Transaction Processing: Concurrency Control

CPS 216

Advanced Database Systems

Announcements (April 26)

- Homework #4 due this Thursday (April 28)
    - Sample solution will be available on Thursday
- Project demo period: April 28 – May 1
    - Remember to email me to sign up for a 30-minute slot
- Final exam on Monday, May 2, 2-5pm
    - 3 hours—no time pressure!
    - Open book, open notes
    - Comprehensive, but with emphasis on the second half of the course and materials exercised in homework
- Sample final (from last year) available
    - Solution will be available on Thursday

Transactions

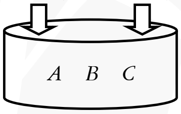
- Transaction: a sequence of operations with ACID properties
    - Atomicity: TX's are either completely done or not done at all
    - Consistency: TX's should leave the database in a consistent state
    - Isolation: TX's must behave as if they are executed in isolation
    - Durability: Effects of committed TX's are resilient against failures
- SQL transactions

```
-- Begins implicitly
SELECT …;
UPDATE …;
ROLLBACK | COMMIT;
```

Concurrency control

- Goal: ensure the "I" (isolation) in ACID

T_1:	T_2:
read(A);	read(A);
write(A);	write(A);
read(B);	read(C);
write(B);	write(C);
commit;	commit;



Good versus bad schedules

Good!

T_1	T_2
r(A)	
w(A)	
r(B)	
w(B)	
	r(A)
	w(A)
	r(C)
	w(C)

Bad!

T_1	T_2
r(A) Read 400	
	r(A) Read 400
Write 400 – 100 w(A)	
	w(A) Write 400 – 50
r(B)	
	r(C)
w(B)	
	w(C)

Good?

T_1	T_2
r(A)	
w(A)	
	r(A)
	w(A)
r(B)	
	r(C)
w(B)	
	w(C)

Serial schedule

- Execute transactions in order, with no interleaving of operations
    - T_1.r(A), T_1.w(A), T_1.r(B), T_1.w(B), T_2.r(A), T_2.w(A), T_2.r(C), T_2.w(C)
    - T_2.r(A), T_2.w(A), T_2.r(C), T_2.w(C), T_1.r(A), T_1.w(A), T_1.r(B), T_1.w(B)

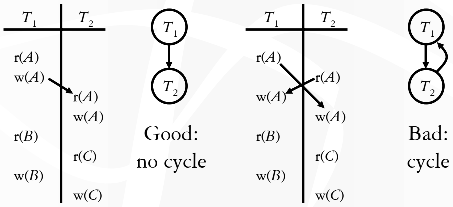
    ☞Isolation achieved by definition!
- Problem: no concurrency at all
- Question: how to reorder operations to allow more concurrency

Conflicting operations

- ❖ Two operations on the same data item conflict if at least one of the operations is a write
  - ▪ r(X) and w(X) conflict
  - ▪ w(X) and r(X) conflict
  - ▪ w(X) and w(X) conflict
  - ▪ r(X) and r(X) do not
  - ▪ r/w(X) and r/w(Y) do not
- ❖ Order of conflicting operations matters
  - ▪ E.g., if T_1.r(A) precedes T_2.w(A), then conceptually, T_1 should precede T_2

Precedence graph

- ❖ A node for each transaction
- ❖ A directed edge from T_i to T_j if an operation of T_i precedes and conflicts with an operation of T_j in the schedule



Conflict-serializable schedule

- ❖ A schedule is conflict-serializable iff its precedence graph has no cycles
- ❖ A conflict-serializable schedule is equivalent to some serial schedule (and therefore is "good")
  - ▪ In that serial schedule, transactions are executed in the topological order of the precedence graph
  - ▪ You can get to that serial schedule by repeatedly swapping adjacent, non-conflicting operations from different transactions

Locking

- ❖ Rules
  - ▪ If a transaction wants to read an object, it must first request a shared lock (S mode) on that object
  - ▪ If a transaction wants to modify an object, it must first request an exclusive lock (X mode) on that object
  - ▪ Allow one exclusive lock, or multiple shared locks

Mode of the lock requested

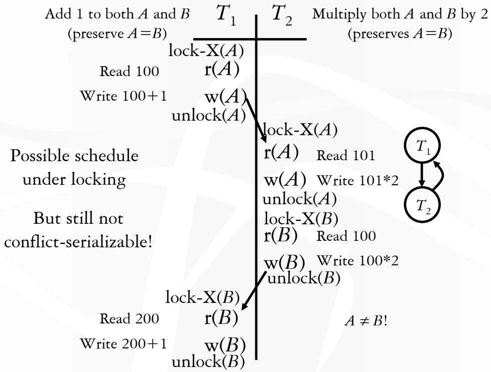
	S	X
S	Yes	No
X	No	No

Mode of lock(s) currently held by other transactions

Grant the lock?

Compatibility matrix

Basic locking is not enough

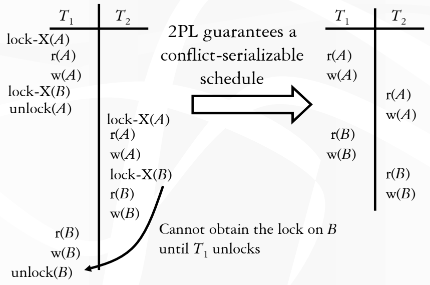


Possible schedule under locking

But still not conflict-serializable!

Two-phase locking (2PL)

- ❖ All lock requests precede all unlock requests
  - ▪ Phase 1: obtain locks, phase 2: release locks



2PL guarantees a conflict-serializable schedule

Problem of 2PL

T_1	T_2
r(A)	
w(A)	
	r(A)
	w(A)
r(B)	
w(B)	
	r(B)
	w(B)
…	
Abort!	

- ❖ T_2 has read uncommitted data written by T_1
- ❖ If T_1 aborts, then T_2 must abort as well
- ❖ Cascading aborts possible if other transactions have read data written by T_2
- ❖ Even worse, what if T_2 commits before T_1?
  - Schedule is not recoverable if the system crashes right after T_2 commits

Strict 2PL

- ❖ Only release locks at commit/abort time
  - A writer will block all other readers until the writer commits or aborts

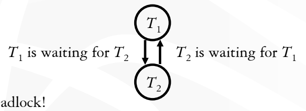
☞ Used in most commercial DBMS (except Oracle)

Deadlocks

T_1	T_2
lock-X(A)	
r(A)	
w(A)	
	lock-X(B)
	r(B)
	w(B)
lock-X(B)	lock-X(A)
r(B)	r(A)
w(B)	w(A)

Deadlock!

Deadlock: cycle in the wait-for graph

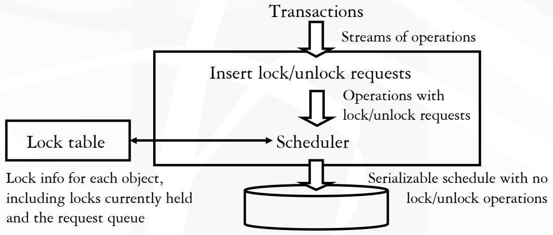


Dealing with deadlocks

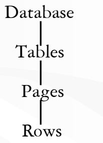
- ❖ Impose an order for locking objects
  - Must know in advance which objects a transaction will access
- ❖ Timeout
  - If a transaction has been blocked for too long, just abort
- ❖ Prevention
  - Idea: abort more often, so blocking is less likely
  - Suppose T is waiting for T'
    - Wait/die scheme: Abort T if it has a lower priority; otherwise T waits
    - Wound/wait scheme: Abort T' if it has a lower priority; otherwise T waits
- ❖ Detection using wait-for graph
  - Idea: deadlock is rare, so only deal it when it becomes an issue
  - When do we detect deadlocks?
  - Which transactions do we abort in case of deadlock?

Implementation of locking

- ❖ Do not rely on transactions themselves to lock/unlock explicitly
- ❖ DBMS inserts lock/unlock requests automatically


Transactions
Streams of operations
Insert lock/unlock requests
Operations with lock/unlock requests
Lock table
Scheduler
Lock info for each object, including locks currently held and the request queue
Serializable schedule with no lock/unlock operations

Multiple-granularity locks



- ❖ Hard to decide what granularity to lock
  - Trade-off between overhead and concurrency
- ❖ Granularities form a hierarchy
- ❖ Allow transactions to lock at different granularity, using intention locks
  - S, X: lock the entire subtree in S, X mode, respectively
  - IS: intend to lock some descendent in S mode
  - IX: intend to lock some descendent in X mode
  - SIX (= S + IX): lock the entire subtree in S mode; intend to lock descendent in X mode

Multiple-granularity locking protocol

Mode of the lock requested

Mode of lock(s) currently held by other transactions

	S	X	IS	IX	SIX
S	Yes		Yes		
X					
IS	Yes		Yes	Yes	Yes
IX			Yes	Yes	
SIX			Yes		

Grant the lock?

Compatibility matrix

- ❖ Lock: before locking an item, *T* must acquire intention locks on all ancestors of the item
  - To get S or IS, must hold IS or IX on parent
    - What if *T* holds S or SIX on parent?
  - To get X or IX or SIX, must hold IX or SIX on parent
- ❖ Unlock: release locks bottom-up
- ❖ 2PL must also be observed

Examples

- ❖ T_1 reads all of R
  - T_1 gets an S lock on R
- ❖ T_2 scans R and updates a few rows
  - T_2 gets an SIX lock on R, and then occasionally get an X lock for some rows
- ❖ T_3 uses an index to read only part of R
  - T_3 gets an IS lock on R, and then repeatedly gets an S lock on rows it needs to access

Phantom problem revisited

- ❖ Lock everything read by a transaction → reads are repeatable, but may see phantoms
- ❖ Example: different average

```
▪ -- T1:                    -- T2:
                            SELECT AVG(GPA)
                            FROM Student WHERE age = 10;
  INSERT INTO Student
  VALUES(789, 'Nelson', 10, 1.0);
  COMMIT;
                            SELECT AVG(GPA)
                            FROM Student WHERE age = 10;
                            COMMIT;
```

☞ How do you lock something that does not exist yet?

Solutions

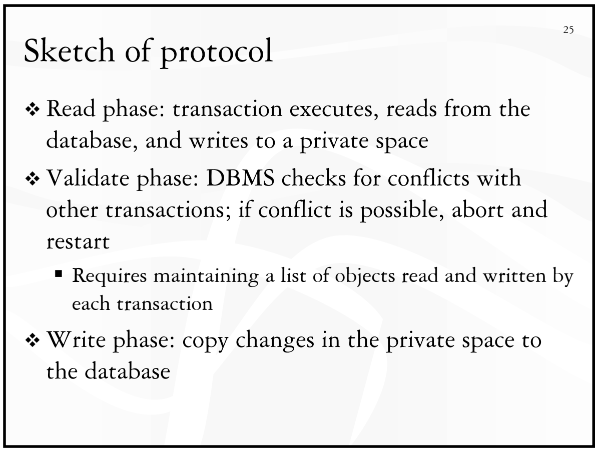
- ❖ Index locking
  - Use the index on *Student*(*age*)
  - T_2 locks the index block(s) with entries for $age = 10$
    - If there are no entries for $age = 10$, T_2 must lock the index block where such entries *would* be, if they existed!
- ❖ Predicate locking
  - "Lock" the predicate ($age = 10$)
  - Reason with predicates to detect conflicts
  - Expensive to implement

Concurrency control without locking

- ❖ Optimistic (validation-based)
- ❖ Timestamp-based
- ❖ Multi-version (Oracle, PostgreSQL)

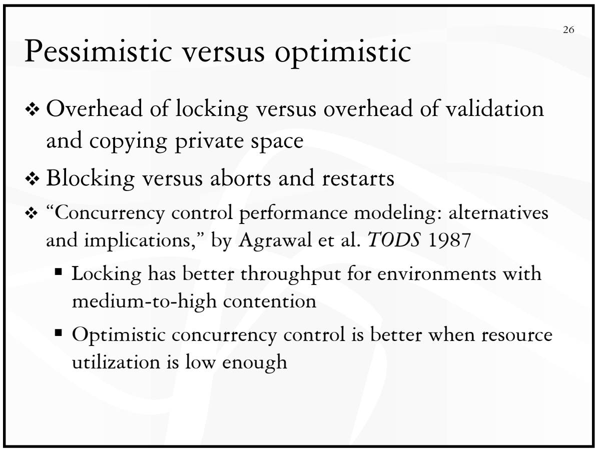
Optimistic concurrency control

- ❖ Locking is pessimistic
  - Use blocking to avoid conflicts
  - Overhead of locking even if contention is low
- ❖ Optimistic concurrency control
  - Assume that most transactions do not conflict
  - Let them execute as much as possible
  - If it turns out that they conflict, abort and restart



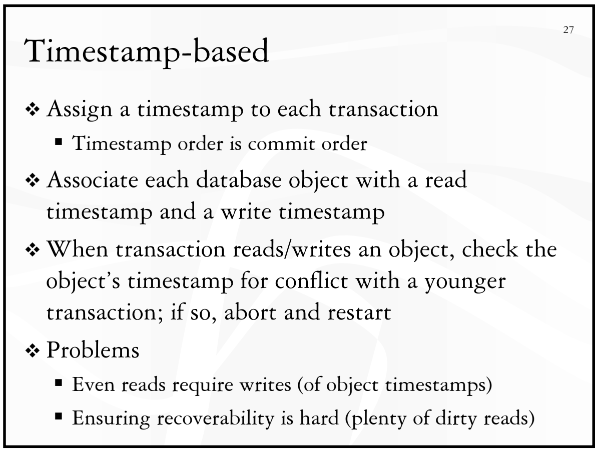
Sketch of protocol

- Read phase: transaction executes, reads from the database, and writes to a private space
- Validate phase: DBMS checks for conflicts with other transactions; if conflict is possible, abort and restart
  - Requires maintaining a list of objects read and written by each transaction
- Write phase: copy changes in the private space to the database



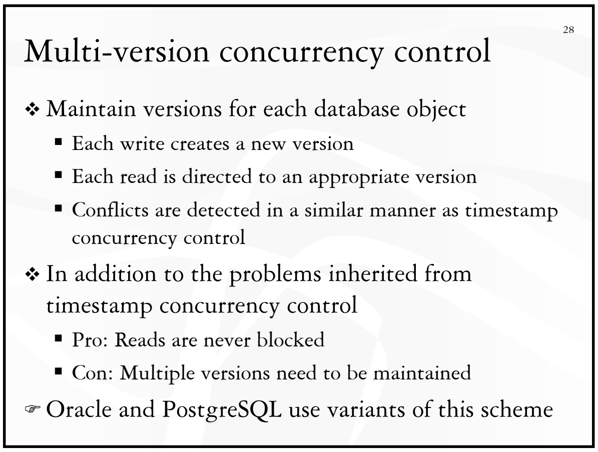
Pessimistic versus optimistic

- Overhead of locking versus overhead of validation and copying private space
- Blocking versus aborts and restarts
- "Concurrency control performance modeling: alternatives and implications," by Agrawal et al. *TODS* 1987
  - Locking has better throughput for environments with medium-to-high contention
  - Optimistic concurrency control is better when resource utilization is low enough



Timestamp-based

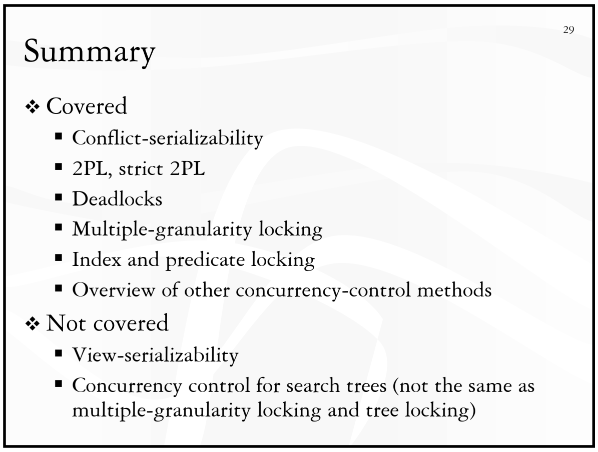
- Assign a timestamp to each transaction
  - Timestamp order is commit order
- Associate each database object with a read timestamp and a write timestamp
- When transaction reads/writes an object, check the object's timestamp for conflict with a younger transaction; if so, abort and restart
- Problems
  - Even reads require writes (of object timestamps)
  - Ensuring recoverability is hard (plenty of dirty reads)



Multi-version concurrency control

- Maintain versions for each database object
  - Each write creates a new version
  - Each read is directed to an appropriate version
  - Conflicts are detected in a similar manner as timestamp concurrency control
- In addition to the problems inherited from timestamp concurrency control
  - Pro: Reads are never blocked
  - Con: Multiple versions need to be maintained

☞ Oracle and PostgreSQL use variants of this scheme



Summary

- Covered
  - Conflict-serializability
  - 2PL, strict 2PL
  - Deadlocks
  - Multiple-granularity locking
  - Index and predicate locking
  - Overview of other concurrency-control methods
- Not covered
  - View-serializability
  - Concurrency control for search trees (not the same as multiple-granularity locking and tree locking)