Intro to DB

CHAPTER 14
TRANSACTIONS

Chapter 14: Transactions

- Transaction Concept
- Transaction Model
- Storage Structure
- Transaction States
- Transaction Isolation & Schedules
- Serializability
- Recoverability
- Isolation Levels
- Transactions as SQL Statements

Transactions

- What is a transaction?
  - (user view)
  - A program unit that accesses and possibly updates various data items (system view)
  - DBMS must guarantee certain properties (ACID properties) for units of works declared as a DB transaction

Examples of Transactions

- Banks
  - Withdraw $100 to account A.
  - Transfer $50 from account A to B.
- Schools
  - Register course #409.433 for student #4321.
- Airlines
  - Check if two seats are available on flight #453.
  - Reserve the two seats on flight #453.
- Companies
  - Increase every employee’s salary by 5%.

Dangers for Transactions

- Various types of failure
  - system crash
  - disk failure
  - system error
-
  - disk access is performed in chunks: page (block)
  - i.e., write operation performed after the right amount of data has been gathered
  - buffer manager may pin a page
- Concurrent execution of multiple transactions

Properties of a Transaction

ACID properties

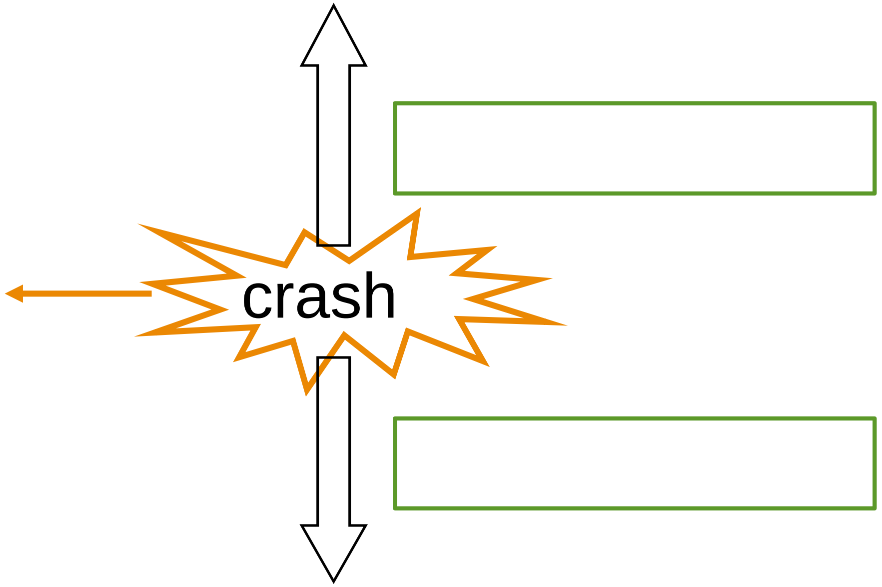
- **A**tomicity
- **C**onsistency (Correctness)
- **I**solation
- **D**urability

Atomicity

- **All or nothing**

  "Transfer $50 from account A to account B"

```
Begin transaction
      read(A,a)
      a = a-50
      write(A,a)
      read(B,b)
      b = b+50
      write(B,b)
End Transaction
```



Consistency

- Move from a **consistent state to another consistent state**

"Withdraw $100 from account A"

```
Begin transaction
      read(A,a)
      a = a-100
      write(A,a)
End Transaction
```

What if A only had $20?

-

Isolation

- Should **not be interfered** by other transactions (concurrency)

“Transaction T1”

```
Begin transaction
   read(A,a1)
   a1 = a1-50
   write(A,a1)
   read(B,b1)
   b1 = b1+50
   write(B,b1)
End Transaction
```

“Transaction T2”

```
Begin transaction
   read(A,a2)
   a2 = a2-100
   write(A,a2)
End Transaction
```

-

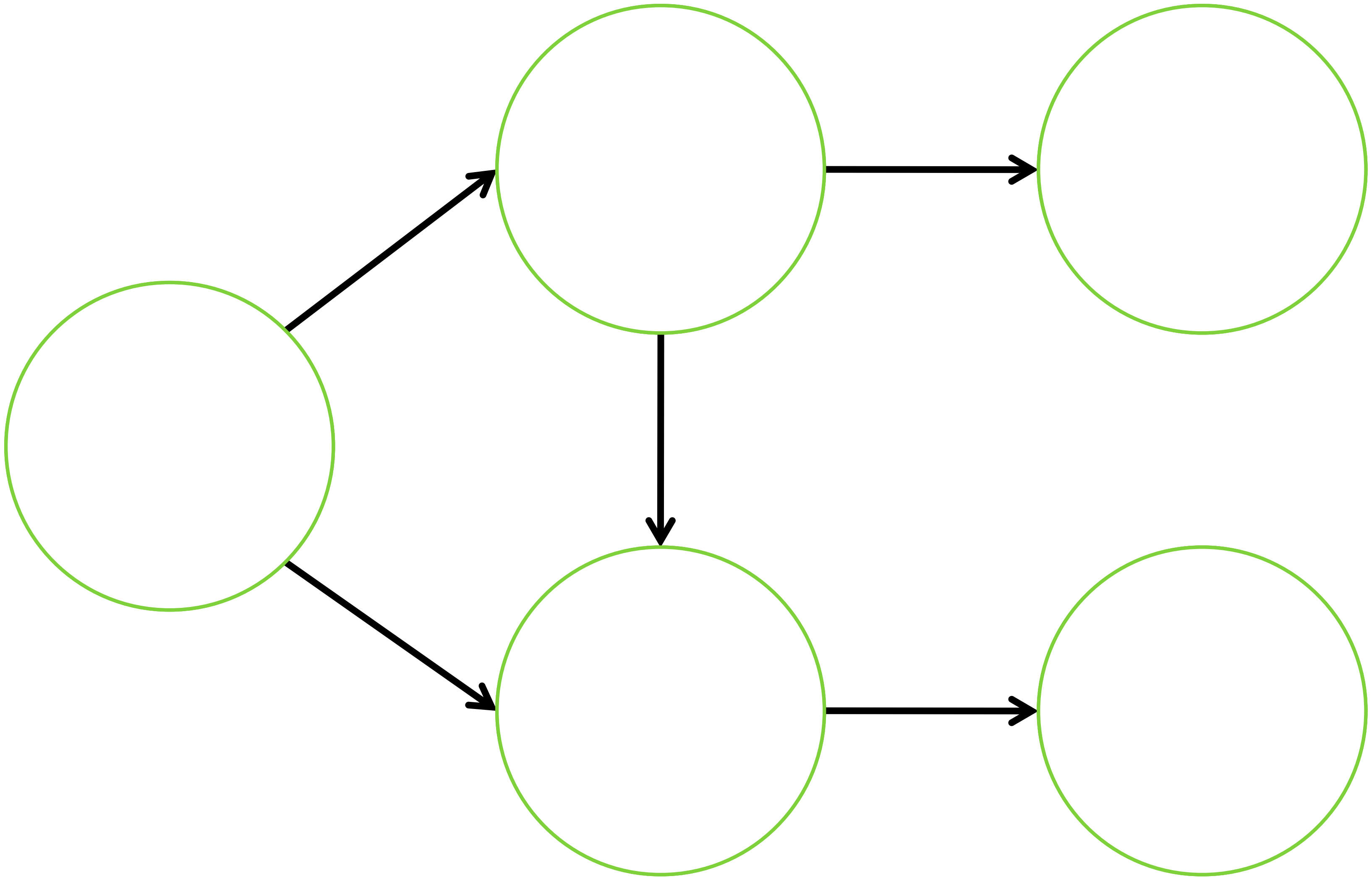
Durability

- The effect of a completed transaction should be **durable & public**

“Withdraw $100 from account A”

```
Begin transaction
      read (A,a)
      a = a-100
      write(A,a)
End Transaction
 ...
*buffer flush*
```

Transaction States



Transaction States (Cont.)

- ***Active***: the initial state, transaction stays in this state while it is executing
- ***Partially committed***: the final statement has been executed
- ***Failed***: normal execution can no longer proceed
- ***Aborted***: transaction has been rolled back and the database restored to its state prior to the start of the transaction.
  - Two options after it has been aborted:
    - restart the transaction – only if no internal logical error
    - kill the transaction
- ***Committed***: after *successful completion*.

Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system
  - increased processor and disk utilization
  - reduced average response time
- – mechanisms to achieve isolation
  - to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

Schedules

- Simplified view of transactions
  - Assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.
  - Ignore operations other than **read** and **write** instructions
    => simplified schedules consist of only **read** and **write** instructions.

- *Schedules*
  -
  - schedule for a set of transactions:
    - must consist of all instructions of those transactions
    - must preserve the order in which the instructions appear in each individual transaction

Example Schedules

- T_1: transfer \$50 from A to B; T_2: transfer 10% of the balance from A to B.
-

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit

Schedule 1 (fig 14.2)

T_1	T_2
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	

Schedule 2 (fig 14.3)

Example Schedule (Cont.)

- Let T_1 and T_2 be the transactions defined previously.
- Schedule 3 is **not** a serial schedule, but it

- In both Schedule 1 and 3, the sum *A+B is preserved.*

T_1	T_2
read (A) $A := A - 50$ write (A)	
	read (A) $temp := A * 0.1$ $A := A - temp$ write (A)
read (B) $B := B + 50$ write (B) commit	
	read (B) $B := B + temp$ write (B) commit

Schedule 3 (fig 14.4)

Example Schedules (Cont.)

- Schedule 4 **does not** preserve the value of the the sum $A + B$.

  Schedule 4 is **not** equivalent to Schedule 1.

T_1	T_2
read (A)	
$A := A - 50$	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	$B := B + temp$
	write (B)
	commit

Schedule 4 (fig 14.5)

Serializability

- Basic Assumption – Each transaction preserves database consistency.
  - Thus, **serial execution** of a set of transactions preserves database consistency.

- A (possibly concurrent) schedule is serializable if it is **equivalent** to a serial schedule.

- *S*1 is *equivalent* to *S*2 if, for every possible instance of the database,

Serializable Schedules

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

Conflict Serializability

- Instructions l_i and l_j of transactions T_i and T_j respectively, conflict if and only if
  - there exists some item Q accessed by both l_i and l_j, and
  - at least one of these instructions is a write(Q)

  1. l_i = read(Q), l_j = read(Q). not conflict.
  2. l_i = read(Q), l_j = write(Q). conflict.
  3. l_i = write(Q), l_j = read(Q). conflict
  4. l_i = write(Q), l_j = write(Q). conflict
- Intuitively, a conflict between l_i and l_j forces a (logical) temporal order between them.

Conflict Serializability (Cont.)

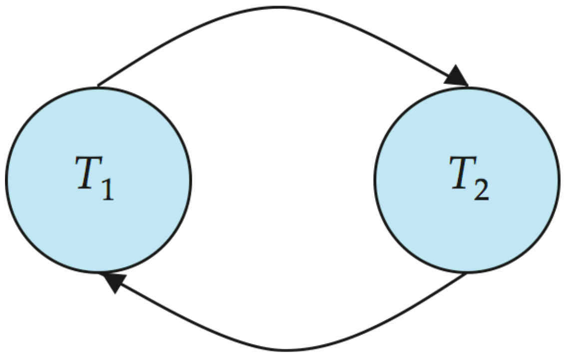
- Schedules *S* and *S´* are *conflict equivalent* if
  - *S* can be transformed into a schedule *S´*
  - by a series of swaps of non-conflicting instructions
- A schedule *S* is *conflict serializable* if it is conflict equivalent to a serial schedule

- Example of a schedule that is **not** conflict serializable:

T_3	T_4
read (Q)	
	write (Q)
write (Q)	

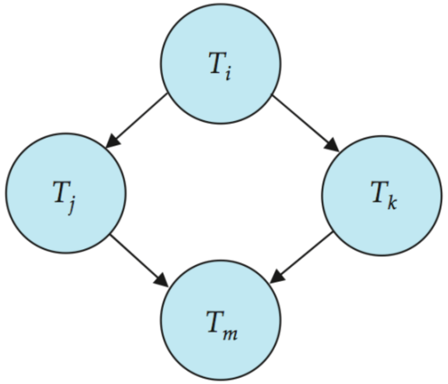
Testing for Serializability

- Consider some schedule of a set of transactions $T_1, T_2, ..., T_n$
- **Precedence graph** — a direct graph where the vertices are the transactions
  - draw an arc from T_i to T_j if the two transaction conflict, and T_i accessed the data item on which the conflict arose earlier.
  - We may label the arc by the item that was accessed.
- Example

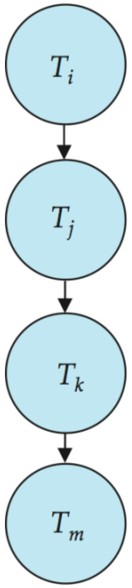


Test for Conflict Serializability

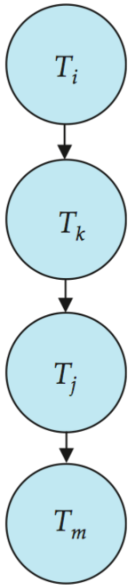
- A schedule is conflict serializable if and only if its precedence graph is acyclic.
- Cycle-detection algorithms exist which take order n^2 time, where n is the number of vertices in the graph.
  - (Better algorithms take order $n + e$ where e is the number of edges.)
- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph.
  - a linear order consistent with the partial order of the graph.



(a)



(b)



(c)

Recoverability

- Need to address the effect of transaction failures on concurrently running transactions
- Example: Schedule 9 is not recoverable

- *Recoverable schedule*
  - if a transaction T_j reads a data item previously written by a transaction T_i ,
  -

- DBMS must ensure that schedules are recoverable.

T_8	T_9
read (A)	
write (A)	
	read (A)
	commit
read (B)	

Schedule 9 (fig 14.14)

Cascading Rollback

-
- Example: If T_{10} fails, T_{11} and T_{12} must also be rolled back.

T_{10}	T_{11}	T_{12}
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)
abort		

Schedule 10 (fig 14.15)

Can lead to the undoing of a significant amount of work

Cascadeless Schedules

- Cascading rollbacks cannot occur if
  - for each pair of transactions T_i and T_j
  - such that T_j reads a data item previously written by T_i,
  -

-

- It is *desirable* to restrict the schedules to those that are cascadeless

END OF CHAPTER 14