Lectures 7: Intro to Transactions & Logging

Goals for this pair of lectures

- **Transactions** are a programming abstraction that enables the DBMS to handle *recovery* and *concurrency* for users.

- **Application:** Transactions are critical for users
  - Even casual users of data processing systems!

- **Fundamentals:** The basics of **how** TXNs work
  - Transaction processing is part of the debate around new data processing systems
  - Give you enough information to understand how TXNs work, and the main concerns with using them

Note that we are not implementing it

Today's Lecture

1. Transactions
2. Properties of Transactions: ACID
3. Logging

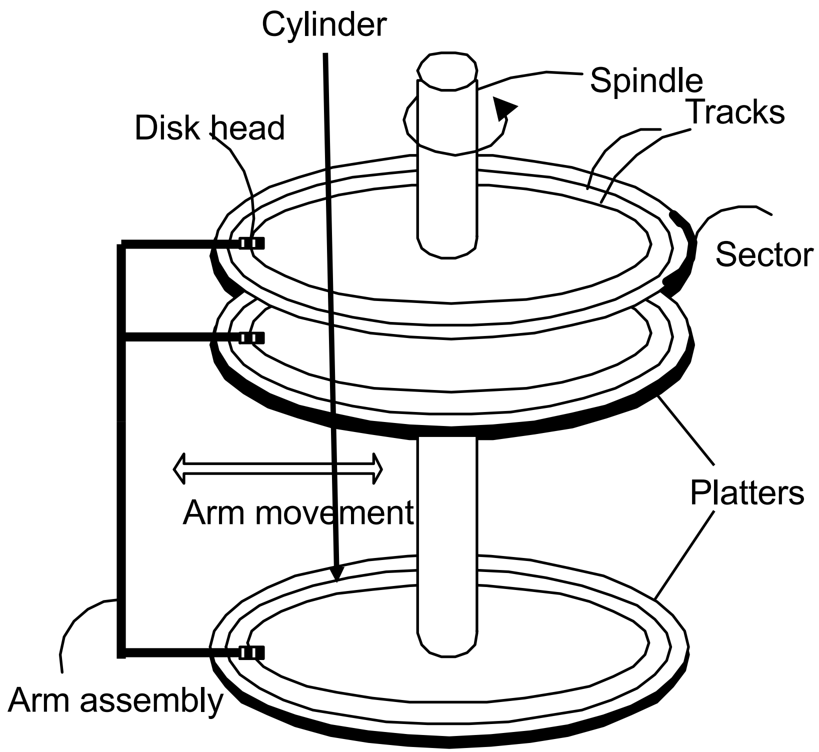
1. Transactions

What you will learn about in this section

1. Our “model” of the DBMS / computer
2. Transactions basics
3. Motivation: Recovery & Durability
4. Motivation: Concurrency *[next lecture]*

High-level: Disk vs. Main Memory

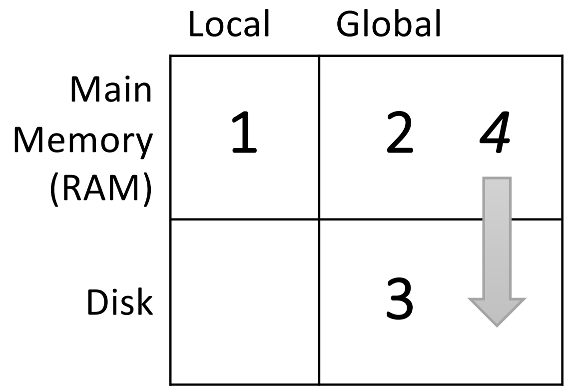
- **Disk**:
  - *Slow*
    - Sequential access
      - (although fast sequential reads)
  - *Durable*
    - We will assume that once on disk, data is safe!
  - Cheap



High-level: Disk vs. Main Memory

- Random Access Memory (RAM) or **Main Memory**:
  - *Fast*
    - Random access, byte addressable
      - ~10x faster for sequential access
      - ~100,000x faster for random access!
  - *Volatile*
    - Data can be lost if e.g. crash occurs, power goes out, etc!
  - Expensive
    - For $100, get 16GB of RAM vs. 2TB of disk!

Our model: Three Types of Regions of Memory



1. **Local:** In our model each process in a DBMS has its own local memory, where it stores values that only it "sees"

2. **Global:** Each process can read from / write to shared data in main memory

3. **Disk:** Global memory can read from / flush to disk

4. ***Log:*** *Assume on stable disk storage- spans both main memory and disk…*

Log is a *sequence* from main memory -> disk

"<u>Flushing</u> to disk" = writing to disk from main memory

High-level: Disk vs. Main Memory

- Keep in mind the tradeoffs here as motivation for the mechanisms we introduce
  - Main memory: fast but limited capacity, volatile
  - Vs. Disk: slow but large capacity, durable

How do we effectively utilize ***both*** ensuring certain critical guarantees?

Transactions

Transactions: Basic Definition

A <u>transaction (“TXN”)</u> is a sequence of one or more ***operations*** (reads or writes) which reflects ***a single real-world transition***.

In the real world, a TXN either happened completely or not at all

```
START TRANSACTION
     UPDATE Product
     SET Price = Price – 1.99
     WHERE pname = ‘Gizmo’
COMMIT
```

Transactions: Basic Definition

A **transaction ("TXN")** is a sequence of one or more ***operations*** (reads or writes) which reflects ***a single real-world transition***.

In the real world, a TXN either happened completely or not at all

Examples:

- Transfer money between accounts
- Purchase a group of products
- Register for a class (either waitlist or allocated)

Transactions in SQL

- In “ad-hoc” SQL:
  - Default: each statement = one transaction

- In a program, multiple statements can be grouped together as a transaction:

```
START TRANSACTION
    UPDATE Bank SET amount = amount – 100
    WHERE name = ‘Bob’
    UPDATE Bank SET amount = amount + 100
    WHERE name = ‘Joe’
COMMIT
```

Model of Transaction for CS 145

Note: For 145, we assume that the DBMS *only* sees reads and writes to data

- User may do much more
- In real systems, databases do have more info...

Motivation for Transactions

Grouping user actions (reads & writes) into *transactions* helps with two goals:

1. **Recovery & Durability**: Keeping the DBMS data consistent and durable in the face of crashes, aborts, system shutdowns, etc.

This lecture!

2. **Concurrency:** Achieving better performance by parallelizing TXNs *without* creating anomalies

Next lecture

Motivation

1. Recovery & Durability of user data is essential for reliable DBMS usage

- The DBMS may experience crashes (e.g. power outages, etc.)
- Individual TXNs may be aborted (e.g. by the user)

Idea: Make sure that TXNs are either **durably stored in full, or not at all**; keep log to be able to "roll-back" TXNs

Protection against crashes / aborts

```
Client 1:
    INSERT INTO SmallProduct(name, price)
        SELECT pname, price
        FROM Product
        WHERE price <= 0.99

    DELETE Product
        WHERE price <=0.99
```

Crash / abort!

What goes wrong?

Protection against crashes / aborts

```
Client 1:
     START TRANSACTION
          INSERT INTO SmallProduct(name, price)
               SELECT pname, price
               FROM Product
               WHERE price <= 0.99

          DELETE Product
               WHERE price <=0.99
     COMMIT OR ROLLBACK
```

Now we'd be fine! We'll see how / why this lecture

Motivation

2. Concurrent execution of user programs is essential for good DBMS performance.

- Disk accesses may be frequent and **slow**- optimize for throughput (# of TXNs), trade for latency (time for any one TXN)
- Users should still be able to execute TXNs as if in **isolation** and such that **consistency** is maintained

Idea: Have the DBMS handle running several user TXNs concurrently, in order to keep CPUs humming…

Multiple users: single statements

```
Client 1: UPDATE Product
          SET Price = Price – 1.99
          WHERE pname = 'Gizmo'

Client 2: UPDATE Product
          SET Price = Price*0.5
          WHERE pname='Gizmo'
```

Two managers attempt to discount products *concurrently*-
What could go wrong?

Multiple users: single statements

```
Client 1: START TRANSACTION
                 UPDATE Product
                 SET Price = Price – 1.99
                 WHERE pname = ‘Gizmo’
          COMMIT

Client 2: START TRANSACTION
                 UPDATE Product
                 SET Price = Price*0.5
                 WHERE pname=‘Gizmo’
          COMMIT
```

Now works like a charm- we’ll see how / why next lecture…

2. Properties of Transactions

What you will learn about in this section

1. Atomicity
2. Consistency
3. Isolation
4. Durability
5. ACTIVITY?

Transaction Properties: ACID

- **A**tomic
  - State shows either all the effects of txn, or none of them
- **C**onsistent
  - Txn moves from a state where integrity holds, to another where integrity holds
- **I**solated
  - Effect of txns is the same as txns running one after another (ie looks like batch mode)
- **D**urable
  - Once a txn has committed, its effects remain in the database

ACID continues to be a source of great debate!

ACID: Atomicity

- TXN's activities are atomic: **all or nothing**
  - Intuitively: in the real world, a transaction is something that would either occur *completely* or *not at all*
- Two possible outcomes for a TXN
  - It *commits*: all the changes are made
  - It *aborts*: no changes are made

ACID: Consistency

- The tables must always satisfy user-specified ***integrity constraints***
  - *Examples:*
    - Account number is unique
    - Stock amount can’t be negative
    - Sum of *debits* and of *credits* is 0

- How consistency is achieved:
  - Programmer makes sure a txn takes a consistent state to a consistent state
  - *System* makes sure that the txn is **atomic**

ACID: Isolation

- A transaction executes concurrently with other transactions

- **Isolation**: the effect is as if each transaction executes in *isolation* of the others.

  - E.g. Should not be able to observe changes from other transactions during the run

ACID: Durability

- The effect of a TXN must continue to exist (***"persist"***) after the TXN
  - And after the whole program has terminated
  - And even if there are power failures, crashes, etc.
  - And etc…

- Means: Write data to **disk**

Change on the horizon?
Non-Volatile Ram (NVRam).
Byte addressable.

Challenges for ACID properties

- In spite of failures: Power failures, but not media failures

This lecture

- Users may abort the program: need to "rollback the changes"
  - Need to *log* what happened

- Many users executing concurrently
  - Can be solved via locking (we'll see this next lecture!)

Next lecture

And all this with... Performance!!

A Note: ACID is contentious!

- Many debates over ACID, both **historically** and **currently**
- Many newer “NoSQL” DBMSs relax ACID
- In turn, now “NewSQL” reintroduces ACID compliance to NoSQL-style DBMSs…

ACID is an extremely important & successful paradigm, but still debated!

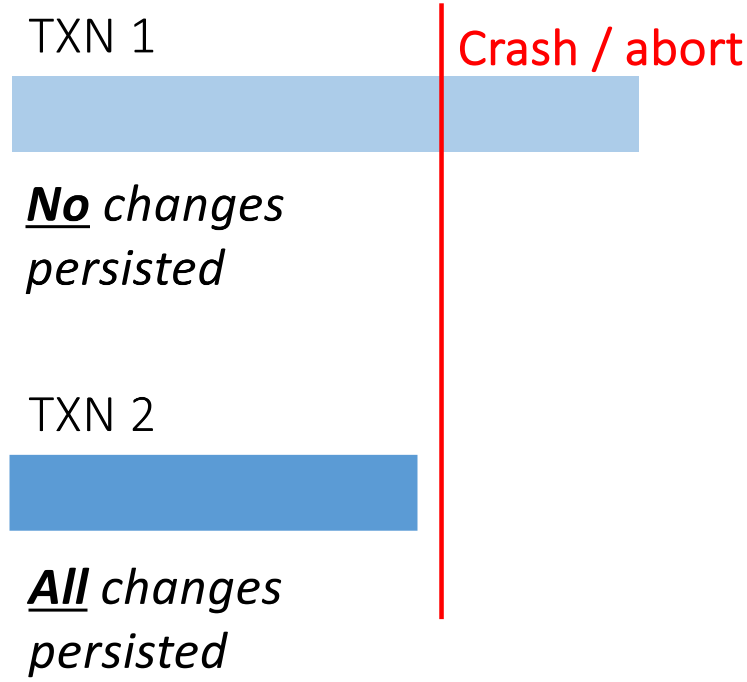
3. Atomicity & Durability via Logging

Motivation & Basics

Goal for this lecture: Ensuring Atomicity & Durability

ACID

- **Atomicity:**
  - TXNs should either happen completely or not at all
  - If abort / crash during TXN, *no* effects should be seen
- **Durability:**
  - If DBMS stops running, changes due to completed TXNs should all persist
  - *Just store on stable disk*



We'll focus on how to accomplish atomicity (via logging)

The Log

- Is a list of modifications

- Log is *duplexed* and *archived* on stable storage.

Assume we don't lose it!

- Can **force write** entries to disk
  - A page goes to disk.

- All log activities ***handled transparently*** the DBMS.

Basic Idea: (Physical) Logging

- Record UNDO information for every update!
    - Sequential writes to log
    - Minimal info (diff) written to log

- The **log** consists of **an ordered list of actions**
    - Log record contains:

      <XID, location, old data, new data>

This is sufficient to UNDO any transaction!

Why do we need logging for atomicity?

- Couldn't we just write TXN to disk **only** once whole TXN complete?
  - Then, if abort / crash and TXN not complete, it has no effect- atomicity!
  - *With unlimited memory and time, this could work…*

- However, we **need to log partial results of TXNs** because of:
  - Memory constraints (enough space for full TXN??)
  - Time constraints (what if one TXN takes very long?)

We need to write partial results to disk!
…And so we need a **log** to be able to ***undo*** these partial results!

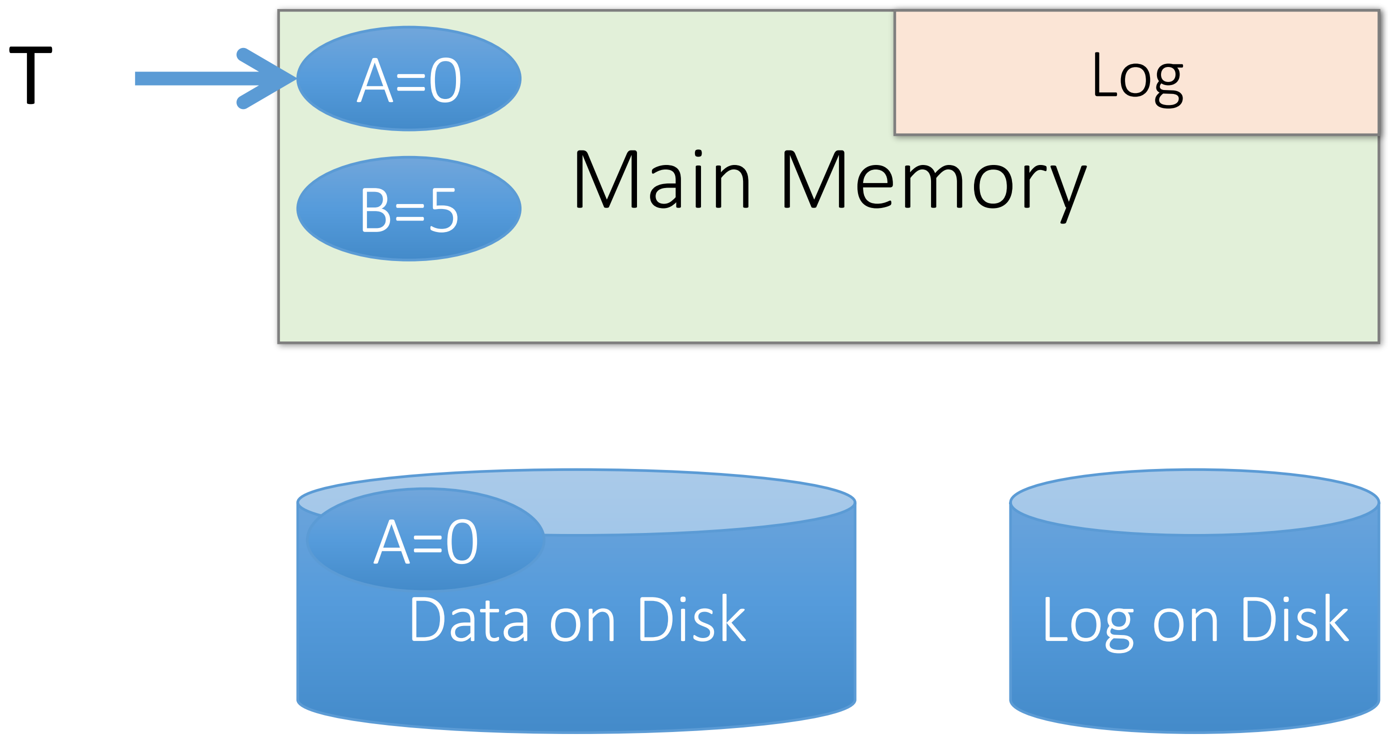
What you will learn about in this section

1. Logging: An animation of commit protocols

A Picture of Logging

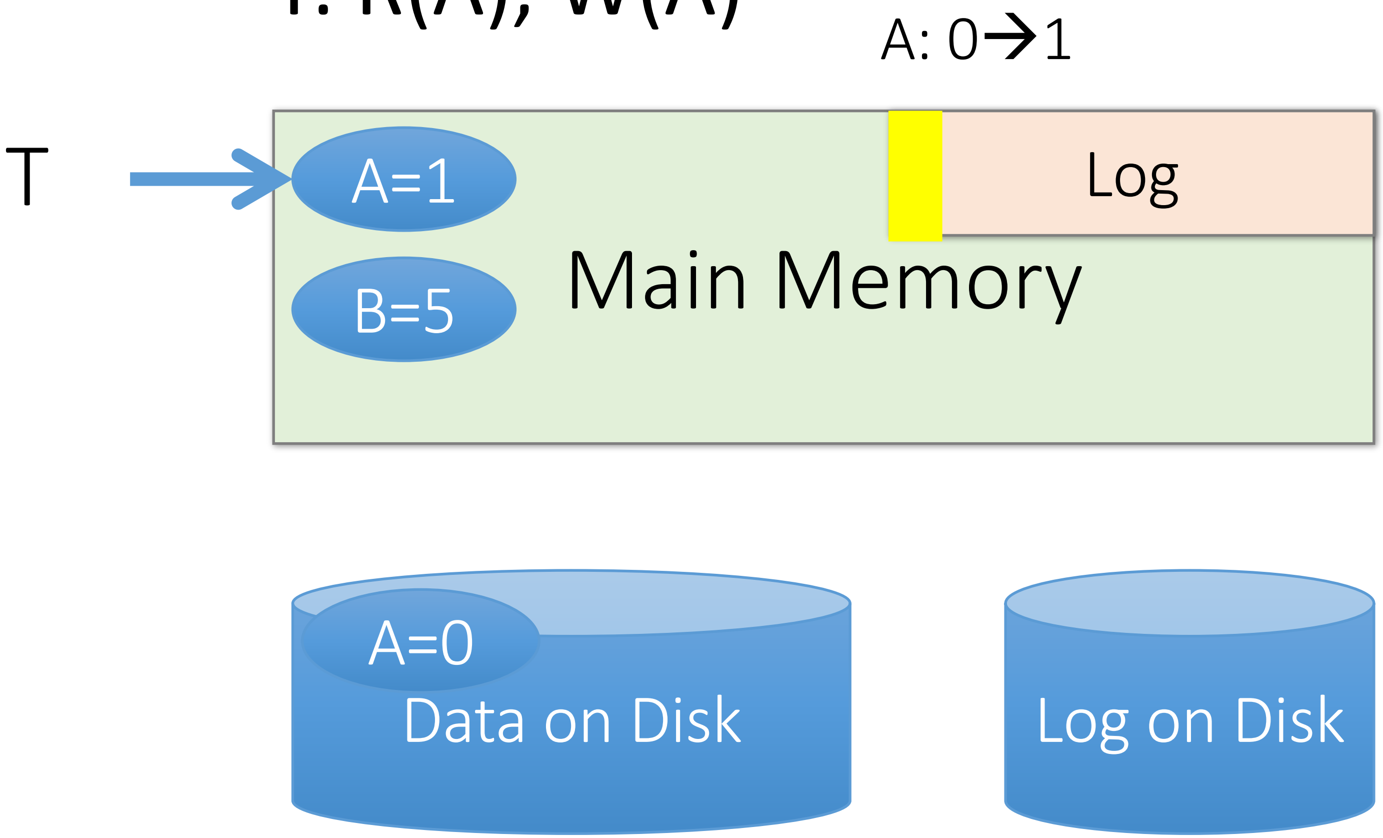
A picture of logging

T: R(A), W(A)



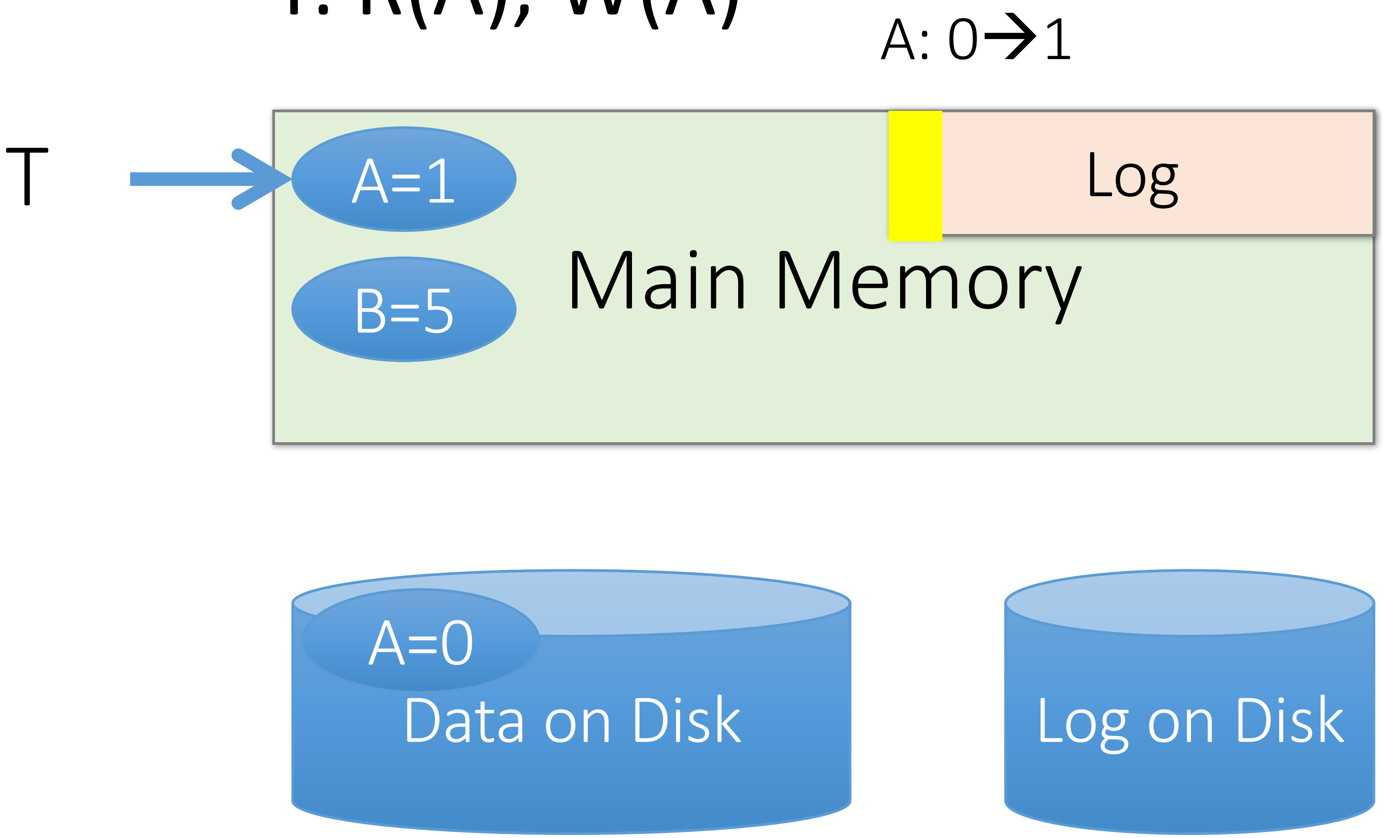
A picture of logging

T: R(A), W(A)



A picture of logging

T: R(A), W(A)



Operation recorded in log in main memory!

What is the correct way to write this all to disk?

- We'll look at the *Write-Ahead Logging (WAL)* protocol

- We'll see why it works by looking at other protocols which are incorrect!

Remember: Key idea is to ensure durability *while* maintaining our ability to "undo"!

Write-Ahead Logging (WAL)
TXN Commit Protocol

Transaction Commit Process

1. FORCE Write **commit** record to log

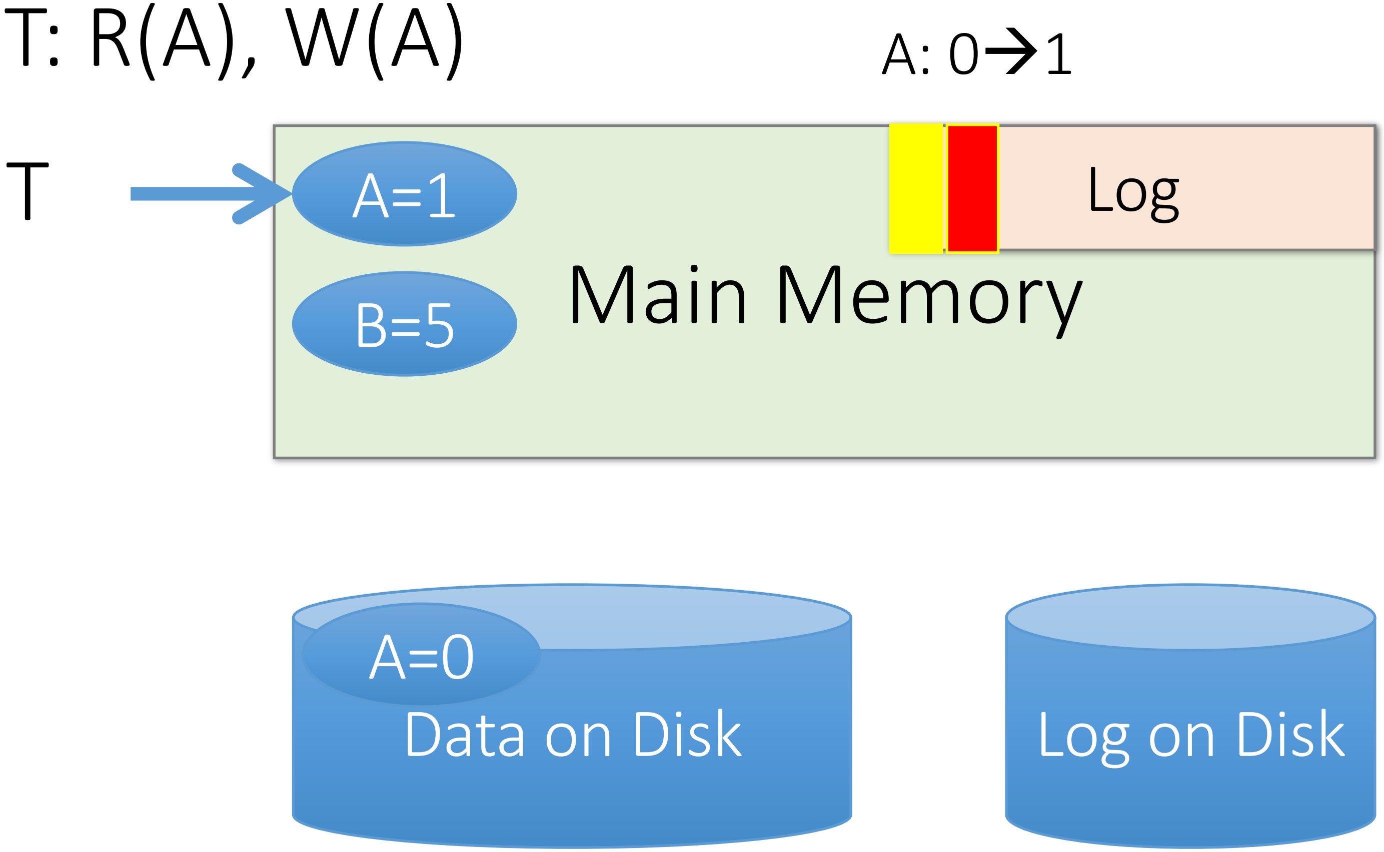
2. All log records up to last update from this TX are FORCED

3. Commit() returns

Transaction is committed *once commit log record is on stable storage*

Incorrect Commit Protocol #1

T: R(A), W(A)



Let's try committing *before* we've written either data or log to disk…

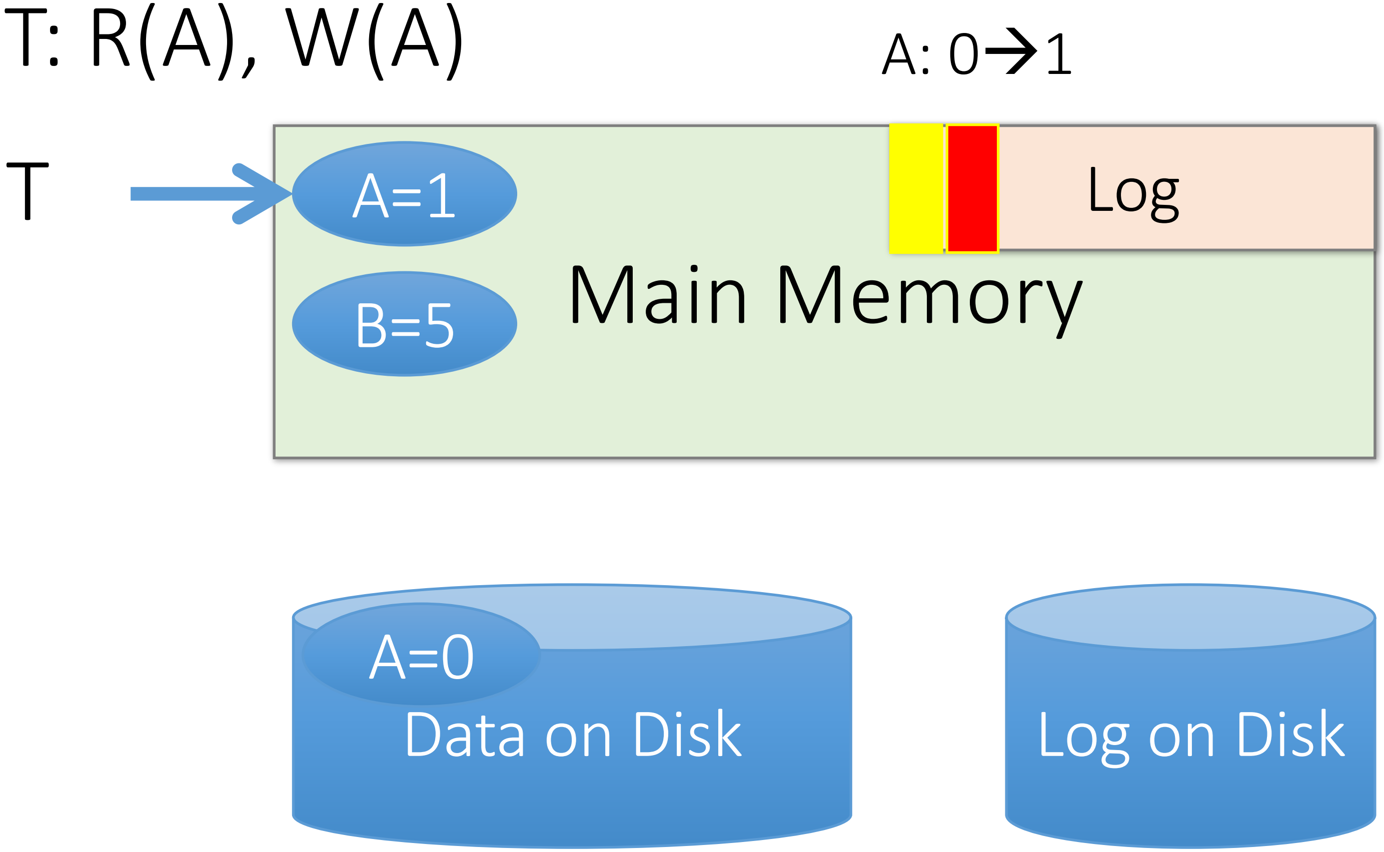
OK, Commit!

If we crash now, is T durable?

Lost T's update!

Incorrect Commit Protocol #2

T: R(A), W(A)



Let's try committing *after* we've written data but *before* we've written log to disk…

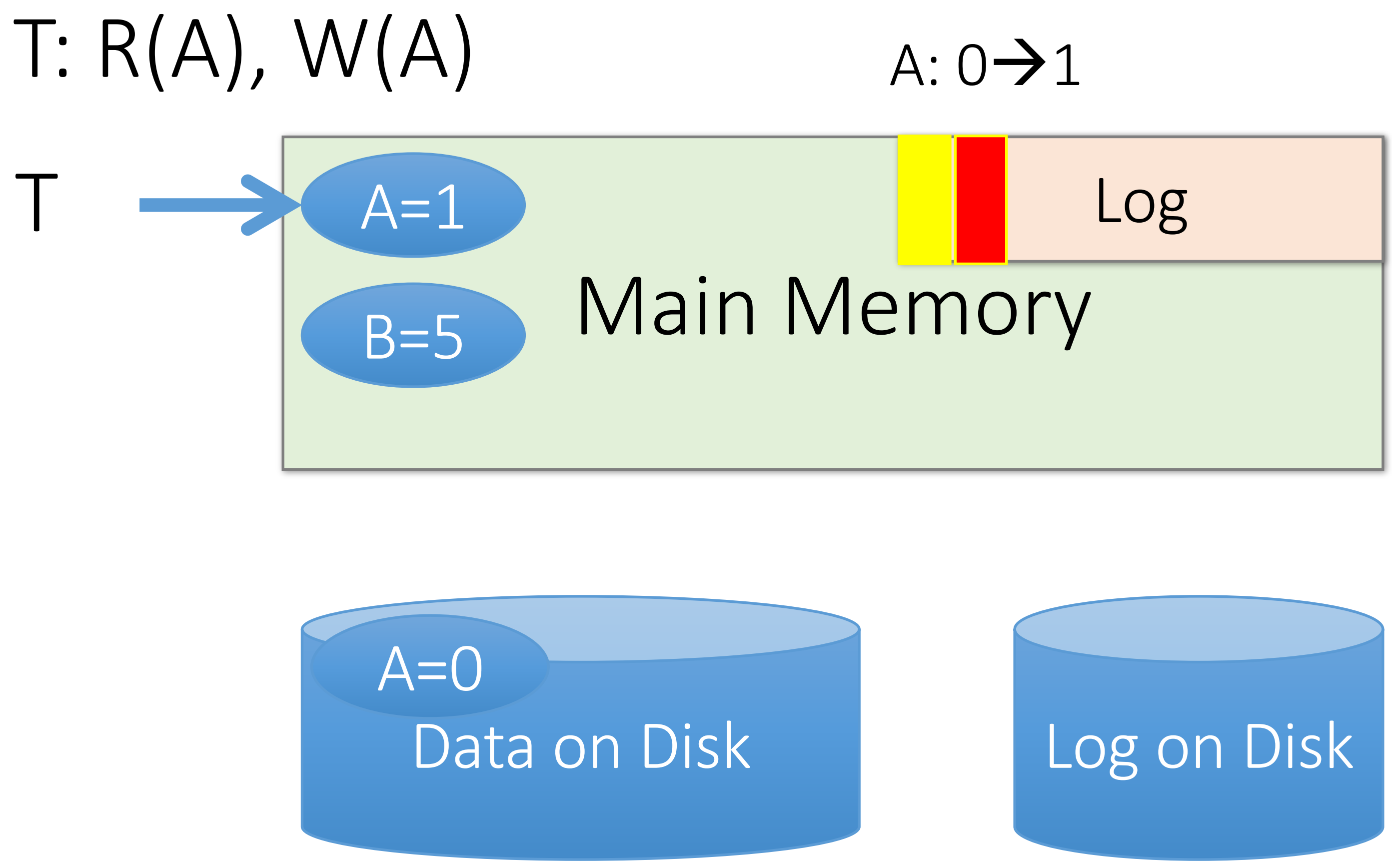
OK, Commit!

If we crash now, is T durable? Yes! Except…

How do we know whether T was committed??

Improved Commit Protocol (WAL)

Write-ahead Logging (WAL) Commit Protocol



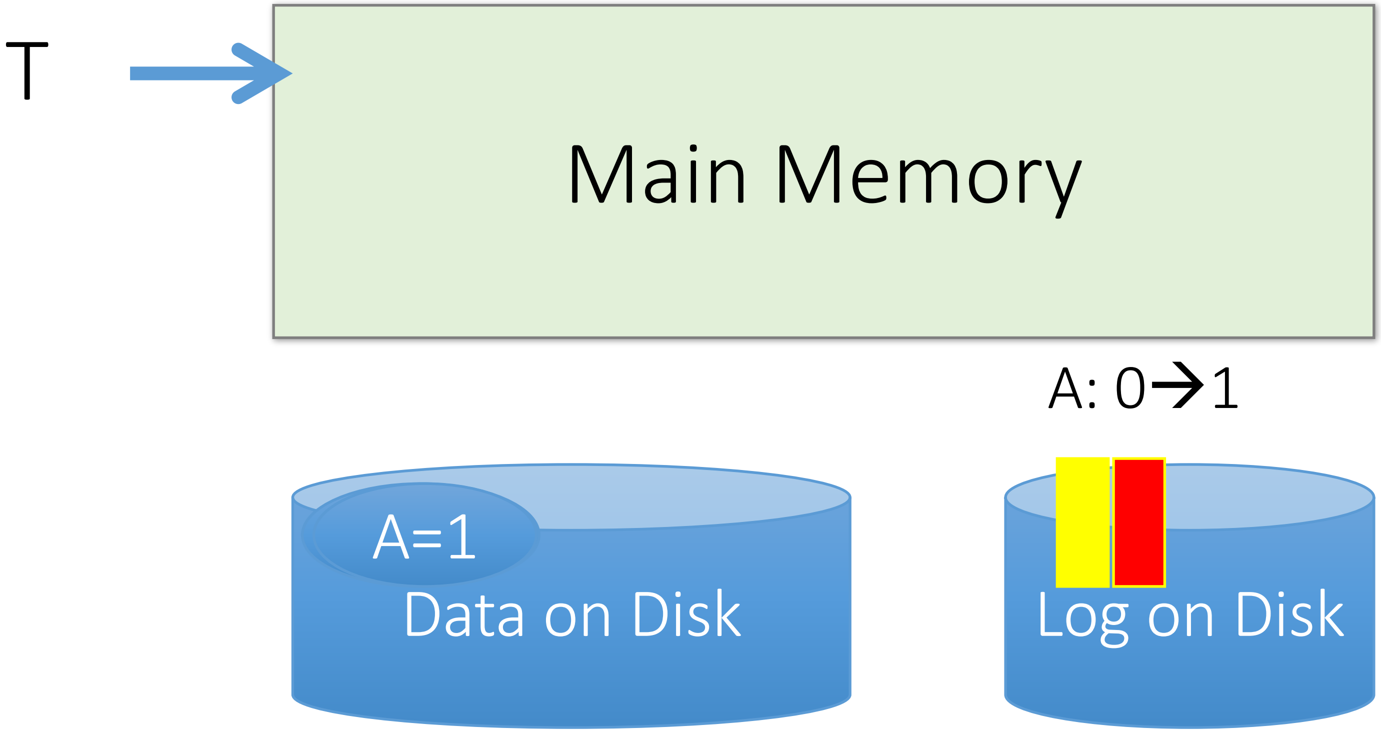
This time, let's try committing ***after* we've written log to disk but *before* we've written data to disk**... this is WAL!

OK, Commit!

If we crash now, is T durable?

Write-ahead Logging (WAL) Commit Protocol

T: R(A), W(A)



This time, let's try committing ***after*** **we've written log to disk but** ***before*** **we've written data to disk**… this is WAL!

OK, Commit!

If we crash now, is T durable?

USE THE LOG!

Write-Ahead Logging (WAL)

- DB uses **Write-Ahead Logging (WAL)** Protocol:

Each update is logged! Why not reads?

1. Must *force log record* for an update *before* the corresponding data page goes to storage

→ Atomicity

2. Must *write all log records* for a TX *before commit*

→ Durability

Logging Summary

- If DB says TX **commits**, TX effect **remains** after database crash
- DB can **undo actions** and help us with **atomicity**
- This is only half the story…