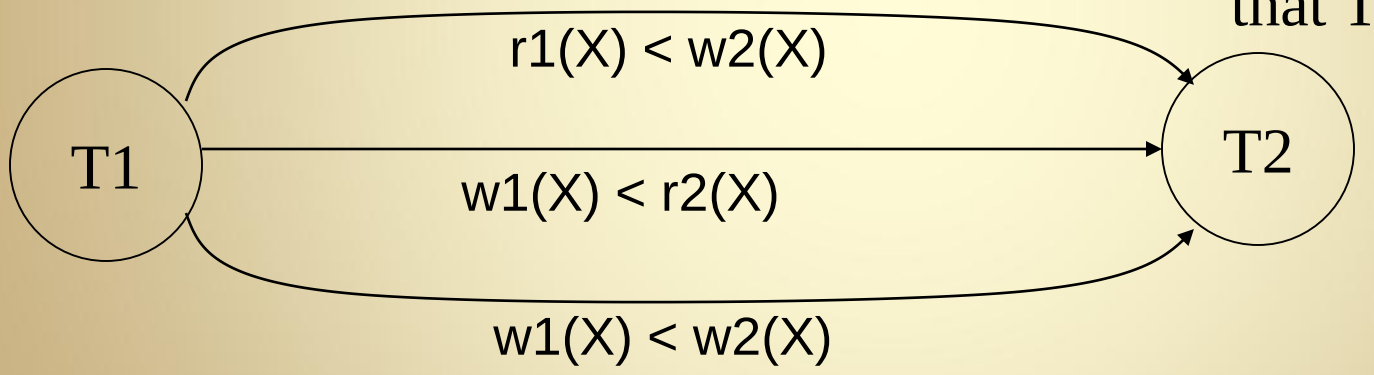
Precedence Graph Examples

schedule 3:

b1,r1(X),w1(X),b2,r2(X),w2(X),e2,c2,r1(Y),w1(Y),e1,c1,

r1(X) < w2(X), w1(X) < r2(X), w1(X) < w2(X)

arrows indicate
that T1 precedes T2



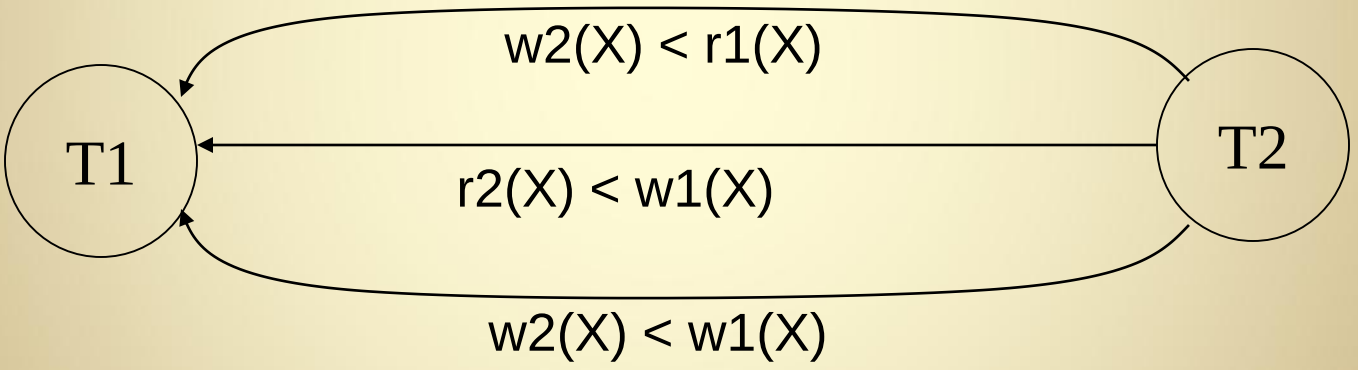
schedule 3 is conflict serializable
it is conflict equivalent to some serial schedule
in which T1 precedes T2

Precedence Graph Examples

schedule 2:

b2,r2(X),w2(X),b1,r1(X),w1(X),r1(Y),w1(Y),e1,c1,e2,c2

w2(X) < r1(X), r2(X) < w1(X), w2(X) < w1(X)



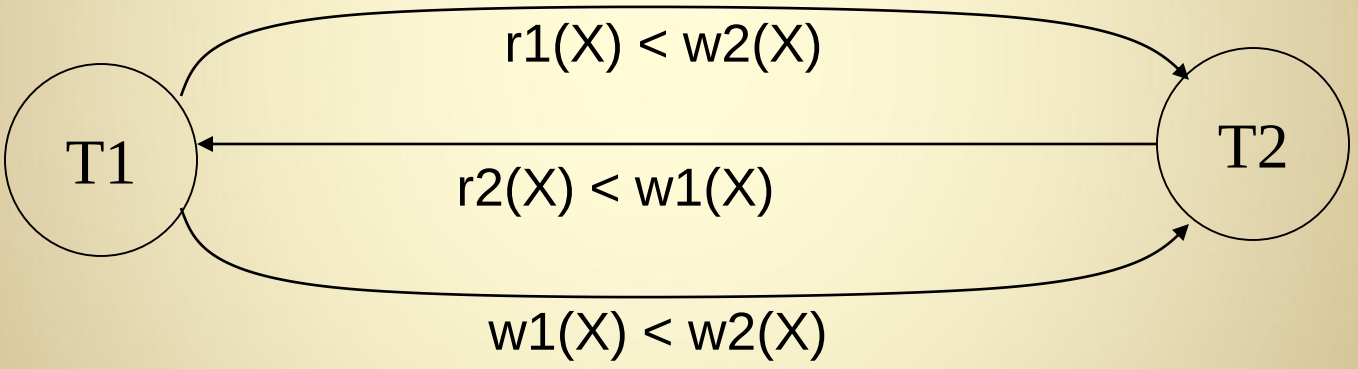
schedule 2 is conflict serializable
it is conflict equivalent to some serial schedule
in which T2 precedes T1

Precedence Graph Examples

schedule 4:

b2,r2(X),b1,r1(X),w1(X),r1(Y),w1(Y),w2(X),e1,c1,e2,c2

r1(X) < w2(X), r2(X) < w1(X), w1(X) < w2(X)



schedule 4 is not conflict serializable
there is no serial schedule
in which T2 precedes T1 and T1 precedes T2