Lecture 12

10/23/2023

OCC Recap

Granularity of Locking

Intro to Recovery

Recap: Transactions

- Group related sequence of actions so they are “all or nothing”
  - If the system crashes, partial effects are not seen
  - Other transactions do not see partial effects

- A set of implementation techniques that provides this abstraction with good performance

ACID Properties of Transactions

- A tomicity – many actions look like one; “all or nothing”
- C onsistency – database preserves invariants
- I solation – concurrent actions don’t see each other’s results
- D urability – completed actions in effect after crash (“recoverable”)

Last Time: 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) txns
  - 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

Tradeoff

- In OCC:
  - Never have to wait for locks
  - No deadlocks
- But...
  - Transactions that conflict often have to be restarted
  - Transactions can "starve" -- e.g., be repeatedly restarted, never making progress
- OCC will do better when the restart rate is low
  - I.e., when there is less contention

In-class exercise

Can you think of a scenario in which OCC is better and a scenario where 2PL is better (i.e., it has higher throughput)

Today

- Locking Granularity
- Introduction to Recovery

Locking Granularity

- So far, we've used an abstract model of "objects" being read, written, and locked, e.g.:

  RX
  WX

- In practice, “X” could be a tuple, page, table, or whole database.
  - Tradeoff between overhead and concurrency

Tradeoff

- A txn that touches many records will have to acquire many locks!
  - This adds overhead to each operation
- A txn that locks a whole table when it only needs to access part of it will limit concurrency
- Would like to allow:
  - Txns that lock a few records to use record or page locks
  - Txns that lock many records to use table locks

Multiple Granularities Complicate Locking

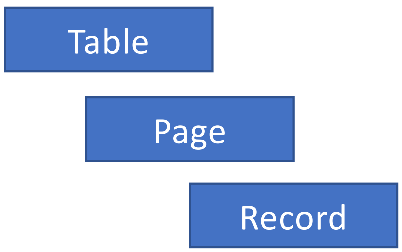
- Need to ensure different granularities co-exist
- Non-trivial, e.g.:
  - T1 shouldn't be able to lock a record in Table A if T2 has write lock on all of A
- Solutions:
  - Hierarchy of locks, e.g.

***Problem**: What if a transaction wants to modify a single record? Does it need an Xlock on the table?*

Seems to defeat the purpose of record-level locking!

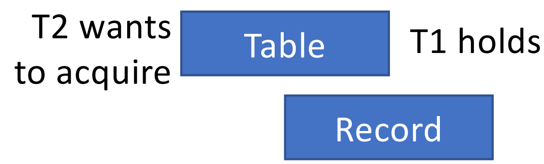
Idea: Acquire locks at higher levels before locking lower-level locks

Solution: Intention Locks



- Suppose T1 wants to write 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 (IS) or write intention (IX) locks
  - Prevent transactions from modifying the whole object when another transaction is working on a lower level
  - New *compatibility table*

Lock compatibility table



Reading / updating whole level (S, X) — *Reading / updating lower level* (IX, IS)

T2 trying to acquire \ T1 holds	**S**	**X**	**IX**	**IS**
S	Y	N	N	Y
X	N	N	N	N
IX	N	N	Y	Y
IS	Y	N	Y	Y

T2 can't read all of level if T1 *updating* lower level

T2 can read all of level if T1 *reading* lower level

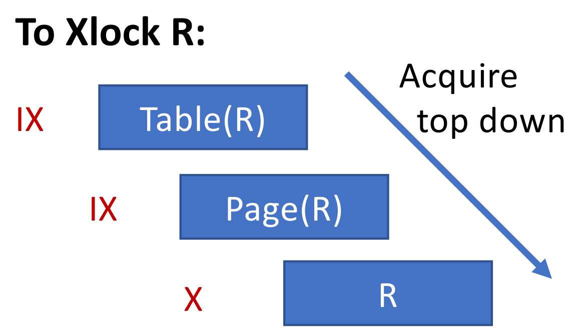
T2 can't update all of level if T1 updating lower level

T2 can't update all of level if T1 reading lower level

T2 can read / update lower level if T1 is reading / updating lower level (If they try to access same lower-level object, locking at lower level will block)

Locking Protocol with IS/IX locks

- Consider transaction T trying to S/X lock record R at level j of hierarchy
- For each level L in 1 … j - 1
  - Acquire IS/IX lock on object containing R at level L
    - Block if not compatible
- Acquire S/X lock on R



- Release in opposite order (bottom up)

https://clicker.mit.edu/6.5830/

- Given this hierarchy
- And three records on two pages of table T
- Given the use of intention locks and locking at different granularities, which of the following pairs of transactions would execute concurrently without blocking?

A) **T1**: Read P1 **T2**: Write A X

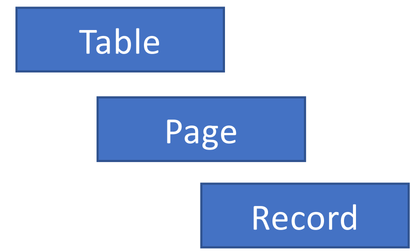
B) **T1**: Write P2 **T2**: Read T X

C) **T1**: Write P1 **T2**: Write P2 ✓

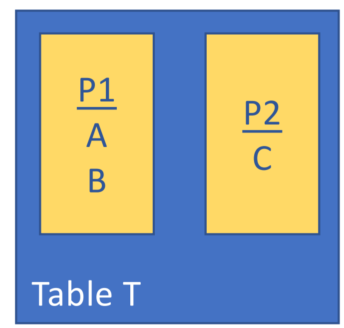
D) **T1**: Read P1 **T2**: Write C ✓

Select all correct schedules

Hierarchy



Database



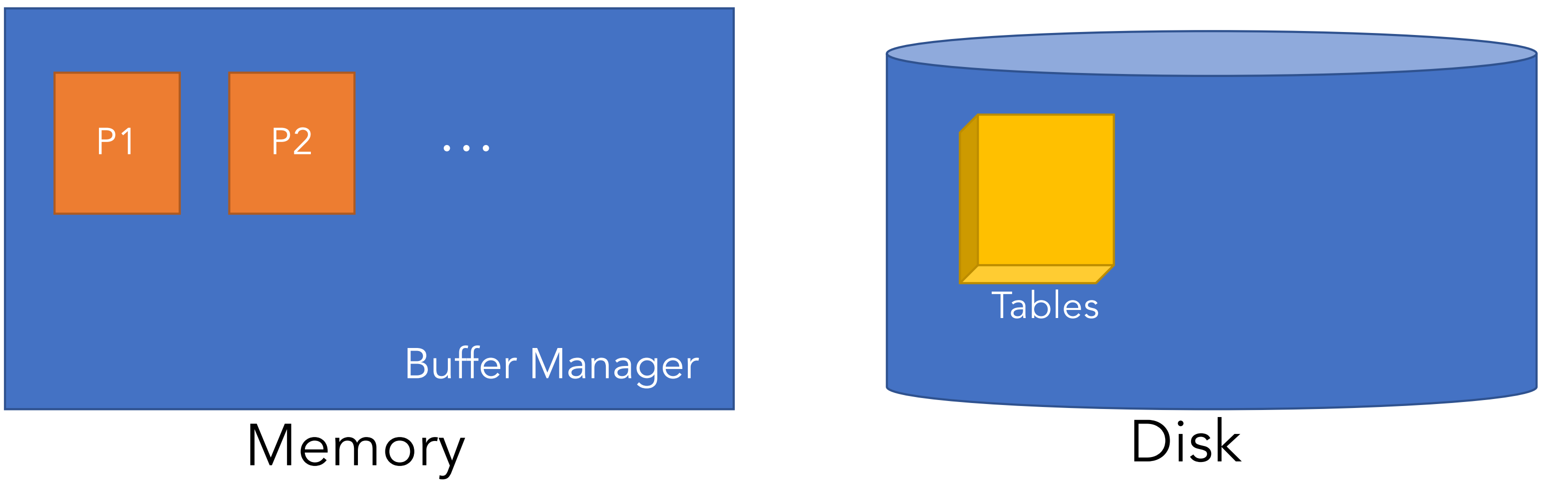
Introduction to Recovery

- What happens during crash:
  - Memory is reset
  - State on disk persists
- After a crash, recovery ensures:
  - **Atomicity:** partially finished txns are rolled back (aborted)
  - **Durability**: committed txns are on stable storage (disk)
- Brings database into a transaction consistent state, where committed transactions are fully reflected, and uncommitted transactions are completely undone

Why Rollback Uncommitted Transactions?

- After system crashes, all client connections gone
- Not generally possible to figure out what work was left to be done
- Best option: restore DB to a state as if transactions did not occur
  - Preserves atomicity
  - Implies that clients must not depend on uncommitted results

Database State During Query Execution

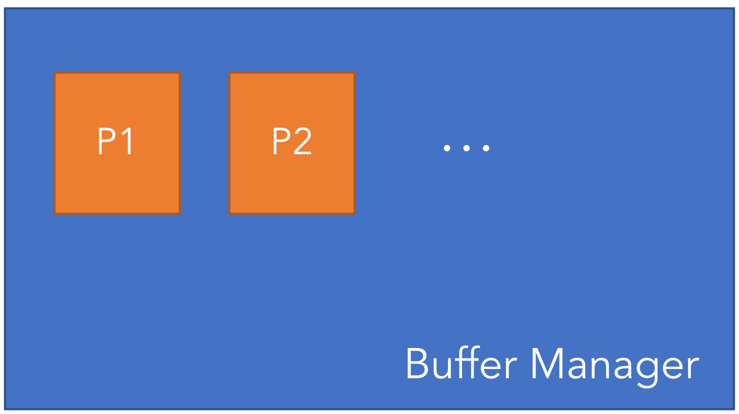


After crash, memory is gone!

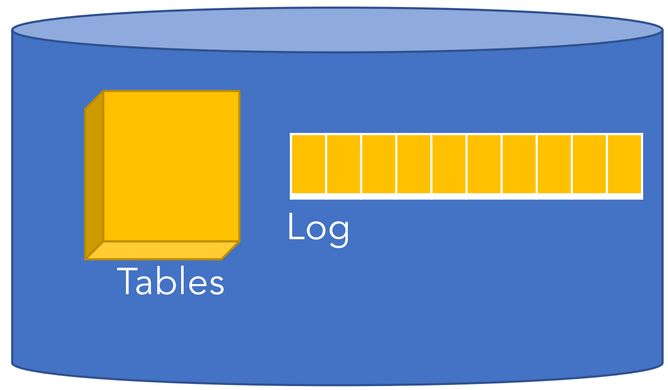
What problems could violate atomicity/ durability?

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

Why is a Log Needed?

- Log captures both *before* and *after* state of all writes
  - E.g., page X was X_0 is now X_1
- Also tells us which transactions committed and which did not

- Why do we need to record write contents?
  - Without this, can't tell whether a write has been applied or not
  - Allows us to *redo* committed writes, if state not in tables on disk
  - Allows us to *undo* uncommitted writes, if state in tables on disk

Logical vs Physical Logging

UNDO	REDO
0x0012 0002 F00E 0xB007 4789 F8F8 0xEEE0 2020 4447	0x0012 0004 F50E 0xB007 4789 F8F8 0xEEE0 2020 4447

Physical Logging

UNDO	REDO
Remove record X from position Y	Insert record X into position Y

Logical Logging

Logical logging is more compact, but it depends on pages fully reflecting (or not reflecting) operations being undone and redone

Write Ahead Logging

- Log records written *before* any action is taken
  - Start or commit transaction
  - Writing a page to tables on disk (reads are not logged)

- What could go wrong if we don't write ahead?

Why Write Ahead?

- Otherwise, we might:
  - update a page as a part of an uncommitted txn,
  - crash (which should cause us to rollback that update), and
  - not have any way to tell that the page was updated

- Write what we plan to do before we do it, and leave enough info in the log so that we can figure out whether we did it or not
  - Note that we do have to write everything twice, but logging is *sequential*, unlike writes to the DB, which are random

Types of Log Records

- Start (SOT) Log Sequence Number (LSN), Transaction ID (TID)
  - LSN is a monotonically increasing log record number
- End (EOT) LSN, TID, outcome (commit or abort)
- UNDO LSN, TID, before image
- REDO LSN, TID, after image

For next time:

- CHECKPOINT LSN, TID, state to limit how much is logged
- CLR LSN, TID, allows us to restart recovery

Two Complexities in Logging

- Sometimes we may want to dirty (uncommitted) pages back to the DB
  - Why?

- After a crash, some committed changes may not have been written back to the DB
  - Why?

Dirty Pages in DB

- If we don't write back dirty pages, they must be held in memory for the duration of the txn
  - Consider a transaction that updates all records in table

- A DB that writes back dirty pages is said to STEAL

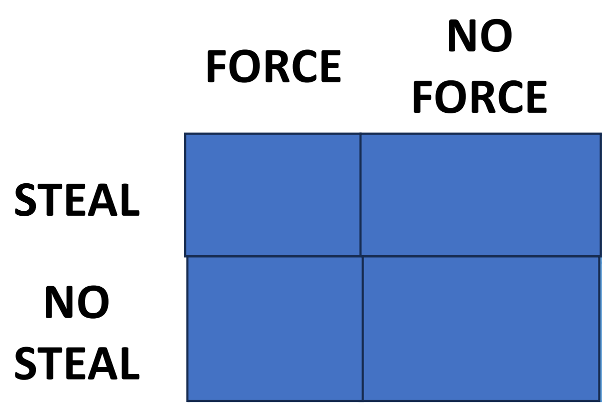
- STEAL requires UNDO to remove uncommitted txns

Some Committed Changes Not Written Back

- If we wrote back all pages at committee, would be slow!
  - Many random writes at commit time
- A DB that doesn't force all writes at commit is NO FORCE
- (Sequential) logging is sufficient to ensure recoverability, so FORCE is unnecessary for recoverability
- However, NO FORCE requires REDO to install logged writes to DB

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



In GoDB, 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

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

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

https://clicker.mit.edu/6.5830/

What do you think commercial OLTP database systems implement?

	FORCE	NO FORCE
STEAL	UNDO	UNDO & REDO
NO STEAL	UNDO	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 GoDB, 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

Recovery with NO FORCE / STEAL

- After crash, we must:

☺ - REDO “winner” transactions that had committed

☹ - UNDO “loser” transactions that had not committed

- Winner are transactions with SOT and COMMIT in log
- Losers are those with SOT and either (no EOT) or ABORT*

- Need to REDO winners from start to end
- Need to UNDO losers in reverse, from end to start
- Also need to UNDO aborted transactions

* Some disagreement in literature about whether ABORTed transactions are losers

3 Phases of Recovery

- Analysis: Scan log to find winners and losers
- REDO: Scan log from beginning to end for winners
- UNDO: Scan log from end to beginning for losers

- Many possible ways to do this, e.g., UNDO then REDO or REDO then UNDO
  - Next time will see a specific proposal and analyze why

Example

- Suppose we have 3 transactions, using NO FORCE, STEAL
- T1 writes A, commits
- T2 writes B, aborts
- T3 write C, system crashes

```
T1 --------------W(A)--------------- Commit
             T2-------------------------W(B)----------- Abort
                           T3--------W(C)------------------------ crash!
```

https://clicker.mit.edu/6.5830/

- Suppose we have 3 transactions, using NO FORCE, STEAL
- T1 writes A, commits
- T2 writes B, aborts
- T3 write C, system crashes

```
T1 --------------W(A)--------------------- Commit
             T2-------------------------W(B)----------- Abort
                           T3--------W(C)------------------------ crash!
```

https://clicker.mit.edu/6.5830/

- Suppose we have 3 transactions, using NO FORCE, STEAL
- T1 writes A, commits
- T2 writes B, aborts
- T3 write C, system crashes

```
T1 --------------W(A)--------------------- Commit
             T2-------------------------W(B)----------- Abort
                           T3--------W(C)------------------------ crash!
```

A) | SOT (T1) | SOT (T2) | W(A) | W(C) | W(B) | A(T1) | C(T2) |

B) | SOT (T1) | SOT (T2) | W(A) | SOT(T3) | W(C) | W(B) | C(T1) | A(T2) |

C) | W(A) | S(T3) | W(C) | W(B) | C(T1) | A(T2) |

Example

- Suppose we have 3 transactions, using NO FORCE, STEAL
- T1 writes A, commits
- T2 writes B, aborts
- T3 write C, system crashes

```
T1 --------------W(A)--------------- Commit
             T2-------------------------W(B)----------- Abort
                           T3--------W(C)------------------------ crash!
```

Log:

SOT (T1)	SOT (T2)	W(A)	S(T3)	W(C)	W(B)	C(T1)	A(T2)

Recovery Sketch

```
T1 --------------W(A)--------------- Commit
             T2------------------------W(B)----------- Abort
                           T3--------W(C)------------------------ crash!
```

- Analysis: T1 winner, T2 / T3 losers
- REDO:
  - Scan forward, replay WA
- UNDO:
  - Scan backward, undo WB, WC

After recovery, T1's effects are present, T2 / T3's aren't'

Recovery and Isolation

- Note that in a properly isolated DB, concurrent txns won't read and write the same pages, if using page locking
  - Don't need to worry about UNDOing a winner operation on the same page; example:

```
T1 -------- WX -------- Commit
                              T2 ---------------- WX ------- Crash
```

T2 WX cannot happen until T1 COMMIT, so when we UNDO T2 WX, we will rollback to T1 committed state. If T1 WX happened after T2 WX, T2 must have committed / aborted already.

Next time we will talk about how to handle different locking granularities

https://clicker.mit.edu/6.5830/

- Q1: Given the following log, if the system crashes which transactions are winners?

				☹					☹
1 SOT T1	2 SOT T2	3 WA T1	4 COMMIT T1	5 WB T2	6 SOT T3	7 WA T3	8 SOT T4	9 COMMIT T3	10 WC T4

- Q2: Which write LSNs will be UNDOne, in which order?
  A. 10, 7, 5
  B. 10, 5 ✓
  C. 5, 10
  D. 5, 7, 10

Need to undo T2 and T4

Next Time: ARIES

- Considered the gold standard in logging
- Specifies **all** the details
- NO FORCE/STEAL
- Shows how its possible to make recovery recoverable
- Shows how to use logical UNDO logging
- Shows how to handle nested transactions
  - (which we won't talk about)
- Shows how to make checkpoints work