Crashes and Recovery

Write-ahead logging

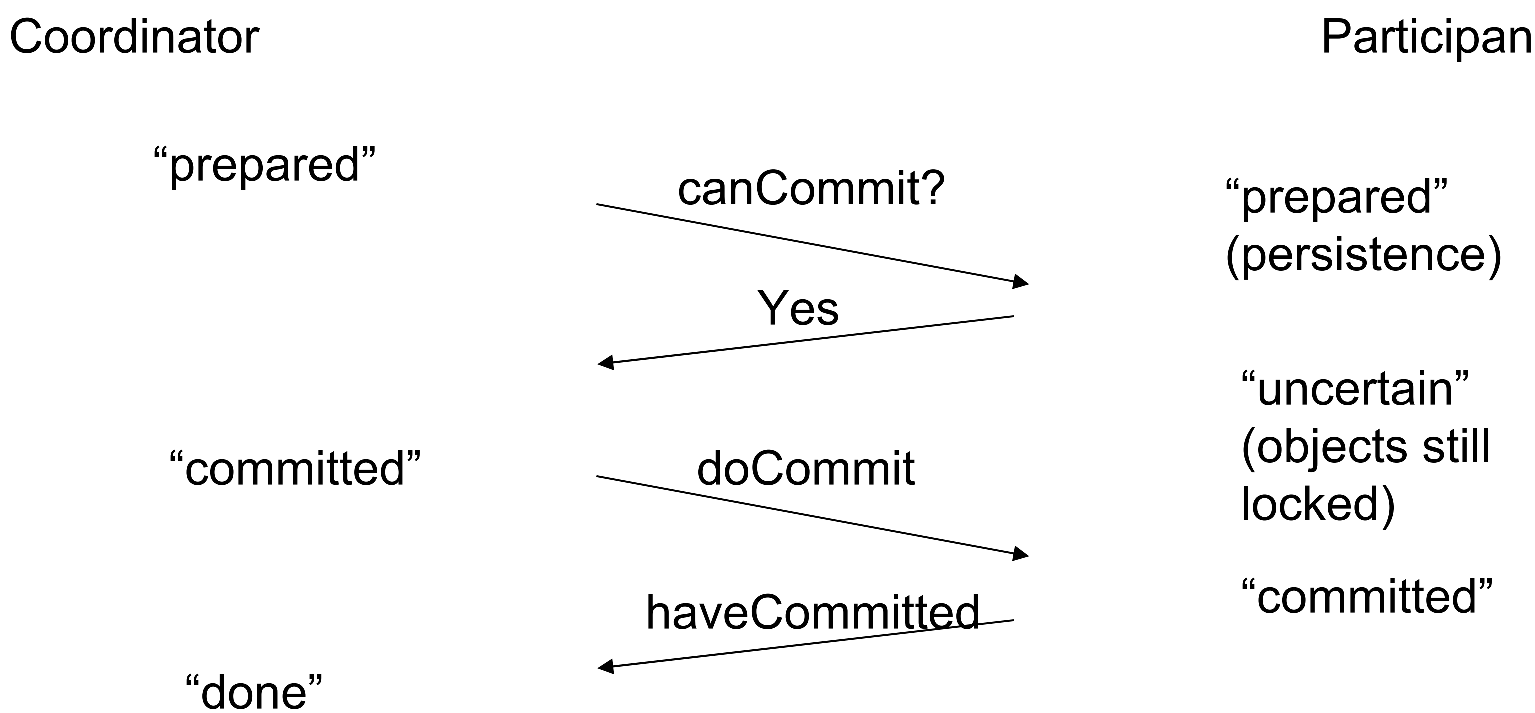
Announcements

- Exams back at the end of class
- Project 2, part 1 grades
  - tags/part1/grades.txt

Last time

- Transactions and distributed transactions
  – The ACID properties
- Isolation with 2-phase locking
  – Needed an atomic commit step, at the end
- 2-phase commit
  – voting phase
  – commit phase

2-phase commit



participant not allowed to cause an abort after it says it canCommit

Failure model

- Network is unreliable
- Servers can fail
  - But their disks don’t fail
  - Can recover state

Today: Crashes and recovery

- Goals: Recover state after crash
  - Committed transactions are not lost
  - Non-committed transactions either continued or aborted
  - Low overhead
- Plan:
  - Consider recovery of local system
  - Then consider role in distributed systems

Write-ahead logging / Journaling

- Keep a separate log of all operations
  - Transaction begin, commit, abort
  - All updates
- A transaction's operations are provisional until commit is logged to disk
  - The log records the consistent state of the system
  - Disk writes of single pages are usually atomic

`begin`/`commit`/`abort` records

- Log Sequence Number (LSN)
  - Usually implicit, the address of the first-byte of the log entry
- LSN of previous record for transaction
  - Linked list of log records for each transaction
- Transaction ID
- Operation type

`update` records

- Need all information to undo and redo the update
  - prevLSN + xID + opType as before
  - The update itself, e.g.:
    - the update location (usually pageID, offset, length)
    - old-value
    - new-value

```
xId = begin();   // suppose xId <- 42
src.bal -= 20;
dest.bal += 20;
commit(xId);
```

Log:

Disk:

Page cache:

10

11

src.bal: 100

14

dest.bal: 3

Transaction table:

Dirty page table:

```
→ xId = begin();   // suppose xId <- 42
  src.bal -= 20;
  dest.bal += 20;
  commit(xId);
```

Log:

780

prevLSN: 0
xId: 42
type: begin

Disk:

10
11
src.bal: 100

14
dest.bal: 3

Page cache:

Transaction table:

42: prevLSN = 780

Dirty page table:

```
xId = begin();    // suppose xId <- 42
src.bal -= 20;
dest.bal += 20;
commit(xId);
```

Disk:

10 11

src.bal: 100

14

dest.bal: 3

Page cache:

11

src.bal: 80

Log:

LSN		
780	prevLSN:	0
	xId:	42
	type:	begin
	⋮	
860	prevLSN:	780
	xId:	42
	type:	update
	page:	11
	offset:	10
	length:	4
	old-val:	100
	new-val:	80

$\left.\begin{array}{l}\text{page: 11}\\ \text{offset: 10}\\ \text{length: 4}\end{array}\right\}$ src.bal

Transaction table:

42: prevLSN = 860

Dirty page table:

11: firstLSN = 860, lastLSN = 860

```
    xId = begin();   // suppose xId <- 42
    src.bal -= 20;
--> dest.bal += 20;
    commit(xId);
```

Disk:

11 (offset 10)
src.bal: 100

14
dest.bal: 3

Page cache:

11
src.bal: 80

14
dest.bal: 23

Transaction table:

42: prevLSN = 902

Dirty page table:

11: firstLSN = 860, lastLSN = 860

14: firstLSN = 902, lastLSN = 902

Log:

780

prevLSN:	0
xId:	42
type:	begin

:

860

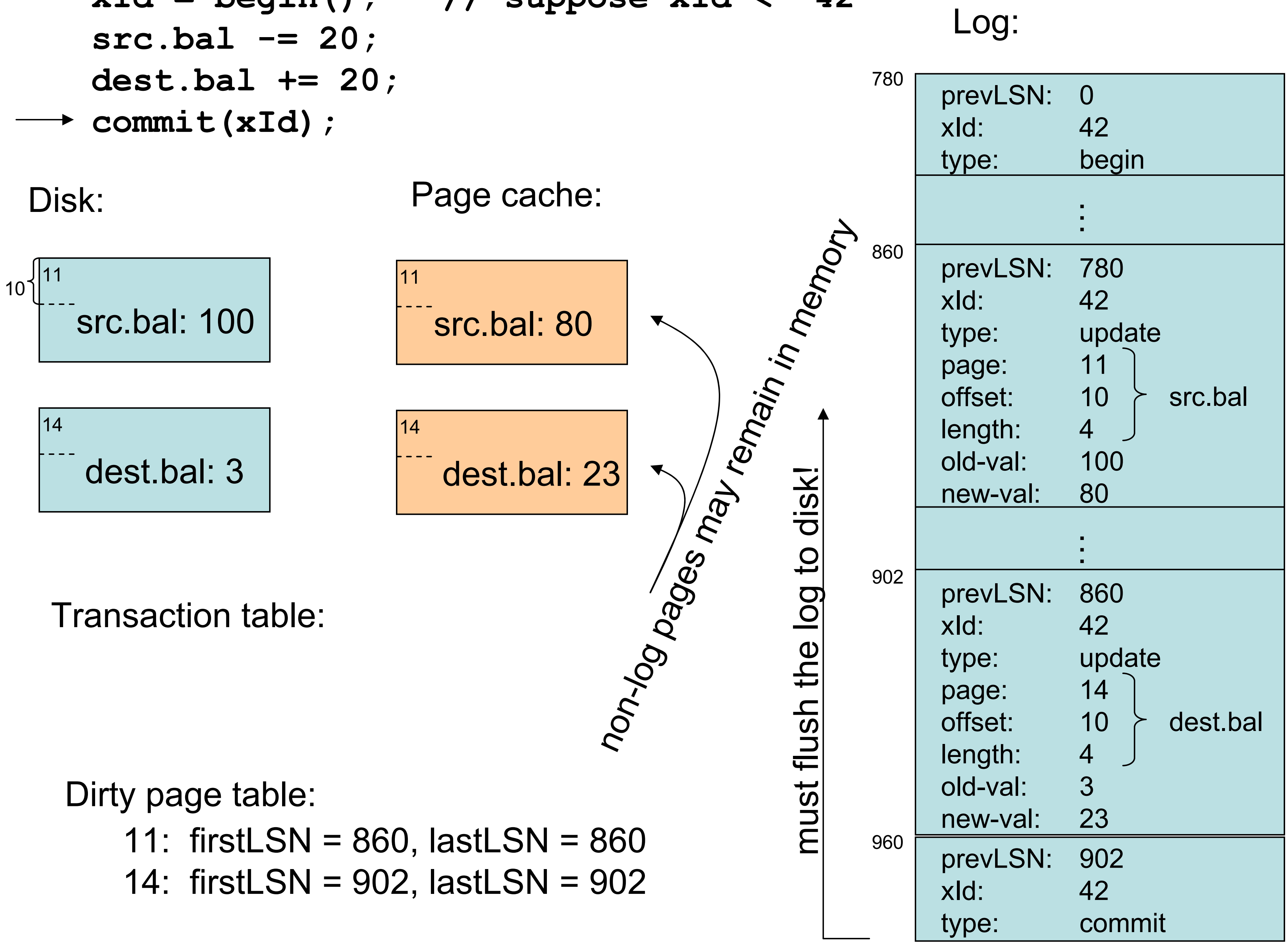
prevLSN:	780	
xId:	42	
type:	update	
page:	11	src.bal
offset:	10	src.bal
length:	4	src.bal
old-val:	100	
new-val:	80	

:

902

prevLSN:	860	
xId:	42	
type:	update	
page:	14	dest.bal
offset:	10	dest.bal
length:	4	dest.bal
old-val:	3	
new-val:	23	

```
xId = begin();    // suppose xId <- 42
src.bal -= 20;
dest.bal += 20;
commit(xId);
```



The tail of the log

- The tail of the log can be kept in memory until a transaction commits
  - …or a buffer page is flushed to disk

Recovering from simple failures

- e.g., system crash
  - For now, assume we can read the log
- "Analyze" the log
- Redo all (usually) transactions (forward)
  - Repeating history!
  - Use new-value in byte-level update records
- Undo uncommitted transactions (backward)
  - Use old-value in byte-level update records

Why redo all operations?

- (Even the loser transactions)
- Interaction with concurrency control
  - Bring system back to a former state
- Generalizes to logical operations
  - Any operation with undo and redo operations
  - Can be much faster than byte-level logging

The performance of WAL

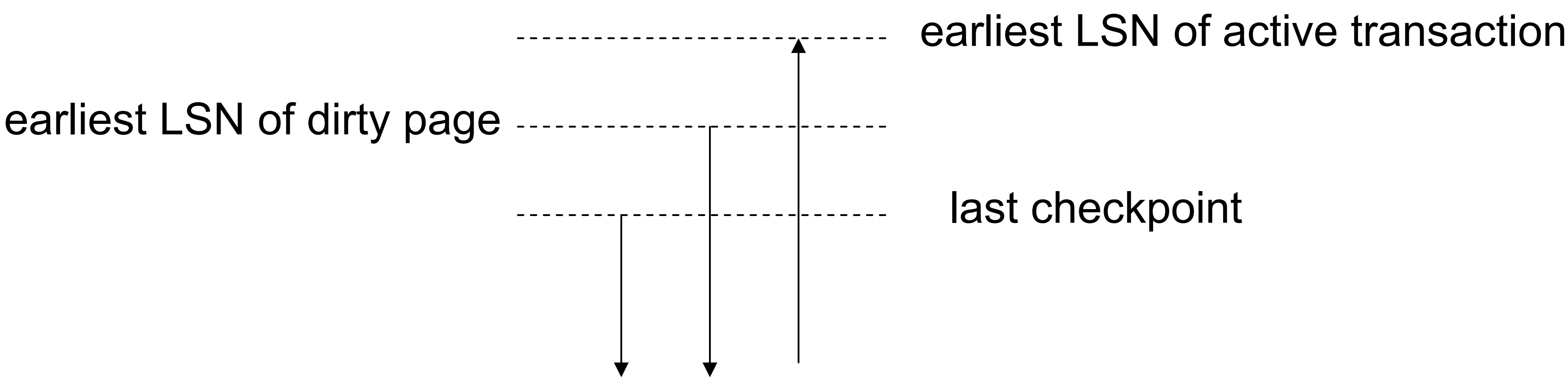
- Problems:
  - Must write disk twice?
    - Not always
  - For byte-level update logging, must know old value for the update record
- Writing the log is sequential
  - Might actually improve performance
    - Can acknowledge a write/commit as soon as the log is written

Improvements to this WAL

- Store LSN of last write on each data page
  - Can avoid unnecessary redoes
- Log checkpoint records
  - Flush buffer cache? Record which pages are in memory?
- Log recovery actions (CLR)
  - Speeds up recovery from repeated failures
- Ordered / metadata-only logging
  - Avoids needing to save old-value of files

Checkpoint records

- Can start analysis with last checkpoint
- Records:
  - Table of active transactions
  - Table of dirty pages in memory
    - And the earliest LSN that might have affected them



Recovering 2-phase commit

- Easy: just log the state-changes
  - Participants:
    - prepared, uncertain, committed/aborted
  - Coordinator:
    - prepared, committed/aborted, done
  - The messages are idempotent!
    - In recovery, resend whatever message was next
    - If coordinator and uncommitted: doAbort

What about other failures?

- What if the log fails?
  - Log and data on different disks?
  - Mirror the log?
- What if the machine room floods?
  - Mirror the log elsewhere

End-to-end solutions?

- WAL can recover the state of a crashed server
  - But we are also building toward end-to-end solutions to handle failures
- Desirable: fault-tolerance
- Redundancy/Replication!
  - Semantics of updating very complicated
    - Consensus, consistency, etc
  - Hard to achieve transparency