M146 Database Systems
Spring 2021

Georgia Koutrika

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 – 3000
        WHERE name = ‘Bob’
        UPDATE Bank SET amount = amount + 3000
        WHERE name = ‘Joe’
COMMIT
```

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.

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

Motivation for Transactions

<u>**1. Recovery & Durability**</u> 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

Motivation for Transactions

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 busy…
The DBMS uses locks to ensure correctness

Transaction Properties: ACID

- Atomicity: all the actions in a transaction are executed as a single atomic operation; either they are all carried out or none are
- Consistency: if a transaction begins with the DB in a consistent state, it must finish with the DB in a consistent state
- Isolation: a transaction should execute as if it is the only one executing; it is protected (isolated) from the effects of concurrently running transactions
- Durability: if a transaction has been successfully completed, its effects should be permanent

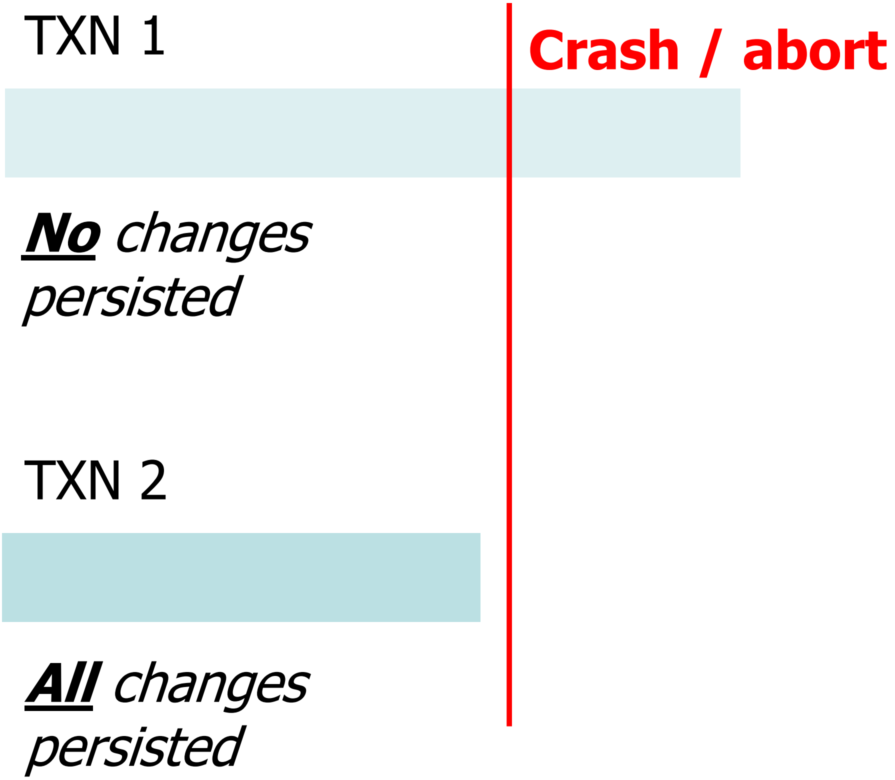
ACID continues to be a source of great debate!

Ensuring Atomicity & Durability

ACID

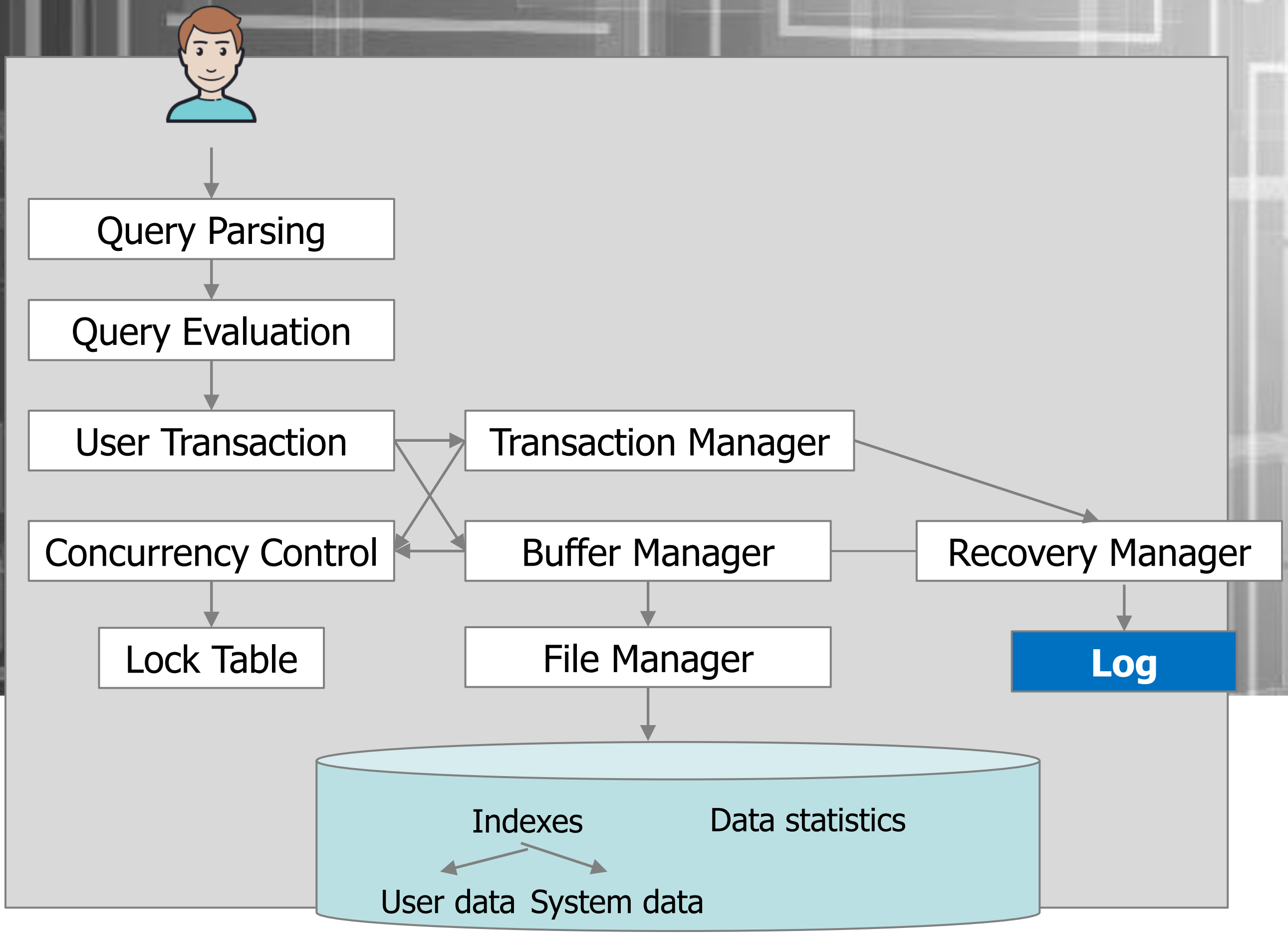
- **Atomicity:**
  - TXNs should either happen completely or not at all
  - If abort / crash during TXN, *no* effects should be seen

- **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)

LOGGING



Query Parsing
Query Evaluation
User Transaction
Transaction Manager
Concurrency Control
Buffer Manager
Recovery Manager
Lock Table
File Manager
Log
Indexes
Data statistics
User data
System data

The log

- The following actions are recorded in the log
  - Whenever a transaction writes an object
  - Whenever a transaction commits/aborts

- The log record must be on disk before the data record reaches the disk

The Log

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

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

- All log-related activities (in fact, all concurrency control related activities) are handled by the DBMS

- The user does not know anything

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>

    Log records are chained by transaction ID (why?)

This is sufficient to UNDO any transaction!

Why do we need logging for atomicity?

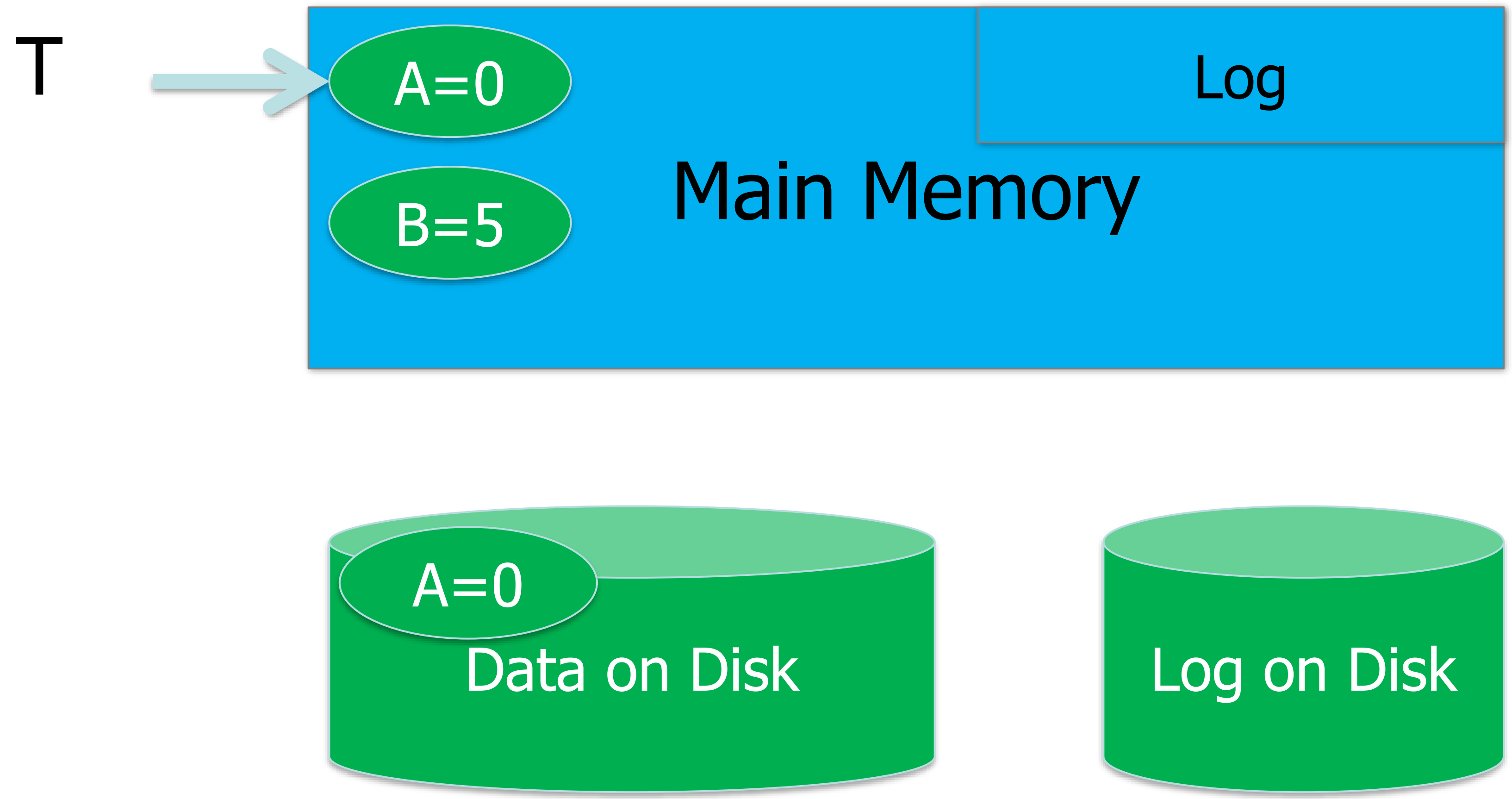
- Couldn't we just write TXN to disk **only** once whole TXN complete?
  - Then, if abort / crash and TXN not complete, it has no effect- atomicity!
  - *With unlimited memory and time, this could work…*

- However, we **need to log partial results of TXNs** because of:
  - Memory constraints (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!

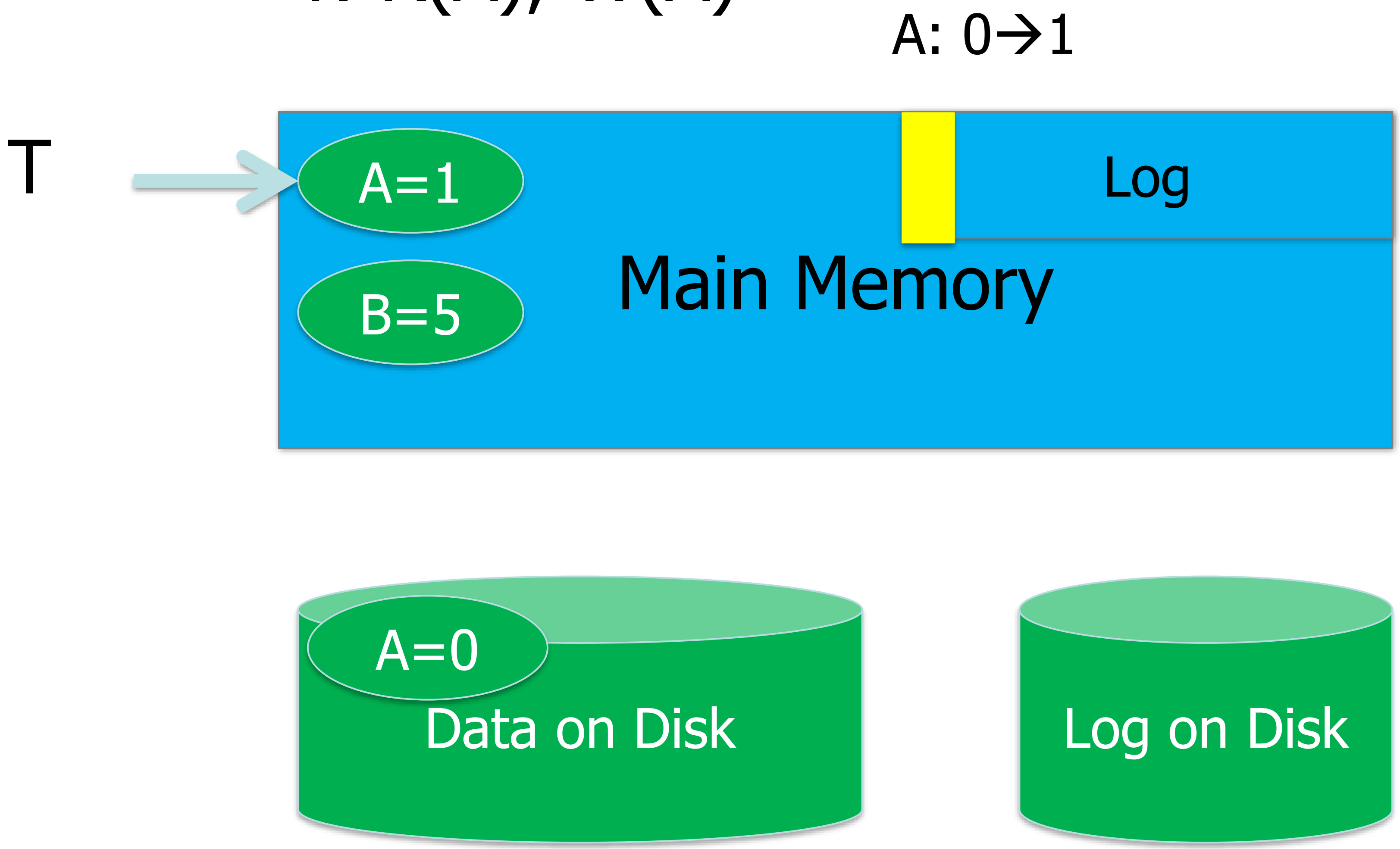
A picture of logging

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?

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"!

Transaction Commit Process

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

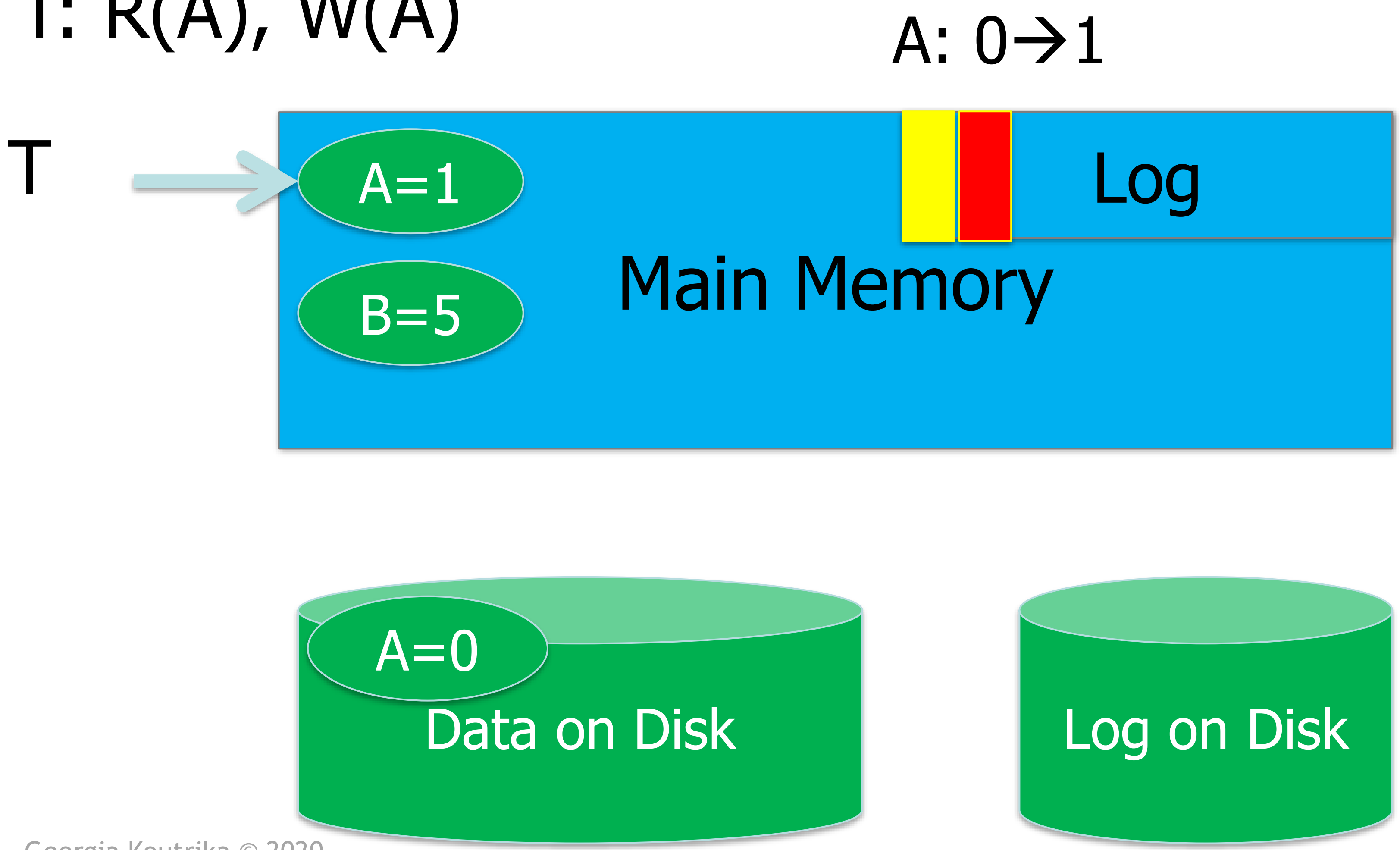
2. All log records up to last update from this TX are FORCED

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…

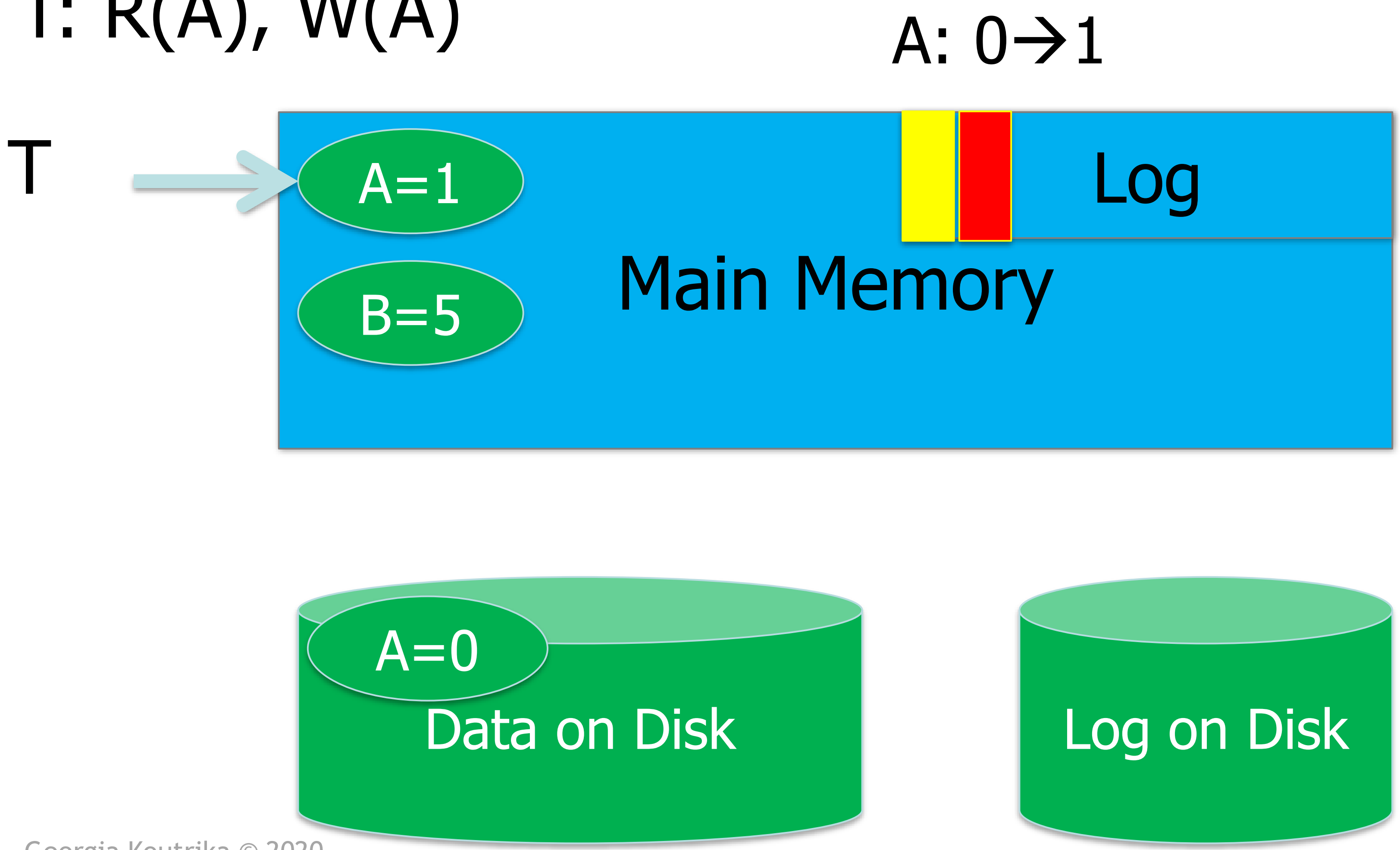
OK, Commit!

If we crash now, is T durable?

Lost T's update!

Incorrect Commit Protocol #2

T: R(A), W(A)



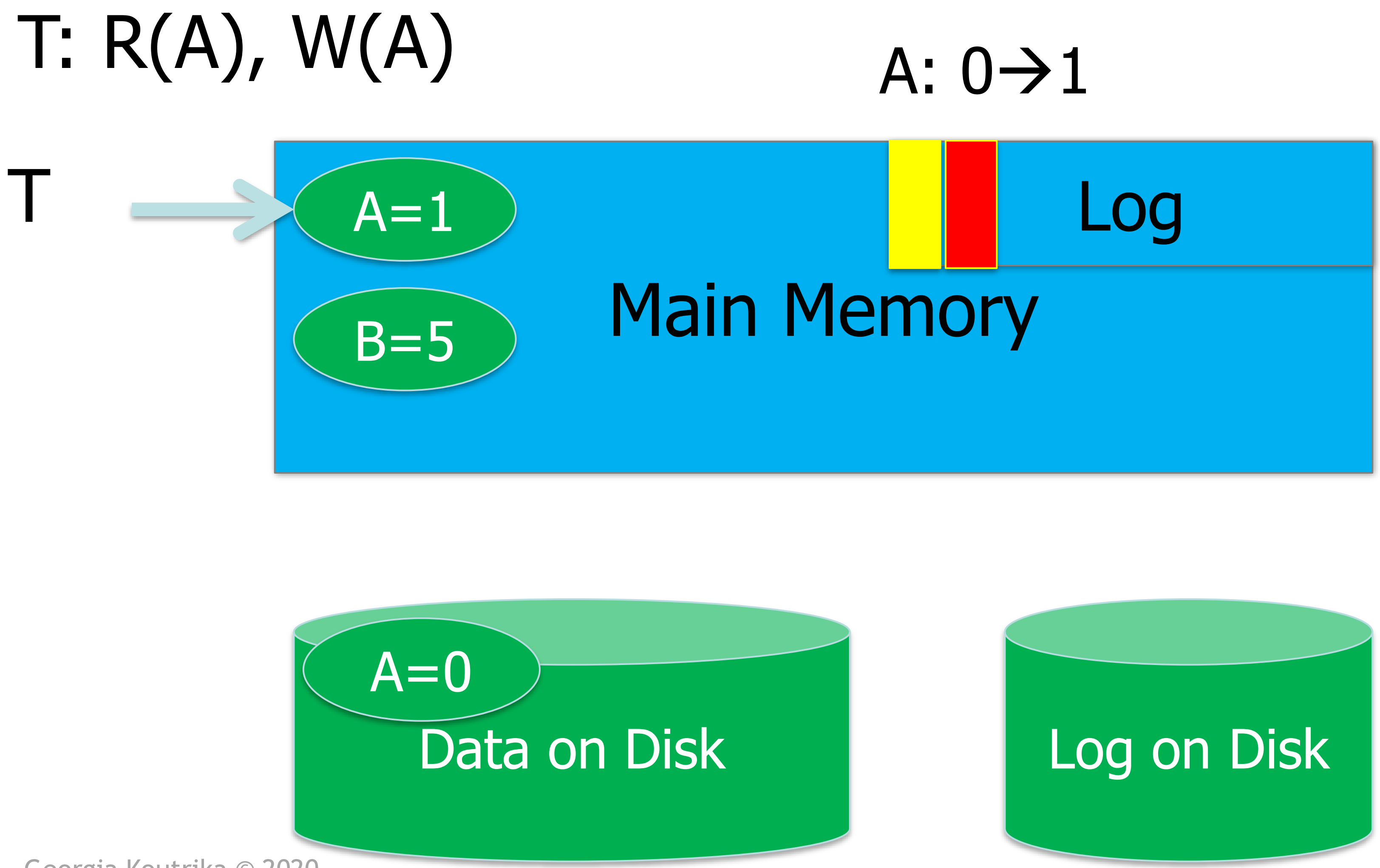
Let's try committing *after* we've written data but *before* we've written log to disk…

OK, Commit!

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

How do we know whether T was committed??

Write-ahead Logging (WAL) Commit Protocol



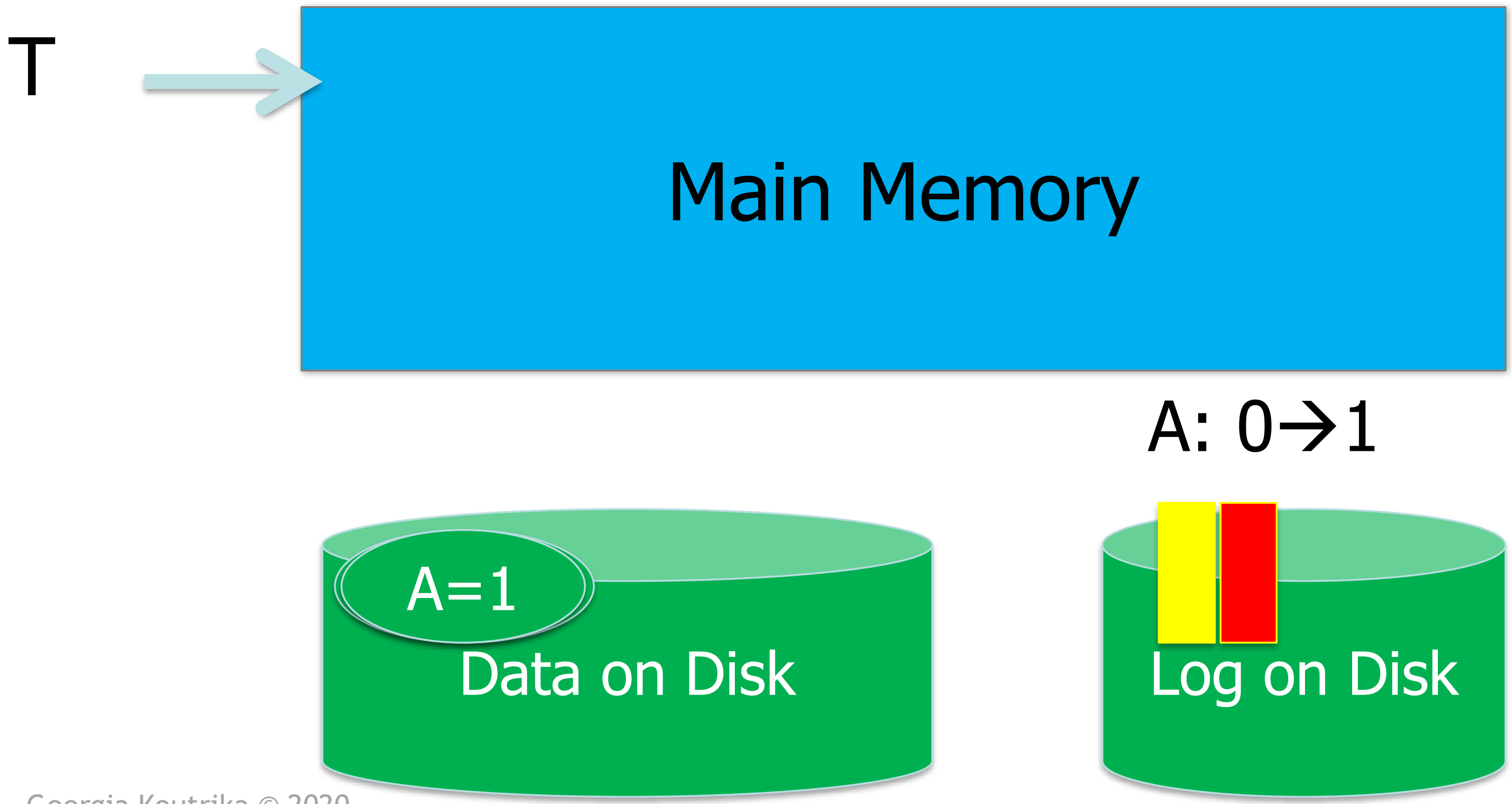
This time, let's try committing ***after* we've written log to disk but *before* we've written data to disk**... this is WAL!

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…

Crash recovery

Three phases to recovery (ARIES)

- Analysis: scan log forward, identifying committed and aborted/unfinished transactions
- Redo: all committed transactions are made durable
- Undo: the actions of all aborted and/or unfinished transactions are undone