CS 2550 / Spring 2006
Principles of Database Systems

09 – Transactions

Alexandros Labrinidis
University of Pittsburgh

New Chapter

Chapter 15

Transactions

Roadmap

- **Concept of Transaction**
  - ACID properties
- Transaction State
- Implementation of Atomicity and Durability
- Concurrent Execution of Transactions

Concept of Transaction

- A ***transaction*** is a ***unit*** of program execution that accesses and possibly updates various data items.
- A transaction must see a consistent database.
  - During transaction execution the database may be inconsistent.
  - When the transaction is committed, the database must be consistent.
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

ACID Properties

To preserve integrity of data, the database system must ensure:

- **Atomicity**
  - Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency**
  - Execution of a transaction alone preserves the consistency of the database.
- **Isolation**
  - Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions.
- **Durability**
  - After a transaction completes successfully, changes it has made to the database persist, even if there are system failures.

ACID Properties (cont.)

- **Consistency**
  - Ensuring Consistency is up to the application programmer
- **Atomicity** (Transaction Management)
  - Keep old values around, until sure all of transaction completes
  - Ensuring Atomicity is up to the database system
- **Durability** (Recovery Management)
  - Ensuring Durability is up to the database system
- **Isolation** (Concurrency Control)
  - Intermediate transaction results must be hidden from other concurrently executed transactions.
  - That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished.

Example of funds transfer transaction

- Transaction to transfer $50 from account *A* to account *B*
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B*)

- Consistency requirement
  - sum of *A* and *B* is unchanged by the execution of the transaction
- Atomicity requirement
  - if the transaction fails after step 3 and before step 6, the system should ensure that its updates are not reflected in the database, else an inconsistency will result.

Example of funds transfer (Cont.)

- Durability requirement
  - once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist despite failures.

- Isolation requirement
  - if between steps 3 and 6, another transaction is allowed to access the partially updated database, it will see an inconsistent database (the sum *A* + *B* will be less than it should be).
  - Can be ensured trivially by running transactions ***serially***: one after the other.
  - However, executing multiple transactions concurrently has significant benefits, as we will see.

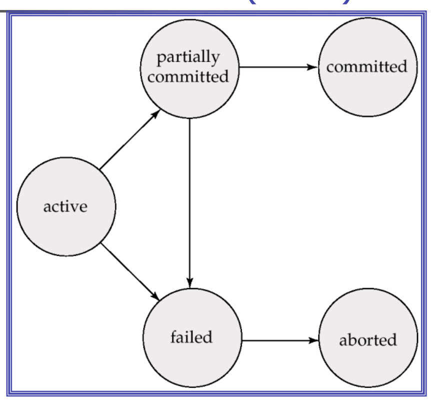
Roadmap

- **Concept of Transaction**
  - ACID properties
- **Transaction State**
- Implementation of Atomicity and Durability
- Concurrent Execution of Transactions

Transaction State

- **Active,** the initial state; the transaction stays in this state while it is executing
- **Partially committed,** after the final statement has been executed.
- **Failed,** after the discovery that normal execution can no longer proceed.
- **Aborted,** after the 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.*

Transaction State (Cont.)



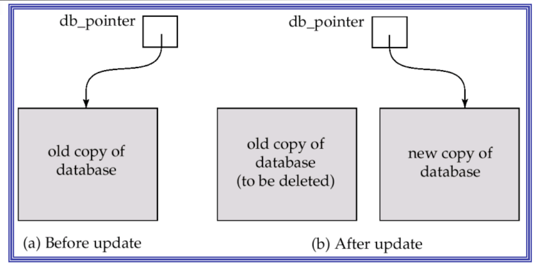
Roadmap

- **Concept of Transaction**
  - ACID properties
- Transaction State
- **Implementation of Atomicity and Durability**
- Concurrent Execution of Transactions

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.

Atomicity and Durability (cont.)



- Assumes:
  - disks do not fail
  - what else?
- Useful for text editors, but extremely inefficient for large databases
  - executing a single transaction requires copying the *entire* database.

Roadmap

- **Concept of Transaction**
  - ACID properties
- **Transaction State**
- Implementation of Atomicity and Durability
- **Concurrent Execution of Transactions**

Concurrent Execution

- Multiple transactions are allowed to run concurrently
- Advantages are:
  - **increased processor and disk utilization**, leading to better transaction *throughput:* one transaction can be using the CPU while another is reading from or writing to the disk
  - **reduced average response time** for transactions: short transactions need not wait behind long ones.
- *Concurrency control schemes*
  - mechanisms to achieve isolation, i.e., to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

Schedules

- sequences of operations 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 Schedule 1

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)

- T_1 transfers $50 from A to B
- T_2 transfers 10% of the balance from A to B
- This is a serial schedule, in which T_1 is followed by T_2.

Example Schedule 3

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)

- T_1 transfers $50 from A to B
- T_2 transfers 10% of the balance from A to B
- This is a not serial schedule, but is equivalent to serial Schedule 1
- In both Schedule 1 and Schedule 3, the sum A+B remains the same

Example Schedule 4

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)

- T_1 transfers $50 from A to B
- T_2 transfers 10% of the balance from A to B
- This concurrent schedule does not preserve the value of the sum $A + B$.