Introduction to Transaction Management

CMPSCI 445

Slide content adapted from Ramakrishnan & Gehrke

Concurrency Control

- Concurrent execution of user programs is essential for good DBMS performance
- We must also cope with partial operations
- The **transaction** is the foundation for:
  - Concurrent execution
  - Recovery from system failure, incomplete ops

What is a Transaction?

- A **transaction** is the DBMS's abstract view of a user program: a sequence of reads and writes.

A simple transaction

- Imagine a simple banking application
  - Two database objects:
    - **A: balance of account A**
    - **B: balance of account B**
- Transaction T1:
  - "Transfer \$100 from account B to account A".

T1: Transfer
Begin A=A+100 B=B-100 End

The ACID Properties

- Database systems ensure the ACID properties:
  - Atomicity
  - Consistency
  - Isolation
  - Durability

Atomicity

- A very important property guaranteed by the DBMS for all transactions is that they are atomic.
  - User can think of a Xact as executing **all its actions** in one step, or executing **no actions at all**.
  - DBMS logs all actions so that it can undo the actions of aborted transactions.
- If it succeeds, the effects of write operations persist (commit);
- If it fails, no effects of write operations persist (abort)

Consistency

- Each transaction must leave the database in a consistent state if the DB is consistent when the transaction begins.
  - DBMS will enforce some ICs, depending on the ICs declared in CREATE TABLE statements.
  - Beyond this, the DBMS does not really understand the semantics of the data. (e.g., it does not understand how the interest on a bank account is computed).
- In banking example, sum (A + B) should be unchanged by execution.

Isolation

- Many concurrent transactions are running at one time.
- Each transaction should be isolated from the effects of other transactions
- Transactions should not be exposed to intermediate states created by other transactions.
- The net effect of concurrently running {T1 and T2 and T3} is equivalent to <u>some</u> serial order
  - No guarantee which serial order

Durability

- If transaction completes, its effects will persist in the database.
- In particular, if the system crashes before effects are written to disk, they will be redone
- Recovery manager is responsible for this.

The ACID Properties

- Database systems ensure the ACID properties:
  - Atomicity: all operations of transaction reflected properly in database, or none are.
  - Consistency: each transaction in isolation keeps the database in a consistent state (this is the responsibility of the user).
  - Isolation: should be able to understand what's going on by considering each separate transaction independently.
  - Durability: updates stay in the DBMS!!!

Two transactions

- “Transfer $100 from account B to account A”
- “Add 6% interest to accounts A and B”

T1: Transfer
Begin
A=A+100
B=B-100
End

T2: Interest
Begin
A=1.06*A
B=1.06*B
End

Serial execution: T1, then T2

- Starting balances
  - A = 1000
  - B = 2000
- Execute T1
  - A = 1100
  - B = 1900
- Execute T2
  - A = 1166
  - B = 2014

T1: Transfer
Begin
A=A+100
B=B-100
End

T2: Interest
Begin
A=1.06*A
B=1.06*B
End

Serial execution: T2, then T1

- Starting balances
  - A = 1000
  - B = 2000
- Execute T2
  - A = 1060
  - B = 2120
- Execute T1
  - A = 1160
  - B = 2020

T2: Interest
Begin
A=1.06*A
B=1.06*B
End

T1: Transfer
Begin
A=A+100
B=B-100
End

Interleaved execution

- ❖ What other results are possible if operations of T1 and T2 are interleaved?

- Starting balances
  - A = 1000
  - B = 2000

T1: Transfer	T2: Interest
…	…
A=A+100	A=1.06*A
…	…
B=B-100	B=1.06*B
…	…

Interleaving operations

T1: Transfer	T2: Interest
A=A+100	
	A=1.06*A
B=B-100	
	B=1.06*B

Is this interleaving okay?

Interleaving operations

T1: Transfer	T2: Interest
A=A+100	
	A=1.06*A
	B=1.06*B
B=B-100	

How about this interleaving?

Goal: interleaved execution, with serial effects

- There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together. However, the net effect *must* be equivalent to these two transactions running **serially** in some order.

Scheduling Transactions

- A transaction is seen by DBMS as sequence of reads and writes
  - read of object O denoted R(O)
  - write of object O denoted W(O)
  - must end with Abort or Commit
- A schedule of a set of transactions is a list of all actions where order of two actions from any transaction must match order in that transaction.

A schedule

T1: Transfer	T2: Interest
A=A+100	
	A=1.06*A
	B=1.06*B
B=B-100	

T1: Transfer	T2: Interest
Read(A)	
Write(A)	
	Read(A)
	Write(A)
	Read(B)
	Write(B)
Read(B)	
Write(B)	

Scheduling Transactions

- *Serial schedule:* Schedule that does not interleave the actions of different transactions.
- *Equivalent schedules*: For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule.
- *Serializable schedule*: A schedule that is equivalent to some serial execution of the transactions.
-

Serializable Schedule

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)
	Commit
Commit	

When can actions be re-ordered?

- Let I, J be two consecutive actions of T1 and T2
  - I=Read(O), J=Read(O)
  - I=Read(O), J=Write(O)
  - I=Write(O), J=Read(O)
  - I=Write(O), J=Write(O)
- If I and J are both reads, then they can be freely reordered.
- In all other cases, order impacts outcome of schedule.

Conflicting operations

- Two operations **conflict** if:
  - they operate on the same data object, and
  - at least one is a WRITE.
- Schedule outcome is determined by order of the conflicting operations.

Conflict Serializable Schedules

- Two schedules are conflict equivalent if:
  - Involve the same actions of the same transactions
  - Every pair of conflicting actions (of committed trans) are ordered the same way.
  - Alternatively: S can be transformed to S′ by swaps of non-conflicting actions.
- Schedule S is conflict serializable if S is conflict equivalent to some serial schedule

Every conflict serializable schedule is serializable.

(exception: dynamic databases)

Conflict-serializable schedule

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)
	Commit
Commit	

Not conflict-serializable

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	Commit
R(B)	
W(B)	
Commit	

Precedence graphs

- Directed graph derived from schedule S:
  - Vertex for each transaction
  - Edge from Ti to Tj if:
    - Ti executes Write(O) before Tj executes Read(O)
    - Ti executes Read(O) before Tj executes Write(O)
    - Ti executes Write(O) before Tj executes Write(O)

If edge Ti -> Tj appears in precedence graph, then in any serial schedule equivalent to S, Ti must appear before Tj.

Dependency Graph

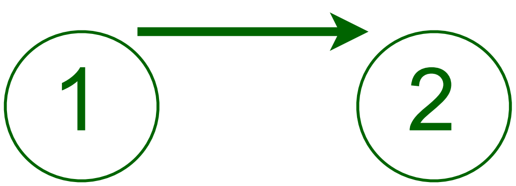
- Theorem: A schedule is **conflict serializable** if and only if its dependency graph is acyclic.

(A serializable order can be found by topological sort of the dependency graph.)

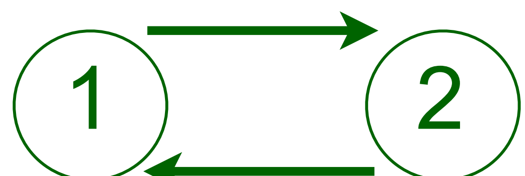
Construct precedence graphs:

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)
	Commit
Commit	

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	Commit
R(B)	
W(B)	
Commit	



Conflict serializable



Non-conflict serializable

Construct precedence graph:

T1	T2	T3
R(A)		
	W(A)	
	Commit	
W(A)		
Commit		
		W(A)
		Commit