Recovery and Logging

Harvard Extension School

Cody Doucette, Ph.D.

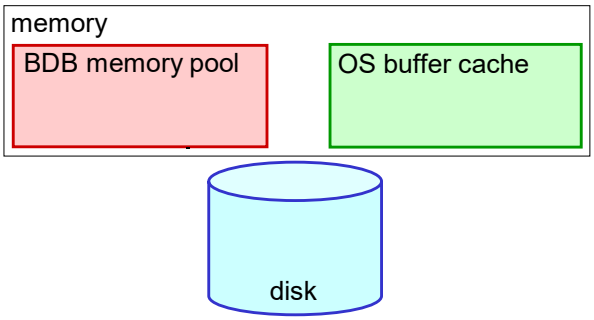
Lecture designed by David G. Sullivan

Review: ACID Properties

- A transaction has the following "ACID" properties:
  - Atomicity: either all of its changes take effect or none do
  - Consistency preservation: its operations take the database from one consistent state to another
  - Isolation: it is not affected by and does not affect other concurrent transactions
  - Durability: once it completes, its changes survive failures
- We'll now look at how the DBMS guarantees atomicity and durability.
  - ensured by the subsystem responsible for *recovery*

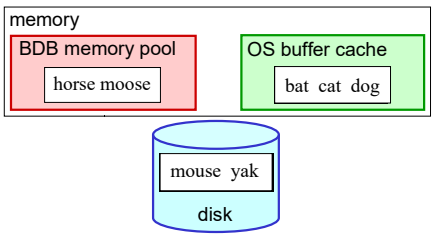
A Quick Look at Caching

- Recently accessed database pages are *cached* in memory so that subsequent accesses to them don't require disk I/O.
- There may be more than one cache:
  - the DBMS's own cache (called the memory pool in BDB)
  - the operating system's buffer cache



Caching Example 1

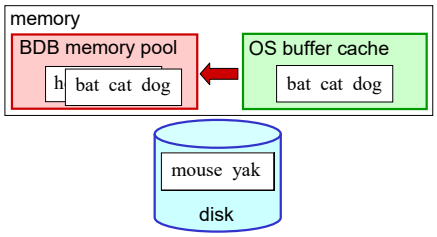
- The user requests the item with the key "horse."



- The page containing "horse" is already in the database's own cache, so no disk I/O is needed.

Caching Example 2

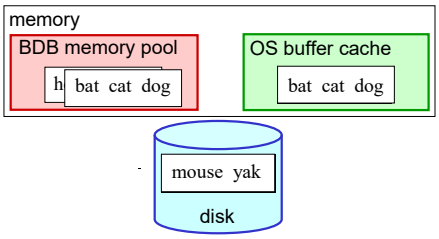
- The user requests the item with the key "cat."



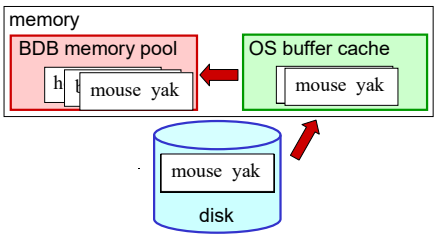
- The page containing "cat" is in the OS buffer cache, so it just needs to be brought into the database's cache. No disk I/O.
- This produces *double buffering* – two copies of the same page in memory.
  - one reason that some DBMSs bypass the filesystem

Caching Example 3

- The user requests the item with the key "yak."

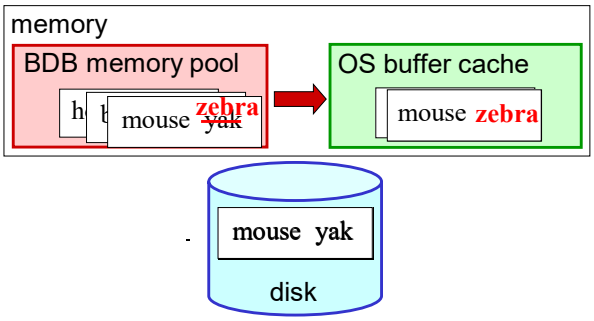


- The page with "yak" is in neither cache, so it is:
  - read from disk into the buffer cache
  - read into the database's own cache



Caching and Disk Writes

- Updates to a page may not make it to disk until the page is evicted from all of the caches.
  - initially, only the page in the DBMS's cache is updated
  - when evicted from the DBMS's cache, it is written to the backing file, but it may not go to disk right away



- This complicates recovery, because changes may not be on disk.

What Is Recovery?

- Recovery is performed after:
  - a crash of the DBMS
  - other non-catastrophic failures (e.g., a reboot)
  - (for catastrophic failures, need an archive or replication)
- It makes everything right again.
  - allows the rest of the DBMS to be built as if failures don't occur
- "the scariest code you'll ever write" (Margo Seltzer)
  - it has to work
  - it's rarely executed
  - it can be difficult to test

What Is Recovery? (cont.)

- During recovery, the DBMS takes the steps needed to:
  - *redo* changes made by any committed txn, if there's a chance the changes didn't make it to disk
    - ➔ durability: the txn's changes are still there after the crash
    - ➔ atomicity: *all* of its changes take effect
  - *undo* changes made by any txn that didn't commit, if there's a chance the changes made it to disk
    - ➔ atomicity: *none* of its changes take effect
    - also used when a transaction is rolled back
- In order for recovery to work, need to maintain enough state about txns to be able to redo or undo them.

Logging

- The *log* is a file that stores the info. needed for recovery.
- It contains:
  - update records, each of which summarizes a write
  - records for transaction begin and commit
- It does *not* record reads.
  - don't affect the state of the database
  - aren't relevant to recovery
- The log is append-only: records are added at the end, and blocks of the log file are written to disk sequentially.
  - more efficient than non-sequential writes to the database files

LSN	*record contents*
100	txn: 1; BEGIN
150	txn: 1; item: D1; old: 3000; new: 2500
225	txn: 1; item: D2; old: 1000; new: 1500
350	txn: 2; BEGIN
400	txn: 2; item: D3; old: 7200; new: 6780
470	txn: 1; item: D1; old: 2500; new: 2750
550	txn: 1; COMMIT
585	txn: 2; item: D2; old: 1500; new: 1300
675	txn: 2; item: D3; old: 6780; new: 6760

Write-Ahead Logging (WAL)

- Both updated database pages and log records are cached.
- It's important that they go to disk in a specific order.
- Example of what can go wrong:

```
read balance1
write(balance1 - 500)
read balance2
write(balance2 + 500)
CRASH
```

  - assume that:
    - `write(balance1 - 500)` made it to disk
    - `write(balance2 + 500)` didn't make it to disk
    - neither of the corresponding log records made it to disk
  - the database is in an inconsistent state
  - without the log records, the recovery system can't restore it

Write-Ahead Logging (WAL) (cont.)

- The write-ahead logging (WAL) policy:

  before a modified database page is written to disk,
  all update log records describing changes on that page
  must be forced to disk

  - the log records are "written ahead" of the database page
- This ensures that the recovery system can restore the database to a consistent state.

Undo-Redo Logging

- Update log records must include both the old *and* new values of the changed data element.
- Example log after a crash:
  - the database could be in an inconsistent state
  - why?
    - some of T1's changes may not have made it to disk.
      need to redo
    - some of T2's changes may have made it to disk.
      need to undo

LSN	*record contents*
100	txn: 1; BEGIN
150	txn: 1; item: D1; old: 3000; new: 2500
225	txn: 1; item: D2; old: 1000; new: 1500
350	txn: 2; BEGIN
400	txn: 2; item: D3; old: 7200; new: 6780
470	txn: 1; item: D1; old: 2500; new: 2750
500	txn: 1; item: D2; old: 1500; new: 2100
550	txn: 1; COMMIT
585	txn: 2; item: D2; old: 1500; new: 1300
675	txn: 2; item: D3; old: 6780; new: 6760

Undo-Redo Logging (cont.)

- To ensure that it can undo/redo txns as needed, undo-redo logging follows the WAL policy.
- In addition, it does the following when a transaction commits:
  1. writes the commit log record to the in-memory log buffer
  2. forces to disk all dirty log records
     (dirty = not yet written to disk)
- It does *not* force the dirty database pages to disk.
- At recovery, it performs two passes:
  - first, a *backward pass* to undo uncommitted transactions
  - then, a *forward pass* to redo committed transactions

Recovery Using Undo-Redo Logging

- Backward pass: begin at the last log record and scan backward
  - for each commit record, add the txn to a *commit list*
  - for each update by a txn *not* on the commit list, undo the update (restoring the old value)
  - for now, we skip:
    - updates by txns that *are* on the commit list
    - all begin records
- Forward pass:
  - for each update by a txn that *is* on the commit list, redo the update (writing the new value)
  - skip updates by txns that are *not* on the commit list, because they were handled on the backward pass
  - skip other records as well

Recovery Using Undo-Redo Logging (cont.)

- Here's how it would work on our earlier example:

LSN	*record contents*	*backward pass*	*forward pass*
100	txn: 1; BEGIN	skip	skip
150	txn: 1; item: D1; old: 3000; new: 2500	skip	redo: D1 = 2500
225	txn: 1; item: D2; old: 1000; new: 1500	skip	redo: D2 = 1500
350	txn: 2; BEGIN	skip	skip
400	txn: 2; item: D3; old: 7200; new: 6780	undo: D3 = 7200	skip
470	txn: 1; item: D1; old: 2500; new: 2750	skip	redo: D1 = 2750
500	txn: 1; item: D2; old: 1500; new: 2100	skip	redo: D2 = 2100
550	txn: 1; COMMIT	add to commit list	skip
585	txn: 2; item: D2; old: 1500; new: 1300	undo: D2 = 1500	skip
675	txn: 2; item: D3; old: 6780; new: 6760	undo: D3 = 6780	skip

- Recovery restores the database to a consistent state that reflects:
  - *all* of the updates by txn 1 (which committed before the crash)
  - *none* of the updates by txn 2 (which did *not* commit)

The Details Matter!

LSN	*record contents*	*backward pass*	*forward pass*
100	txn: 1; BEGIN	skip	skip
150	txn: 1; item: D1; old: 3000; new: 2500	skip	redo: D1 = 2500
225	txn: 1; item: D2; old: 1000; new: 1500	skip	redo: D2 = 1500
350	txn: 2; BEGIN	skip	skip
400	txn: 2; item: D3; old: 7200; new: 6780	undo: D3 = 7200	skip
470	txn: 1; item: D1; old: 2500; new: 2750	skip	redo: D1 = 2750
500	txn: 1; item: D2; old: 1500; new: 2100	skip	redo: D2 = 2100
550	txn: 1; COMMIT	add to commit list	skip
585	txn: 2; item: D2; old: 1500; new: 1300	undo: D2 = 1500	skip
675	txn: 2; item: D3; old: 6780; new: 6760	undo: D3 = 6780	skip

1) Scanning backward at the start of recovery provides the info needed for undo / redo decisions.
   - when we see an update, we already know whether the txn has committed!

The Details Matter!

LSN	*record contents*	*backward pass*	*forward pass*
100	txn: 1; BEGIN	skip	skip
150	txn: 1; item: D1; old: 3000; new: 2500	skip	**redo: D1 = 2500**
225	txn: 1; item: D2; old: 1000; new: 1500	skip	redo: D2 = 1500
350	txn: 2; BEGIN	skip	skip
400	txn: 2; item: D3; old: 7200; new: 6780	undo: D3 = 7200	skip
470	txn: 1; item: D1; old: 2500; new: 2750	skip	**redo: D1 = 2750**
500	txn: 1; item: D2; old: 1500; new: 2100	skip	redo: D2 = 2100
550	txn: 1; COMMIT	add to commit list	skip
585	txn: 2; item: D2; old: 1500; new: 1300	undo: D2 = 1500	skip
675	txn: 2; item: D3; old: 6780; new: 6760	undo: D3 = 6780	skip

2) To ensure the correct values are on disk after recovery, we:
   - put all redos after all undos (consider D2 above)
   - perform the undos in reverse order (consider D3 above)
   - perform the redos in the same order as the original updates (consider D1 above)

Logical Logging

- We've assumed that update records store the old + new values of the changed data element.
- It's also possible to use *logical logging*, which stores a logical description of the update operation.
  - example: increment D1 by 1
- Logical logging is especially useful when we use pages or blocks as data elements, rather than records.
  - storing the old and new contents of a page or block would take up a lot of space
  - instead, store a logical description
    - for example: "add record r somewhere on D1"

Logical Logging (cont.)

- When we store old and new data values, the associated undo/redo operations are *idempotent* .
  - can be performed multiple times without changing the result
- Problem: logical update operations *may not be* idempotent.
  - example: if "increment D1 by 1" has already been performed, we don't want to redo it
  - example: if "increment D1 by 1" has not been performed, we don't want to undo it
  - example: if "add record r to page D1" has already been performed, we don't want to redo it
- To ensure that only the necessary undo/redos are made, the DBMS makes use of the *log sequence numbers* (*LSNs*) associated with the update log records.

Storing LSNs with Data Elements

- When a data element is updated, the DBMS:
  - stores the LSN of the update log record with the data element
    - known as the *datum LSN*
  - stores the old LSN of the data element in the log record

log file

LSN	*record contents*
100	txn: 1; BEGIN
150	txn: 1; item: D1; **new: "bar"**; **old: "foo"; olsn: 0**

data elements (value / datum LSN)

D1	*D2*	*D3*
"foo" / 0	"oh" / 0	"moo" / 0
"bar"/ 150		

Recovery Using LSNs

- During recovery, there are three LSNs to consider for each update record:
  1) the ***record LSN****:* the one for the update record itself

LSN	*record contents*
700	txn: 3; BEGIN
770	txn: 3; item: D5; old: "foo"; new: "bar"; olsn: 0
825	txn: 4; BEGIN
850	txn: 4; item: D4; old: 9000; new: 8500; olsn: 0
900	txn: 4; item: D6; old: 5.7; new: 8.9; olsn: 0
930	txn: 3; item: D7; old: "zoo"; new: "cat"; olsn: 0
980	txn: 4; COMMIT
1000	txn: 3; item: D4; old: 8500; new: 7300; olsn: 850
1100	txn: 3; item: D6; old: 8.9; new: 4.1; olsn: 900

Recovery Using LSNs

- During recovery, there are three LSNs to consider for each update record:
  1) the ***record LSN****:* the one for the update record itself
  2) the on-disk ***datum LSN*** for the data item
     - the one associated with it in the database file

on-disk datum LSNs:
D4: 0, D5: 0, **D6**: **1100**, D7: 930

LSN	*record contents*
700	txn: 3; BEGIN
770	txn: 3; item: D5; old: "foo"; new: "bar"; olsn: 0
825	txn: 4; BEGIN
850	txn: 4; item: D4; old: 9000; new: 8500; olsn: 0
900	txn: 4; item: D6; old: 5.7; new: 8.9; olsn: 0
930	txn: 3; item: D7; old: "zoo"; new: "cat"; olsn: 0
980	txn: 4; COMMIT
1000	txn: 3; item: D4; old: 8500; new: 7300; olsn: 850
1100	txn: 3; item: **D6**; old: 8.9; new: 4.1; olsn: 900

Recovery Using LSNs

- During recovery, there are three LSNs to consider for each update record:
  1) the ***record LSN****:* the one for the update record itself
  2) the on-disk ***datum LSN*** for the data item
     - the one associated with it in the database file
  3) the ***olsn***: the old datum LSN for the data item
     - the one associated with it when the update was originally requested

on-disk datum LSNs:
D4: 0, D5: 0, **D6**: **1100**, D7: 930

LSN	*record contents*
700	txn: 3; BEGIN
770	txn: 3; item: D5; old: "foo"; new: "bar"; olsn: 0
825	txn: 4; BEGIN
850	txn: 4; item: D4; old: 9000; new: 8500; olsn: 0
900	txn: 4; item: D6; old: 5.7; new: 8.9; olsn: 0
930	txn: 3; item: D7; old: "zoo"; new: "cat"; olsn: 0
980	txn: 4; COMMIT
1000	txn: 3; item: D4; old: 8500; new: 7300; olsn: 850
1100	txn: 3; item: **D6**; old: 8.9; new: 4.1; olsn: **900**

The Backward Pass Using LSNs

- During the backward pass, we undo an update if:
  - the txn did *not* commit
  - **datum LSN** == **record LSN**
- When we undo, we also set: datum LSN = olsn

on-disk datum LSNs:
D4: 0, D5: 0, **D6**: **1100**, D7: 930

LSN	*record contents*
700	txn: 3; BEGIN
770	txn: 3; item: D5; old: "foo"; new: "bar"; olsn: 0
825	txn: 4; BEGIN
850	txn: 4; item: D4; old: 9000; new: 8500; olsn: 0
900	txn: 4; item: D6; old: 5.7; new: 8.9; olsn: 0
930	txn: 3; item: D7; old: "zoo"; new: "cat"; olsn: 0
980	txn: 4; COMMIT
1000	txn: 3; item: D4; old: 8500; new: 7300; olsn: 850
1100	txn: 3; item: **D6**; old: 8.9; new: 4.1; olsn: **900**

Which updates will be undone?

- datum LSNs: D4: 0 D5: 0 D6: 1100 D7: 930

LSN	*record contents*	*backward pass*	*forward pass*
700	txn: 3; BEGIN		
770	txn: 3; item: D5; old: "foo"; new: "bar"; olsn: 0		
825	txn: 4; BEGIN		
850	txn: 4; item: D4; old: 9000; new: 8500; olsn: 0		
900	txn: 4; item: D6; old: 5.7; new: 8.9; olsn: 0		
930	txn: 3; item: D7; old: "zoo"; new: "cat"; olsn: 0		
980	txn: 4; COMMIT		
1000	txn: 3; item: D4; old: 8500; new: 7300; olsn: 850		
1100	txn: 3; item: D6; old: 8.9; new: 4.1; olsn: 900		

Which updates will be undone?

- datum LSNs: D4: 0 D5: 0 D6: ~~1100~~, 900 D7: ~~930~~, 0

LSN	*record contents*	*backward pass*	*forward pass*
700	txn: 3; BEGIN	skip	
770	txn: 3; item: D5; old: "foo"; new: "bar"; olsn: 0	0 != 770 don't undo	
825	txn: 4; BEGIN	skip	
850	txn: 4; item: D4; old: 9000; new: 8500; olsn: 0	skip	
900	txn: 4; item: D6; old: 5.7; new: 8.9; olsn: 0	skip	
930	txn: 3; item: D7; old: "zoo"; new: "cat"; olsn: 0	930 == 930 undo: D7 = "zoo" datum LSN = 0	
980	txn: 4; COMMIT	add to commit list	
1000	txn: 3; item: D4; old: 8500; new: 7300; olsn: 850	0 != 1000 don't undo	
1100	txn: 3; item: D6; old: 8.9; new: 4.1; olsn: 900	1100 == 1100 undo: D6 = 8.9 datum LSN = 900	

The Forward Pass Using LSNs

- During the forward pass, we redo an update if:
  - the txn *did* commit
  - **datum LSN == olsn**
- When we redo, we also set:
  datum LSN = record LSN

on-disk datum LSNs:
D4: **0**, D5: 0, D6: 900, D7: 0

LSN	*record contents*
700	txn: 3; BEGIN
770	txn: 3; item: D5; old: "foo"; new: "bar"; olsn: 0
825	txn: 4; BEGIN
850	txn: 4; item: D4; old: 9000; new: 8500; olsn: **0**
900	txn: 4; item: D6; old: 5.7; new: 8.9; olsn: 0
930	txn: 3; item: D7; old: "zoo"; new: "cat"; olsn: 0
980	txn: 4; COMMIT
1000	txn: 3; item: D4; old: 8500; new: 7300; olsn: 850
1100	txn: 3; item: D6; old: 8.9; new: 4.1; olsn: 900

Which updates will be redone?

- datum LSNs: D4: 0 D5: 0 D6: ~~1100~~, 900 D7: ~~930~~, 0

LSN	*record contents*	*backward pass*	*forward pass*
700	txn: 3; BEGIN	skip	
770	txn: 3; item: D5; old: "foo"; new: "bar"; olsn: 0	0 != 770 don't undo	
825	txn: 4; BEGIN	skip	
850	txn: 4; item: D4; old: 9000; new: 8500; olsn: 0	skip	
900	txn: 4; item: D6; old: 5.7; new: 8.9; olsn: 0	skip	
930	txn: 3; item: D7; old: "zoo"; new: "cat"; olsn: 0	930 == 930 undo: D7 = "zoo" datum LSN = 0	
980	txn: 4; COMMIT	add to commit list	
1000	txn: 3; item: D4; old: 8500; new: 7300; olsn: 850	0 != 1000 don't undo	
1100	txn: 3; item: D6; old: 8.9; new: 4.1; olsn: 900	1100 == 1100 undo: D6 = 8.9 datum LSN = 900	

Which updates will be redone?

- datum LSNs: D4: ~~0~~, 850 D5: 0 D6: ~~1100~~, 900 D7: ~~930~~, 0

LSN	*record contents*	*backward pass*	*forward pass*
700	txn: 3; BEGIN	skip	skip
770	txn: 3; item: D5; old: "foo"; new: "bar"; olsn: 0	0 != 770 don't undo	skip
825	txn: 4; BEGIN	skip	skip
850	txn: 4; item: D4; old: 9000; new: 8500; olsn: 0	skip	0 == 0 redo: D4 = 8500 datum LSN = 850
900	txn: 4; item: D6; old: 5.7; new: 8.9; olsn: 0	skip	900 != 0 don't redo
930	txn: 3; item: D7; old: "zoo"; new: "cat"; olsn: 0	930 == 930 undo: D7 = "zoo" datum LSN = 0	skip
980	txn: 4; COMMIT	add to commit list	skip
1000	txn: 3; item: D4; old: 8500; new: 7300; olsn: 850	0 != 1000 don't undo	skip
1100	txn: 3; item: D6; old: 8.9; new: 4.1; olsn: 900	1100 == 1100 undo: D6 = 8.9 datum LSN = 900	skip

Checkpoints

- As a DBMS runs, the log gets longer and longer.
  - thus, recovery could end up taking a very long time!
- To avoid long recoveries, periodically perform a *checkpoint*.
  - force data and log records to disk to create a consistent on-disk database state
  - during recovery, don't need to consider operations that preceded this consistent state

Static Checkpoints

- Stop activity and wait for a consistent state.
  1) prohibit new transactions from starting and wait until all current transactions have aborted or committed.
- Once there is a consistent state:
  2) force all dirty log records to disk (dirty = not yet written to disk)
  3) force all dirty database pages to disk
  4) write a *checkpoint record* to the log
  - these steps must be performed in the specified order!
- When performing recovery, go back to the most recent checkpoint record.
- Problem with this approach?

Dynamic Checkpoints

- Don't stop and wait for a consistent state.
  Steps:
  1) prevent all update operations
  2) force all dirty log records to disk
  3) force all dirty database pages to disk
  4) write a *checkpoint record* to the log
     - include a list of all active txns
- When performing recovery:
  - backward pass: go back until you've seen the start records of all uncommitted txns in the most recent checkpoint record
  - forward pass: begin from the log record that comes after the most recent checkpoint record. why?
  - note: if all txns in the checkpoint record are on the commit list, we stop the backward pass at the checkpoint record

Example of Recovery with Dynamic Checkpoints

- Initial datum LSNs: D4: 110 D5: 140,0 D6: 80

LSN	*record contents*	*backward pass*	*forward pass*
100	txn: 1; BEGIN		
110	txn: 1; item: **D4**; old: 20; new: 15; olsn: 0		
120	txn: 2; BEGIN	***stop here***	
130	txn: 1; COMMIT	add to commit list	
140	txn: 2; item: D5; old: 12; new: 13; olsn: 0	undo: D5 = 12 datum LSN = 0	
150	**CHECKPOINT (active txns = 2)**	note active txns	
160	txn: 2; item: **D4**; old: 15; new: 50; olsn: 110	don't undo	***start here*** skip
170	txn: 3; BEGIN	skip	skip
180	txn: 3; item: D6; old: 6; new: 8; olsn: 80	don't undo	skip

Could D4 have a datum LSN of less than 110?

Undo-Only Logging

- Only store the info. needed to undo txns.
  - update records include only the old value
- Like undo-redo logging, undo-only logging follows WAL.
- In addition, all database pages changed by a transaction must be forced to disk before allowing the transaction to commit. Why?
- At transaction commit:
  1. force all dirty log records to disk
  2. force database pages changed by the txn to disk
  3. write the commit log record
  4. force the commit log record to disk
- During recovery, the system only performs the backward pass.

Redo-Only Logging

- Only store the info. needed to redo txns.
  - update records include only the new value
- Like the other two schemes, redo-only logging follows WAL.
- In addition, all database pages changed by a txn are held in memory until it commits and its commit record is forced to disk.
- At transaction commit:
  1. write the commit log record
  2. force all dirty log records to disk

(changed database pages are allowed to go to disk anytime after this)

- If a transaction aborts, none of its changes can be on disk.
- During recovery, perform the backward pass to build the commit list (no undos). Then perform the forward pass as in undo-redo.

Comparing the Three Logging Schemes

- Factors to consider in the comparison:
  - complexity/efficiency of recovery
  - size of the log files
  - what needs to happen when a txn commits
  - other restrictions that a logging scheme imposes on the system
- We'll list advantages and disadvantages of each scheme.
- Undo-only:
  + smaller logs than undo-redo
  + simple and quick recovery procedure (only one pass)
  – forces log and data to disk at commit; have to wait for the I/Os

Comparing the Three Logging Schemes (cont.)

- Redo-only:
  - \+ smaller logs than undo-redo
  - +/ – recovery: more complex than undo-only, less than undo-redo
  - – must be able to cache all changes until the txn commits
    - limits the size of transactions
    - constrains the replacement policy of the cache
  - \+ forces only log records to disk at commit
- Undo-redo:
  - – larger logs
  - – more complex recovery
  - \+ forces only log records to disk at commit
  - \+ don't need to retain all data in the cache until commit

Reviewing the Log Record Types

- Why is each type needed?
  - assume undo-redo logging
- update records: hold the info. needed to undo/redo changes
- commit records: allow us to determine which changes should be undone and which should be redone
- begin records: allow us to determine the extent of the backward pass in the presence of dynamic checkpoints
- checkpoint records: limit the amount of the log that is processed during recovery

Review Problem

- Recall the three logging schemes:
  - undo-redo, undo-only, redo-only
- *What type of logging is being used to create the log at right?*

```
txn 1
  writes 75 for D1
  writes 90 for D3
txn 2
  writes 25 for D2
  writes 60 for D3
```

LSN	*record contents*
100	txn: 1; BEGIN
210	txn: 1; item: D1; old: 45
300	txn: 2; BEGIN
420	txn: 2; item: D2; old: 80
500	txn: 2; item: D3; old: 30
525	txn: 2; COMMIT
570	txn: 1; item: D3; old: 60

Review Problem

- Recall the three logging schemes:
  - undo-redo, undo-only, redo-only
- *What type of logging is being used to create the log at right?*
  undo-only
- To make the rest of the problem easier, add the new values to the log…

```
txn 1
  writes 75 for D1
  writes 90 for D3
txn 2
  writes 25 for D2
  writes 60 for D3
```

LSN	*record contents*
100	txn: 1; BEGIN
210	txn: 1; item: D1; old: 45; **new: 75**
300	txn: 2; BEGIN
420	txn: 2; item: D2; old: 80; **new: 25**
500	txn: 2; item: D3; old: 30; **new: 60**
525	txn: 2; COMMIT
570	txn: 1; item: D3; old: 60; **new: 90**

Review Problem

- Recall the three logging schemes:
  - undo-redo, undo-only, redo-only
- At the start of recovery, what are the possible on-disk values under ***undo-only***?
  - does *not* pin values in memory
    ➔ *may* go to disk at any time
  - at commit, forces dirty data values to disk
    ➔ older values are no longer possible

```
txn 1
  writes 75 for D1
  writes 90 for D3
txn 2
  writes 25 for D2
  writes 60 for D3
```

LSN	*record contents*
100	txn: 1; BEGIN
210	txn: 1; item: D1; old: 45; new: 75
300	txn: 2; BEGIN
420	txn: 2; item: D2; old: 80; new: 25
500	txn: 2; item: D3; old: 30; new: 60
525	txn: 2; COMMIT
570	txn: 1; item: D3; old: 60; new: 90

	in-memory	possible on-disk
D1:		
D2:		
D3:		

Review Problem

- Recall the three logging schemes:
  - undo-redo, undo-only, redo-only
- At the start of recovery, what are the possible on-disk values under ***redo-only***?
  - *does* pin values in memory
    ➔ *can't* go to disk until commit
  - at commit, unpins values but does *not* force them to disk
    ➔ older values are still possible

```
txn 1
  writes 75 for D1
  writes 90 for D3
txn 2
  writes 25 for D2
  writes 60 for D3
```

LSN	*record contents*
100	txn: 1; BEGIN
210	txn: 1; item: D1; old: 45; new: 75
300	txn: 2; BEGIN
420	txn: 2; item: D2; old: 80; new: 25
500	txn: 2; item: D3; old: 30; new: 60
525	txn: 2; COMMIT
570	txn: 1; item: D3; old: 60; new: 90

	in-memory	possible on-disk
D1:		
D2:		
D3:		

Review Problem

- Recall the three logging schemes:
  - undo-redo, undo-only, redo-only
- At the start of recovery, what are the possible on-disk values under ***undo-redo***?
  - does *not* pin values in memory
    ➔ *may* go to disk at any time
  - at commit, does *not* force dirty data to disk
    ➔ older values are still possible

```
txn 1
  writes 75 for D1
  writes 90 for D3
txn 2
  writes 25 for D2
  writes 60 for D3
```

LSN	*record contents*
100	txn: 1; BEGIN
210	txn: 1; item: D1; old: 45; new: 75
300	txn: 2; BEGIN
420	txn: 2; item: D2; old: 80; new: 25
500	txn: 2; item: D3; old: 30; new: 60
525	txn: 2; COMMIT
570	txn: 1; item: D3; old: 60; new: 90

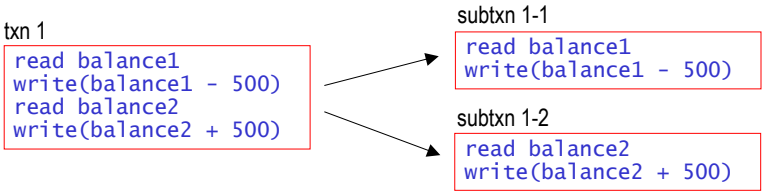
	in-memory	possible on-disk
D1:		
D2:		
D3:		

Atomicity

- In a centralized database, logging and recovery are enough to ensure atomicity.
  - if a txn's commit record makes it to the log, *all* of its changes will eventually take effect
  - if a txn's commit record isn't in the log when a crash occurs, *none* of its changes will remain after recovery
- What about atomicity in a distributed database?

Recall: Distributed Transactions

- A *distributed transaction* involves data stored at multiple sites.
- One of the sites serves as the *coordinator* of the transaction.
- The coordinator divides a distributed transaction into *subtransactions*, each of which executes on one of the sites.



Distributed Atomicity

- In a distributed database:
  - each site performs local logging and recovery of its subtxns
  - that alone is *not* enough to ensure atomicity
- The sites must coordinate to ensure that either:
  - *all* of the subtxns are committed
    or
  - *none* of them are

Distributed Atomicity (cont.)

- Example of what could go wrong:
  - a subtxn at one of the sites deadlocks and is aborted
  - before the coordinator of the txn finds out about this, it tells the other sites to commit, and they do so
- Another example:
  - the coordinator notifies the other sites that it's time to commit
  - most of the sites commit their subtxns
  - one of the sites crashes before committing

Two-Phase Commit (2PC)

- A protocol for deciding whether to commit a distributed txn.
- Basic idea:
  - coordinator asks sites if they're ready to commit
  - if a site is ready, it:
    1. *prepares* its subtxn – putting it in the *ready state*
    2. tells the coordinator it's ready
  - if all sites say they're ready, all subtxns are committed
  - otherwise, all subtxns are aborted (i.e., rolled back)
- Preparing a subtxn means ensuring it can be *either* committed or rolled back – even after a failure.
  - need to at least…
  - some logging schemes need additional steps
- After saying it's ready, a site *must wait* to be told what to do next.

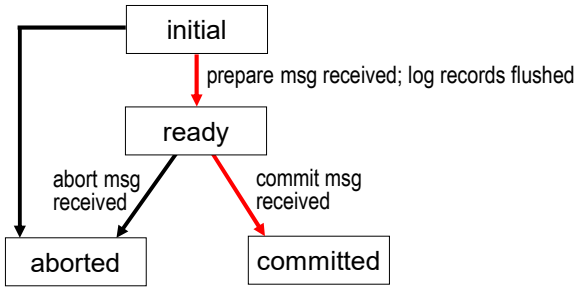
2PC Phase I: Prepare

- When it's time to commit a distributed txn T, the coordinator:
  - force-writes a *prepare record* for T to its own log
  - sends a *prepare message* to each participating site
- If a site can commit its subtxn, it:
  - takes the steps needed to put its txn in the ready state
  - force-writes a *ready record* for T to its log
  - sends a *ready message* for T to the coordinator and waits
- If a site needs to abort its subtxn, it:
  - force-writes a *do-not-commit record* for T to its log
  - sends a *do-not-commit message* for T to the coordinator
  - can it abort the subtxn now?
- Note: we always log a message before sending it to others.
  - allows the decision to send the message to survive a crash

2PC Phase II: Commit or Abort

- The coordinator reviews the messages from the sites.
  - if it doesn't hear from a site within some time interval, it assumes a do-not-commit message
- If all sites sent ready messages for T, the coordinator:
  - force-writes a commit record for T to its log
    - T is now officially committed
  - sends *commit messages* for T to the participating sites
- Otherwise, the coordinator:
  - force-writes an abort record for T to its log
  - sends *abort messages* for T to the participating sites
- Each site:
  - force-writes the appropriate record (commit or abort) to its log
  - commits or aborts its subtxn as instructed

2PC State Transitions



- A subtxn can enter the aborted state from the initial state at any time.
- After entering the ready state, it can only enter the aborted state after receiving an abort message.
- A subtxn can only enter the committed state from the ready state, and only after receiving a commit message.

Recovery When Using 2PC

- When a site recovers, its decides whether to undo or redo its subtxn for a txn T based on the last record for T in its log.
- Case 1: the last log record for T is a commit record.
  - redo the subtxn's updates as needed
- Case 2: the last log record for T is an abort record.
  - undo the subtxn's updates as needed
- Case 3: the last log record for T is a do-not-commit record.
  - undo the subtxn's updates as needed
  - why is this correct?

Recovery When Using 2PC (cont.)

- Case 4: the last log record for T is from before 2PC began (e.g., an update record).
  - undo the subtxn's updates as needed
  - this works in both of the possible situations:
    - 2PC has already completed without hearing from this site
      why?
    - 2PC is still be going on
      why?

- Case 5: the last log record for T is a ready record.
  - contact the coordinator (or another site) to determine T's fate
  - why can the site still commit or abort T as needed?

  - if it can't reach another site, it must block until it can reach one!

What if the Coordinator Fails?

- The other sites can either:
  - wait for the coordinator to recover
  - elect a new coordinator
- In the meantime, each site can determine the fate of any current distributed transactions.
- Case 1: a site has *not* received a prepare message for txn T
  - can abort its subtxn for T
  - preferable to waiting for the coordinator to recover, because it allows the T's fate to be decided
- Case 2: a site *has* received a prepare message for T, but has *not* yet sent ready message
  - can also abort its subtxn for T now. why?

What if the Coordinator Fails? (cont.)

- Case 3: a site sent a ready message for T but didn't hear back
  - poll the other sites to determine T's fate

evidence	conclusion/action
at least one site has a commit record for T	???
at least one site has an abort record for T	???
no commit/abort records for T; at least one site does *not* have a ready record for T	???
no commit/abort records for T; *all* surviving sites have ready records for T	can't know T's fate unless coordinator recovers. why?

Extra Practice

- What type of logging is being used to create the log at right?
- At the start of recovery, what are the possible on-disk values?

original values:
D1=1000, D2=3000

LSN	*record contents*
100	txn: 1; BEGIN
150	txn: 1; item: D1; new: 2500
350	txn: 2; BEGIN
400	txn: 2; item: D2; new: 6780
470	txn: 1; item: D1; new: 2750
550	txn: 1; COMMIT
585	txn: 2; item: D1; new: 1300

Extra Practice

- What if the DBMS were using ***undo-only*** logging instead?
- At the start of recovery, what are the possible on-disk values?

original values:
D1=1000, D2=3000

LSN	*record contents*
100	txn: 1; BEGIN
150	txn: 1; item: D1; new: 2500
350	txn: 2; BEGIN
400	txn: 2; item: D2; new: 6780
470	txn: 1; item: D1; new: 2750
550	txn: 1; COMMIT
585	txn: 2; item: D1; new: 1300

	in-memory	possible on-disk
D1:		1000
D2:		3000

Extra Practice

- What if the DBMS were using ***undo-redo*** logging instead?

- At the start of recovery, what are the possible on-disk values?

original values:
D1=1000, D2=3000

LSN	*record contents*
100	txn: 1; BEGIN
150	txn: 1; item: D1; new: 2500
350	txn: 2; BEGIN
400	txn: 2; item: D2; new: 6780
470	txn: 1; item: D1; new: 2750
550	txn: 1; COMMIT
585	txn: 2; item: D1; new: 1300

	in-memory	possible on-disk
D1:		1000
D2:		3000