Advanced DB

CHAPTER 15
TRANSACTION MNGMNT

Chapter 15: Transactions

- Transaction Concept
- Transaction State
- Implementation of Atomicity and Durability
- Concurrent Executions
- Serializability
- Recoverability
- Implementation of Isolation
- Testing for Serializability

Transactions

- What is a transaction?
  - Logical unit of work (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
- Delayed disk write
  - 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

Properties of a Transaction

ACID properties

- **A**tomicity

  “all or nothing”

- **C**onsistency (Correctness)

  Move from a consistent state to another consistent state

- **I**solation

  Should not be interfered by other transactions (concurrency)

- **D**urability

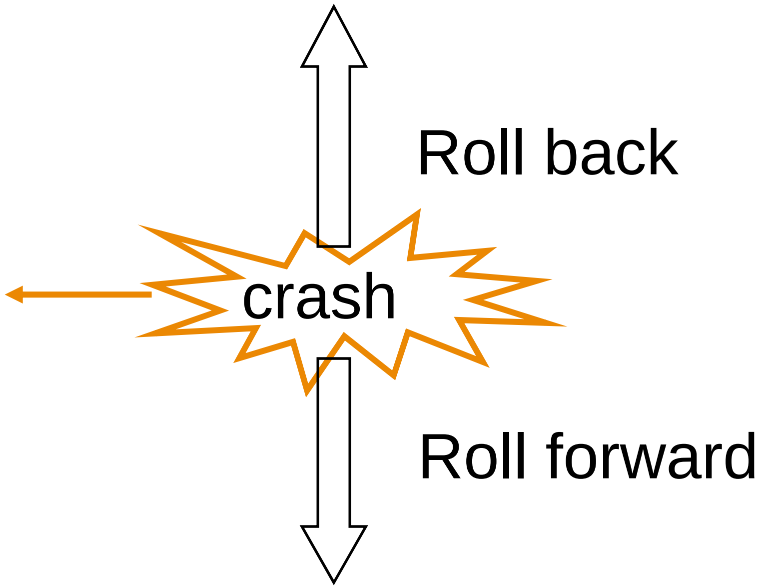
  The effect of a completed transaction should be durable & public

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?

- Responsibility of programmer

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
```

- Serial Execution VS Concurrent Execution

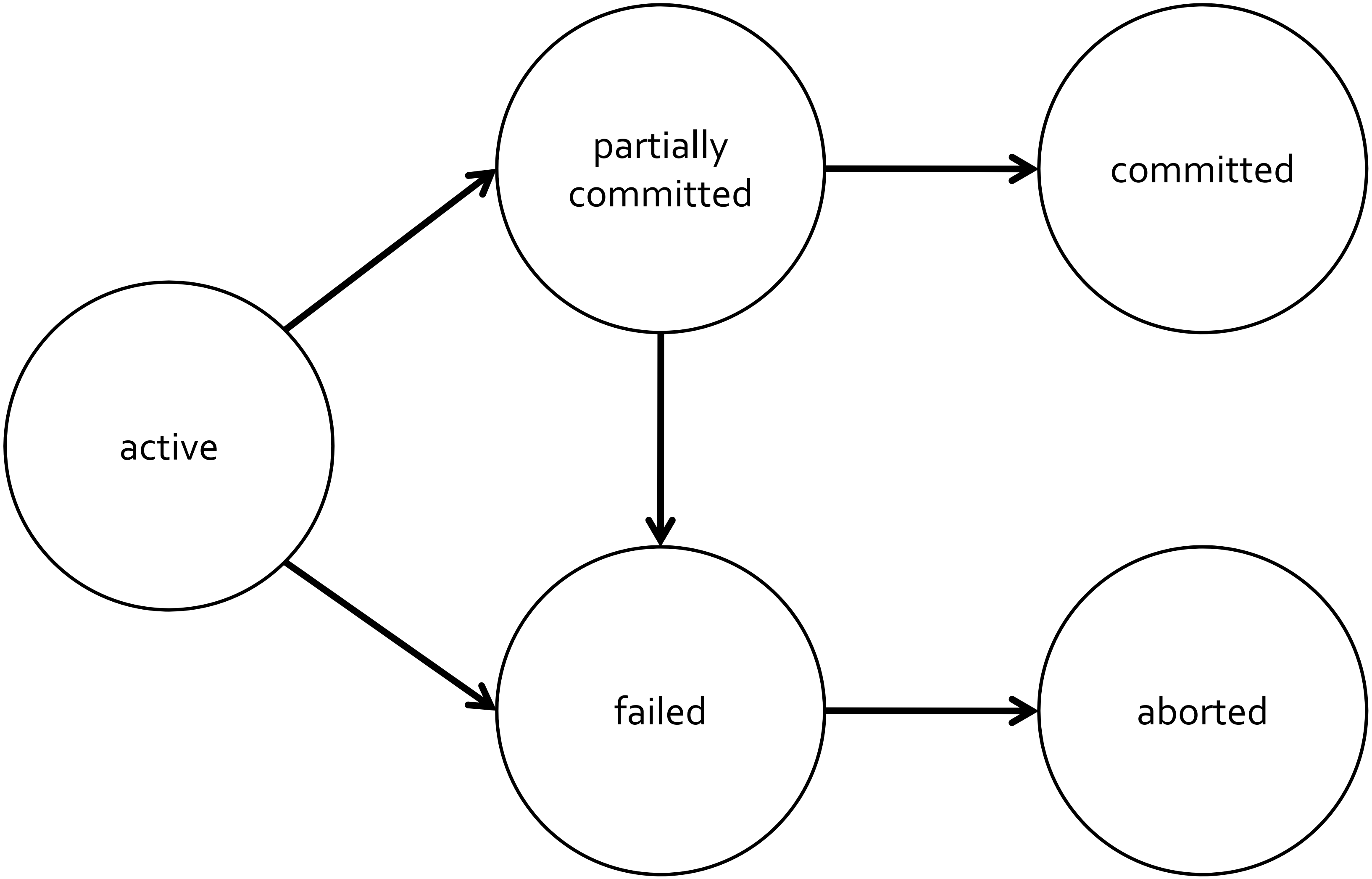
Durability

- 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



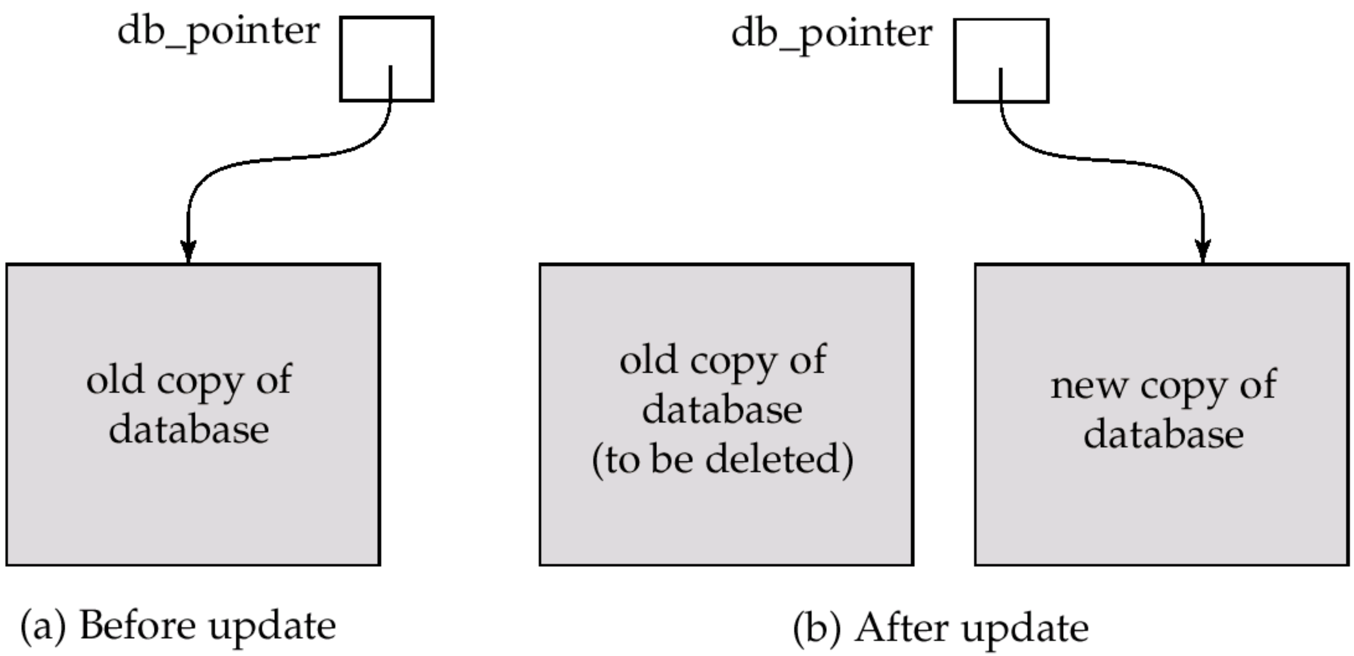
Transaction States (Cont.)

- *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*.

Implementation of Atomicity and Durability

- The *recovery-management* component of a database system implements the support for atomicity and durability.
- The *shadow-database* scheme:
  - assume that only one transaction is active at a time.
  - a pointer called db_pointer always points to the current consistent copy of the database.
  - all updates are made on a *shadow copy* of the database, and **db_pointer** is made to point to the updated shadow copy only after the transaction reaches partial commit and all updated pages have been flushed to disk.
  - in case transaction fails, old consistent copy pointed to by **db_pointer** can be used, and the shadow copy can be deleted.

Shadow Database Scheme



- Assumes disks to not fail
- Useful for text editors
- But extremely inefficient for large databases:
  - executing a single transaction requires copying the *entire* database.
  - Will see better schemes in Chapter 17.

Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system
  - increased processor and disk utilization
  - reduced average response time
- *Concurrency control schemes* – mechanisms to achieve isolation
  - to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database
  - (Chapter 16)

Schedules

- *Schedules*
  - sequences that indicate the chronological order in which instructions of concurrent transactions are executed
  - a 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

- Let T_1 transfer \$50 from A to B, and T_2 transfer 10% of the balance from A to B.
- The following is a serial schedule in which T_1 is followed by T_2.

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

Schedule 1 (fig 15.3)

Example Schedule (Cont.)

- Let T_1 and T_2 be the transactions defined previously.
- Schedule 3 is not a serial schedule, but it is equivalent to Sch.1.
- *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)	
	read(B)
	$B := B + temp$
	write(B)

Schedule 3 (fig 15.5)

Example Schedules (Cont.)

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

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)	
	$B := B + temp$
	write(B)

Schedule 4 (fig 15.6)

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.
  1. conflict serializability
  2. view serializability (not covered in this semester)
- We ignore operations other than *read* and *write* instructions.

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)	

Conflict Serializability (Cont.)

- A conflict serializable schedule

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

Recoverability

- Need to address the effect of transaction failures on concurrently running transactions
- *Recoverable schedule*
  - if a transaction T_j reads a data item previously written by a transaction T_i ,
  - the commit of T_i appears before the commit of T_j.
- The following schedule (Schedule 11) is not recoverable if T_9 commits immediately after the read

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

- DBMS must ensure that schedules are recoverable.

Cascading Rollback

- A single transaction failure can lead to a series of transaction rollbacks.
- 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)

Can lead to the undoing of a significant amount of work

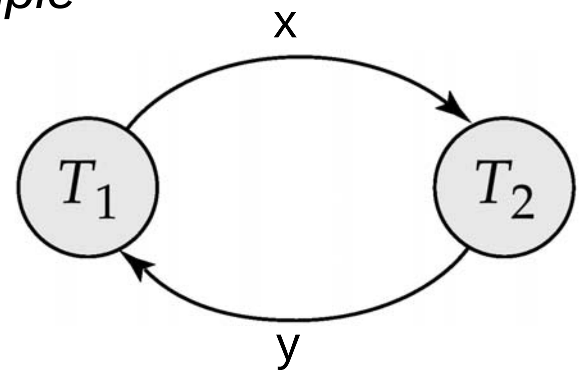
Cascadeless Schedules

- 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,
  - the commit of T_i appears before the read of T_j
- Every cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless

Testing for Serializability

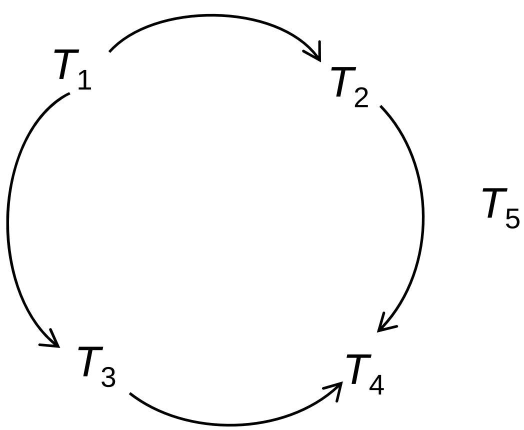
- Consider some schedule of a set of transactions T_1, T_2, ..., T_n
- Precedence graph — a direct graph where the vertices are the transactions (names).
- Draw an arc from T_i to T_j if
  - the two transaction conflict, and
  - T_i accessed the data item on which the conflict arose
- A schedule is conflict serializable if and only if its precedence graph is acyclic

example



Example

T_1	T_2	T_3	T_4	T_5
	read(X)			
read(Y)				
read(Z)				
				read(V)
				read(W)
				read(W)
	read(Y)			
	write(Y)			
		write(Z)		
read(U)				
			read(Y)	
			write(Y)	
			read(Z)	
			write(Z)	
read(U)				
write(U)				



Precedence graph for the schedule

Serializability Tests vs. Concurrency Control

- Testing a schedule for serializability *after* it has executed is a little too late!
- Goal
  - to develop concurrency control protocols that will assure serializability
  - protocol will impose a discipline that avoids nonseralizable schedules (Chapter 16)
- Tests for serializability help understand why a concurrency control protocol is correct.

END OF CHAPTER 15