6.814/6.830 Quiz 2 Review

Logistics

- Wednesday during lecture
- 75 min + 5 min to upload solution
- Covers material from lecture 12 to 21
- Open book/notes, but no googling please.
- Email staff for special accommodation

Topics

- Transactions
- Recovery
- Distributed Databases

Serializability

- An ordering of actions in concurrent transactions that is serially equivalent

T1	T2
RA	
	RA
	WA
WA	
RB	
WB	
	RB
	WB

RA: Read A
WA: Write A, may depend on anything read previously

A/B are "objects" – e.g., records, disk pages, etc

Assume arbitrary application logic between reads and writes

Not serially equivalent – T2's write to A is lost, couldn't occur in a serial schedule

In T1-T2, T2 should see T1's write to A
In T2-T1, T1 should see T2's write to A

View Serializability

A particular ordering of instructions in a schedule S is *view equivalent* to a serial ordering S' iff:

- Every value read in S is the same value that was read by the same read in S'.
- The final write of every object is done by the same transaction T in S and S’
- Less formally, all transactions in S “view” the same values they view in S', and the final state after the transactions run is the same.

Conflict Serializability

A schedule is *conflict serializable* if it is possible to swap non-conflicting operations to derive a serial schedule.

Equivalently

For all pairs of conflicting operations {O1 in T1, O2 in T2} either

- O1 always precedes O2, or
- O2 always precedes O1.

Two Phase Locking (2PL) Protocol

- Before every read, acquire a shared lock
- Before every write, acquire an exclusive lock (or "upgrade") a shared to an exclusive lock
- Release locks only after last lock has been acquired, and ops on that object are finished

Strict Two-Phase Locking Protocol

- Before every read, acquire a shared lock
- Before every write, acquire an exclusive lock (or "upgrade") a shared to an exclusive lock
- Release *shared* locks only after last lock has been acquired, and ops on that object are finished
- Release *exclusive* locks only after the transaction commits
- Ensures cascadeless-ness

Rigorous Two-Phase Locking Protocol

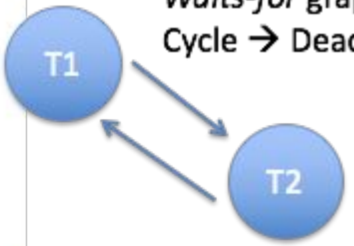
- Before every read, acquire a shared lock
- Before every write, acquire an exclusive lock (or "upgrade") a shared to an exclusive lock
- Release locks only after the transaction commits
- Ensures cascadeless-ness, and
- *Commit order = serialization order*

Deadlocks

- Possible for Ti to hold a lock Tj needs, and vice versa

	T1	T2	
	RA		
	WA		
		RB	
		WB	
T1 waits for T2 →	RB		
	WB		
		RA	← *T2 waits for T1*
		WA	

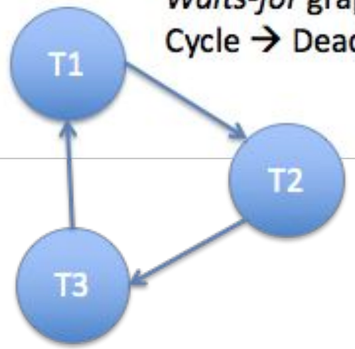
Waits-for graph
Cycle → Deadlock



Complex Deadlocks Are Possible

T1	T2	T3
RA		
WA		
		RC
	RB	
	WB	
		RA ← *T3 waits for T1*
		WA
T1 waits for T2 → RB		
WB		
	RC ← *T2 waits for T3*	
	WC	

Waits-for graph
Cycle → Deadlock



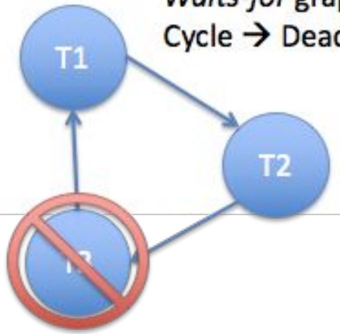
Resolving Deadlock

- Solution: abort one of the transactions
  - Recall: users can abort too

	T1	T2	
	RA		
	WA		
		RB	
		WB	
T1 waits for T2 →	RB		
	WB		
		RC	← ~~*T2 waits for T3*~~
		WC	

Equivalent to T2 - T1

Waits-for graph
Cycle → Deadlock



Optimistic Concurrency Control (OCC)

- Alternative to locking for isolation
- Approach:
  - Store writes in a per-transaction buffer
  - Track read and write sets
  - At commit, check if transaction conflicted with earlier (concurrent) transactions
  - Abort transactions that conflict
  - Install writes at end of transaction
- “Optimistic” in that it does not block, hopes to “get lucky” arrive in serial interleaving

OCC Implementation

- Divide transaction execution in 3 phases
  - **Read**: transaction executes on DB, stores local state
  - **Validate**: transaction checks if it can commit
  - **Write**: transaction writes state to DB

Serial Validation

```
validateAndWrite(pastT[], start_tn, my_read_set, my_write_set)
{
    lock();
    int finish_tn = tnc;  //prior transaction
    bool valid = true;
    for(int t = start_tn + 1; t <= finish_tn; t++)
        if(pastT[t].write_set intersects with my_read_set)
            valid = false;
    if (valid) {
        write_phase();
        tnc = tnc+1;
        tn = tnc;
    }
    unlock();
}
```

1. W(Ti) ∩ R(Tj) ≠ { }, and Tj does not finish writing before Tj starts, Tj must abort
2. W(Ti) ∩ (W(Tj) U R(Tj)) ≠ { }, and Tj overlaps with Ti validation or write phase, Tj must abort

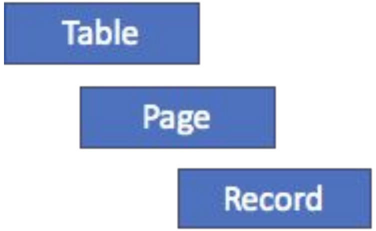
2nd condition doesn't occur because if Ti completes its read phase before Tj, it will also complete its write phase before Tj.

What If Serializability Isn't Needed?

- E.g., application only needs to read committed data
- Databases provide different isolation levels
  - READ UNCOMMITTED
    - Ok to read other transaction's dirty data
  - READ COMMITTED
    - Only read committed values
  - REPEATABLE READS
    - If R1 read A=x, R2 will read A=x ∀ A

- Many database systems default to READ COMMITTED

Intention Locks

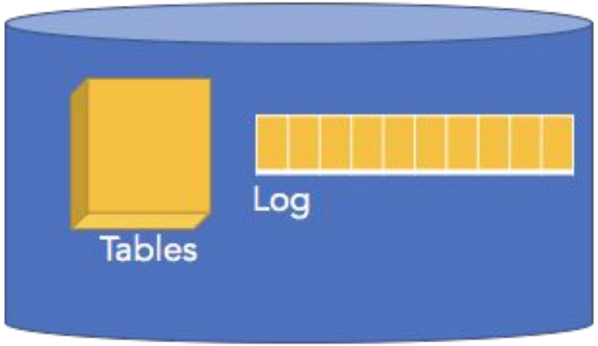


- Suppose T1 wants to read record R1
- Needs to acquire *intention lock* on the Table and Page that T1 is in
- Intention lock marks higher levels with the fact that a transaction has a lock on a lower level
- Intention locks
  - Can be read intention or write intention locks
  - Prevent transactions from writing or reading the whole object when another transaction is working on a lower level
  - New *compatibility table*

Database State During Query Execution

Log records start and end of transactions, and contents of writes done to tables so we can solve both problems

Memory



Disk

After crash, memory is gone!

Problem 1: Some transactions may have written their uncommitted state to tables – need to UNDO

Problem 2: Some transactions may not have flushed all of their state to tables prior to commit – need to REDO

STEAL/NO FORCE ←→ UNDO/REDO

- If we STEAL pages, we will need to UNDO
- If we don't FORCE pages, we will need to REDO

	FORCE	NO FORCE
STEAL	UNDO	UNDO & REDO
NO STEAL	? UNDO	REDO

In SimpleDB, we do FORCE / NO STEAL, and assume DB won't crash between FORCE and COMMIT

All commercial DBs do NO FORCE / STEAL for performance reasons

- If we FORCE pages, we will need to be able to UNDO if we crash between the FORCE and the COMMIT

ARIES Normal Operation

- Two key data structures:
  - *Transaction table* -- list of active transactions
  - *Dirty page table* -- List of pages that have been modified and not yet written to disk

Transaction Table

xactionTable

lastLSN	TID
13	3

- All active transactions in table
- **lastLSN**: most recent log record written by that transaction

Dirty Page Table

dirtyPgTable

pgNo	recLSN
D	8
B	10
A	11
E	13

Dirty pages are periodically flushed to disk by a background process (flushes are not logged)
On flush, remove from dirtyPageTable

- One entry for each page that has been modified but not flushed to disk
- recLSN: log record that first dirtied the page

Checkpoints

- Taken periodically
- Record the state of the dirty page table and transaction table
  - Doesn't require pages to be flushed to disk during checkpoint
- Allow us to limit amount of log we have to keep and replay during crash

Crash Recovery

- 3 Phases
  - Analysis
    - Rebuild data structures
    - Determine winners & losers
  - Redo
    - “Repeat history”
    - Why?
  - Undo
    - Undo Losers

Redo

- Where to begin?
  - Checkpoint?
  - Min(recLSN)! – earliest unflushed update
- What to REDO
  - Everything?
    - Slow
    - Problematic if using operational (escrow) logging
  - Redo an update UNLESS:
    - Page is not in dirtyPgTable
      - Page flushed prior to checkpoint, didn't redirty
    - LSN < recLSN
      - Page flushed & redirtied prior to checkpoint
    - LSN <= pageLSN
      - Page flushed after checkpoint

Only step that requires going to disk

dirtyPgTable

pgNo	recLSN
A	$\boxed{2}$
B	3
C	6
D	8
E	13

Disk

Page	pageLSN
A	2
B	3
C	6
D	?
E	?

Compensation Log Records (CLRs)

- CLR record written after each UNDO
- Avoid repeating UNDO work
- Why?
  - Because UNDO Is logical, and we don't check if records have already been UNDONE. Could get into trouble if re-undid some logical operation.

UNDO with CLR

LSN	Type	Tid	PrevLSN	Data
5	SOT	3		
6	UP	1	3	C
7	SOT	2		
8	UP	2	7	D
9	EOT	1	6	
10	UP	3	5	B
11	UP	2	8	A
12	EOT	2	11	
13	UP	3	10	E
14	CLR	3	13	E, 10
15	CLR	3	14	B, 5
16	EOT	3	15	

REDO with CLR

- REDO CLRs on crash recovery
  - Use REDO rules to check if updates in CLRs have already been done
    - Avoids repeating operational (escrow) operations
  - After processing CLR, update lastLSN field in dirtyPgTable
    - Allows UNDO to start from the right place, should we checkpoint while UNDOing

Distributed & Parallel Databases

- Same semantics as a single-node ACID SQL database, but on multiple cores/machines

- Parallel databases are more about performance

- Distributed databases must deal with failures

Implementation

Architecture:

- Shared-memory
- Shared-disk
- Shared-nothing

Parallelism:

- Pipeline
- Partitioning
- Replication

Types of partitioning

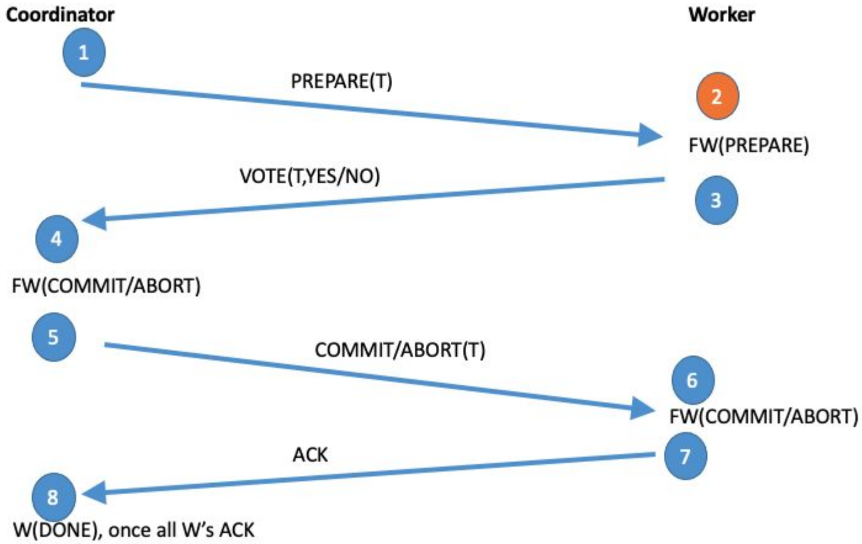
- Round-robin
  - Perfect load-balancing (data-wise)
  - Often all nodes need to participate in a query
- Hash
  - Pretty good load balancing
  - Bad at range / localized queries
- Range
  - Good at range / localized queries
  - Bad at load-balancing

2-phase commit

- Important to ensure ACID across failure domains
- Often considered a bottleneck

2-Phase Commit

1. Log start of transaction
2. Execute transaction on worker nodes
3. PREPARE each worker
4. Log transaction commit if all OK
5. Commit each worker
6. Log Done



Modern Transactions

- 2PC is slow
- Can leverage partitioning so we don’t 2PC for all transactions
- Can deterministically execute transactions to avoid 2PC in execution
- Can use epochs to trade-off latency and throughput

CAP theorem

- In general, trade-off between availability and consistency
- ACID has strong consistency but will appear down if machines go down or network becomes partitioned
- Many systems choose availability over consistency (e.g. NoSQL)

Dynamo

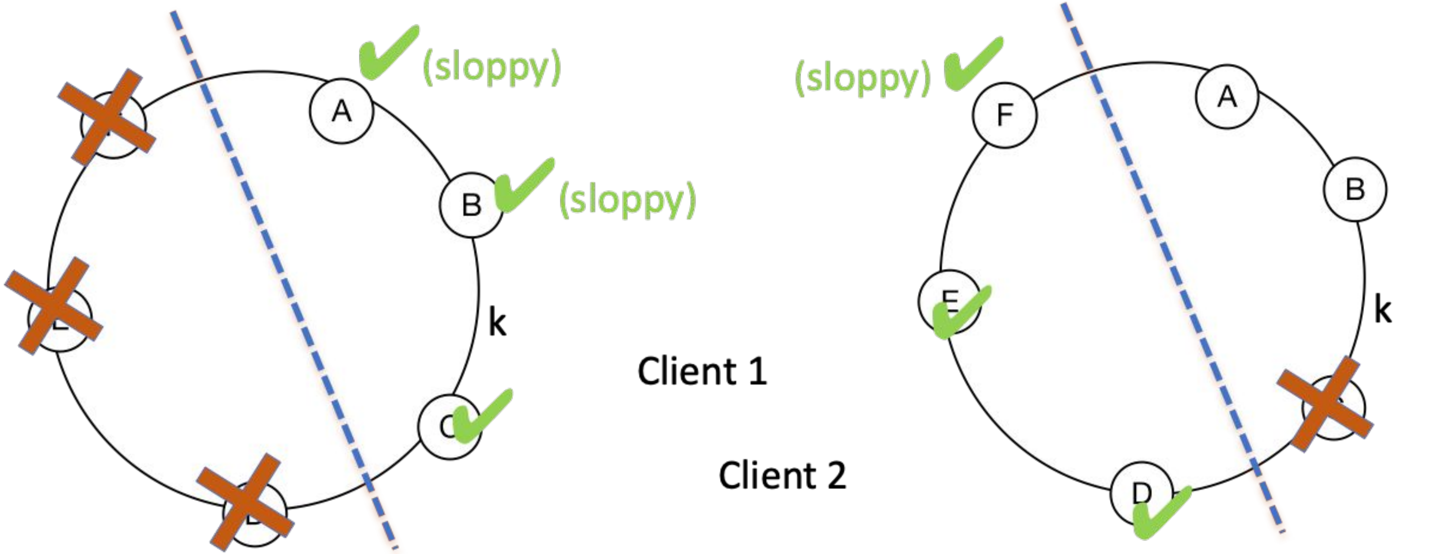
- Availability
- Partitioning
  - for scaling
  - consistent hashing
- Replication
  - for fault tolerance and performance
  - ‘N’ successors in the ring stores the key

Dynamo

- Handling reads/writes
- Quorums - R reads, W writes
  - Full: consistent ($R + W > N$)
  - Sloppy: availability
- Vector clocks for detecting conflicting writes
- Hinted handoff, Read repair, Anti-entropy

Vector Clock

- Sloppy quorums can lead to divergence



Two different versions of key k, k1 and k2 now exist

Vector Clock

- Use vector clocks to check for divergence
- Maintain a clock for each data item

Vector Clock

- Each node maintains a version counter that increments for every write it coordinates
- Maintain a clock for each data item
  - Vector - each entry corresponds to the most recent version from each coordinator
  - Clock at node i, V = V[1], V[2] ... V[i] … V[N]

Vector Clock

- Write - coordinator increments its version and sends to all nodes in the quorum
- Clock at coordinator i before write V = V[1], V[2] ... V[i] … V[N]
- Update clock to V = V[1], V[2] ... V[i] + 1 … V[N]
- Send V to write quorum
- Receiver will update V to max of what it gets
- V = max (V, V_recv)

Vector Clock

- Read - Read from the quorums
- V1, V2, V3 - If one of these, say V1, is greater than the others for every component, V1 is the latest value and we can reconcile based on vector clocks
- What if they are incomparable?
    - V1 = [1, 1], V2 = [2, 0]
    - Application-specific reconciliation

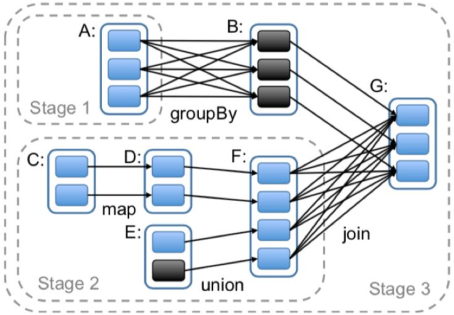
Vector Clock

- Run Sam’s example

MapReduce

- Paradigm for distributed programs
- Map and Reduce functions
    - Word count example
- Fault-tolerance
    - What if a worker dies?
    - Stragglers?
- Pros and Cons

Spark



- Distributed "dataflow" language
- Programs operate on partitions of data in parallel
- Resilient Distributed Datasets (RDDs)
  - Read only partitioned collection of records
- Lineage graph - transformations used to build the dataset
- Scheduling done in stages, with stage boundaries between *wide dependencies*
- Checkpointing and Failures
  - Lineage graph tracks dependencies, enables recovery via replay (instead of writing all intermediate data to disk)
  - Smart about when to persist to disk

Questions?