Transactions and ACID properties

Introduction to Database Design 2011, Lecture 13

Rasmus Ejlers Møgelberg

Course overview

- Communicating with DBMSs
- Designing databases
- Making databases efficient
- Making databases reliable

Overview

- Transactions
- ACID properties
- Concurrency and safe schedules

Transactions

- Transactions are units of work
- Example
  - Read balance of account A
  - Compute new balance = balance(A) - 50
  - Write new balance of A
  - Read balance of account B
  - Compute new balance = balance(B) + 50
  - Write balance of account B
- Transaction should be atomic: all or nothing

Transactions

- In database practice transactions are sequences of SQL statements
- Possibly intertwined by computations
- Can be written in
  - programming language (e.g. Java) accessing a database
  - procedural component of SQL
- SQL: `begin atomic ... end`
- Here we simplify and consider just sequences of reads and writes

ACID

- ACID stands for
  - Atomicity
  - Consistency
  - Isolation
  - Durability

Consistency

- Database must always be consistent wrt real world rules, e.g.,
  - Integrity constraints such as referential integrity must be satisfied
  - Rules of real world situation must be satisfied, e.g.,
    - Account balance must always be above a certain number (e.g. 0)
- Should also reflect real world as it is now
  - e.g. balance stored should correspond to actual balance of account

Transactional consistency

- Transaction leaves database in consistent state
  - (may assume database consistent before transaction start)
- May be required to satisfy other rules, e.g. leave the sum of the balances unchanged
  - (no money created or lost)
- During transaction consistency requirement may be violated temporarily
- Transactional consistency responsibility of transaction designer

Atomicity

- Transaction can be considered a unit of work
  - All or nothing!
- Consistency is impossible without atomicity
- Sometimes not possible to complete a started transaction, e.g.
  - In case of hardware failure or loss of connection
  - The application program may choose to abandon transaction
  - The DBMS may refuse to complete the transaction
- In these cases we say that transaction **fails**

Atomicity

- If a transaction fails it must be **aborted**
- This involves **rolling back** the transaction
- i.e., undoing all changes made by transaction
- Concurrency makes this complicated
  - e.g., changes made by transaction may have already been read
- Ability to roll back is implemented using a **log** of changes made to the database

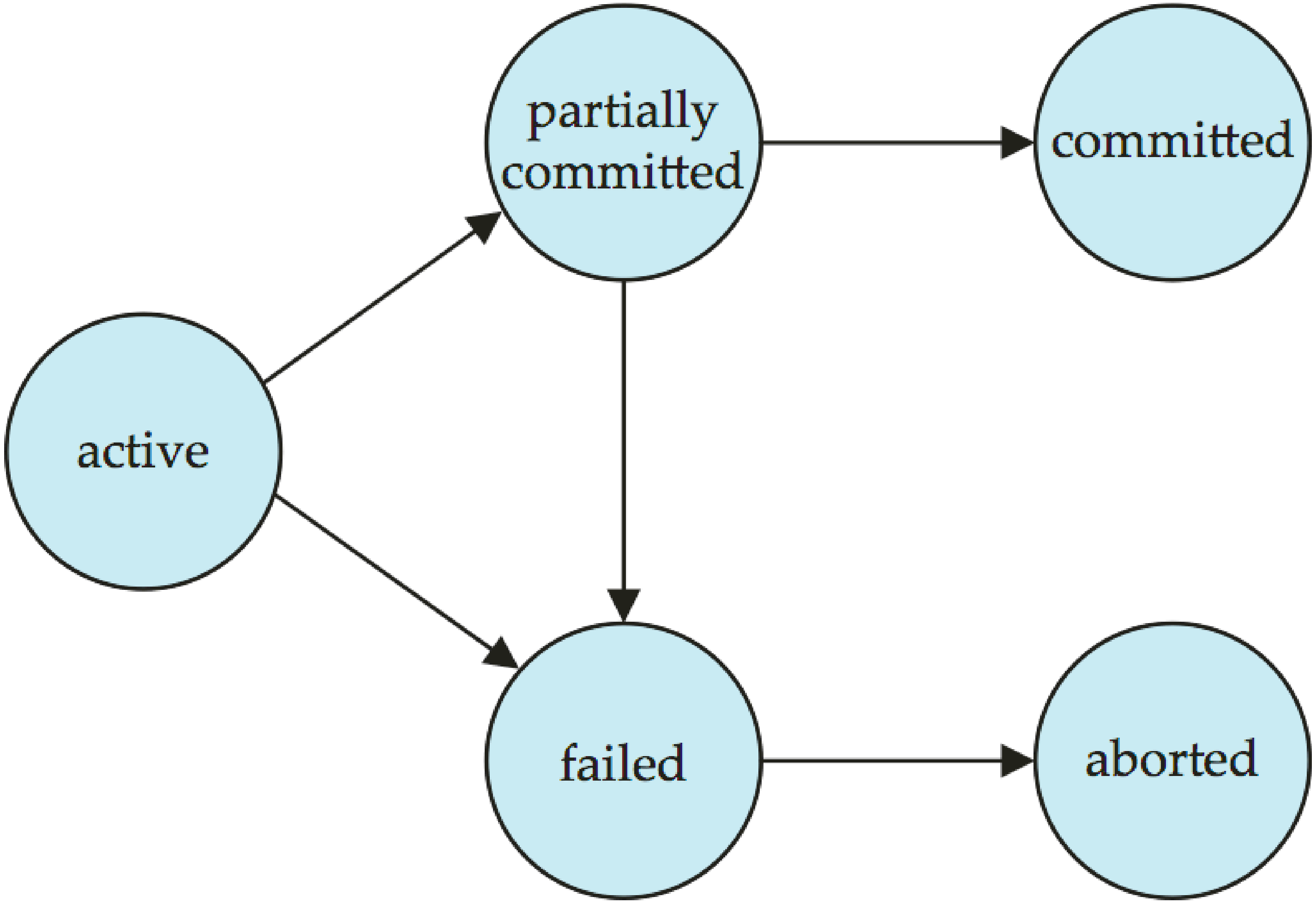
Durability

- Durability is about trustworthy storage
- A transaction that has successfully completed is said to be **committed**
- Changes made by a committed transaction must be durable, i.e., able to survive
  - Power failure
  - Hardware failure etc

Durability

- When committing, changes must be written to non-volatile storage
- In practice, log is written to non-volatile storage
- Storage must be able to survive hardware failure
  - Maintain multiple copies of data
  - RAID

Transaction model



Isolation

- Transactions may not interfere with each other
- In reality transactions are executed concurrently
- Statements of transactions intertwined
- DBMS should create illusion of transactions being executed sequentially
- Isolation is necessary for consistency

Need for concurrency

- Resources operate in parallel
  - Multiple CPUs
  - Data stored on multiple disks
- Computation may be stalled while waiting for data
- Gains of concurrency
  - Increased throughput: idle resources can be utilised
  - Decreased waiting time: new transactions can start executing immediately

Concurrency and safe schedules

Rasmus Ejlers Møgelberg

A serial schedule

T_1	T_2
read(A)	
A := A - 50	
write(A)	
read(B)	
B := B + 50	
write(B)	
	read(A)
	A := A + 20
	write(A)

A good concurrent schedule

T_1	T_2
read(A)	
A := A - 50	
write(A)	
	read(A)
	A := A + 20
	write(A)
read(B)	
B := B + 50	
write(B)	

A bad concurrent schedule

T_1	T_2
read(A)	
A := A - 50	
	read(A)
	A := A + 20
	write(A)
write(A)	
read(B)	
B := B + 50	
write(B)	

Schedules

- The DBMS receives a sequence of read and write requests from different transactions
- A schedule is an ordering of the reads and writes respecting the internal ordering in each transaction

- Two schedules

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

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

- Schedule on right is equivalent to first T_1 then T_2
- It is up to DBMS to avoid bad schedules such as the one on the left

Conflicting operations

- Two operations **commute** if the order in which they are executed does not affect the result

$$\mathrm{read}(A), \mathrm{read}(B) = \mathrm{read}(B), \mathrm{read}(A)$$
$$\mathrm{write}(A), \mathrm{read}(B) = \mathrm{read}(B), \mathrm{write}(A)$$
$$\mathrm{write}(A), \mathrm{write}(B) = \mathrm{write}(B), \mathrm{write}(A)$$
$$\mathrm{read}(A), \mathrm{read}(A) = \mathrm{read}(A), \mathrm{read}(A)$$

- If they do not commute we say that they **conflict**

$$\mathrm{write}(A), \mathrm{read}(A) \neq \mathrm{read}(A), \mathrm{write}(A)$$
$$\mathrm{write}(A), \mathrm{write}(A) \neq \mathrm{write}(A), \mathrm{write}(A)$$

Conflict equivalence

- Two schedules are **conflict equivalent** if they differ only up to swapping commuting operations

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

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

- Executing conflict equivalent schedules gives same result

A non-example

- The following schedules are not conflict equivalent

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

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

- Suppose e.g. T_1 transfers all money available in A to B

Conflict serializability

- A schedule is **conflict serializable** if it is conflict equivalent to a serial schedule
- Example:

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

A non-serializable schedule

- The following is neither conflict equivalent to T_1T_2 nor T_2T_1

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

Serializable schedules

- Serializable schedules are the ‘good schedules’
- Parallel executions of transactions
- But still maintain illusion of serial execution
- DBMS should ensure that only serializable schedules occur
- This is usually done using locks

Detecting non-serializability

T_1	T_2	T_3	T_4
read(A)			
	write(A)		
		read(A)	
		write(B)	
		read(C)	
			write(C)
read(B)			

Precedence graph

- A cycle, so not conflict serializable

$$\begin{tikzcd} T_1 \arrow[r] & T_2 \arrow[r] & T_3 \arrow[r] \arrow[ll, bend left] & T_4 \end{tikzcd}$$

- **Theorem.** A schedule is conflict serializable if and only if its precedence graph is acyclic

Another example

T_1	T_2	T_3	T_4
read(A)			
	write(C)		
write(B)			
		write(B)	
		read(D)	
			read(C)
	write(A)		
			write(D)

Precedence graph

$$
\begin{tikzcd}
& T_2 \arrow[dr] & \\
T_1 \arrow[ur] \arrow[dr] & & T_4 \\
& T_3 \arrow[ur] &
\end{tikzcd}
$$

- Equivalent serial schedules

$$T_1 T_2 T_3 T_4$$

$$T_1 T_3 T_2 T_4$$

Yet another example

T_1	T_2	T_3
	read(B)	
write(B)		
write(A)		
	write(A)	
		write(A)

- Not conflict serializable
- But result the same as running

$$T_2 T_1 T_3$$

View serializability

- Two schedules are **view equivalent** if
  - Corresponding reads in the two schedules always read same value
  - The changes made to the database are always the same
- A schedule is **view serializable** if it is view equivalent to a serial schedule
- Schedule on previous slide is view serializable

View serializability

- The following is not view serializable

T_1	T_2	T_3
	read(B)	
write(B)		
read(A)		
	write(A)	
		write(A)

- To see this need to check all serial combinations

View serializability

- We have two notions of serializability
- Conflict serializable schedules are also view serializable
- (because swapping commuting operations does not change behaviour of schedule)
- View serializable schedules need not be conflict serializable
- (see example a few slides back)

Summary

- ACID requirements for databases
  - Atomicity, consistency, isolation, durability
- Isolation is an illusion
  - In reality transactions are evaluated in parallel
- Two notions of good schedules
  - Conflict serializability
  - View serializability
- For exam you should be able to use these