CS150: Database & Datamining

Lecture 8: Transactions I: Intro to Transactions & Logging

ShanghaiTech-SIST

Spring 2019

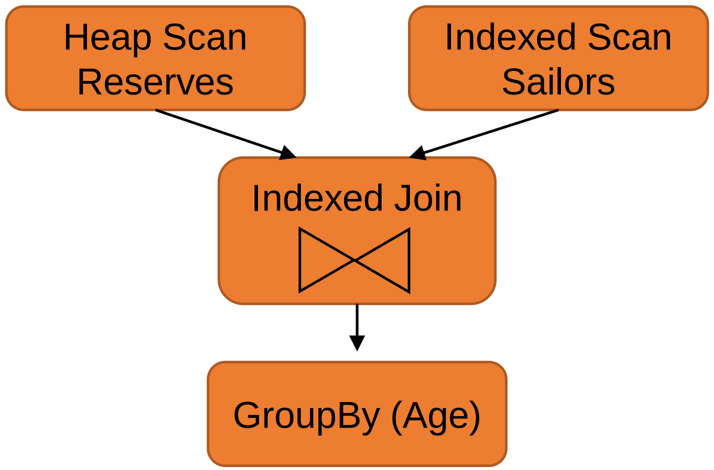
Acknowledgement: Slides are adopted from the Berkeley course CS186 by Joey Gonzalez and Joe Hellerstein, Stanford CS145 by Peter Bailis.

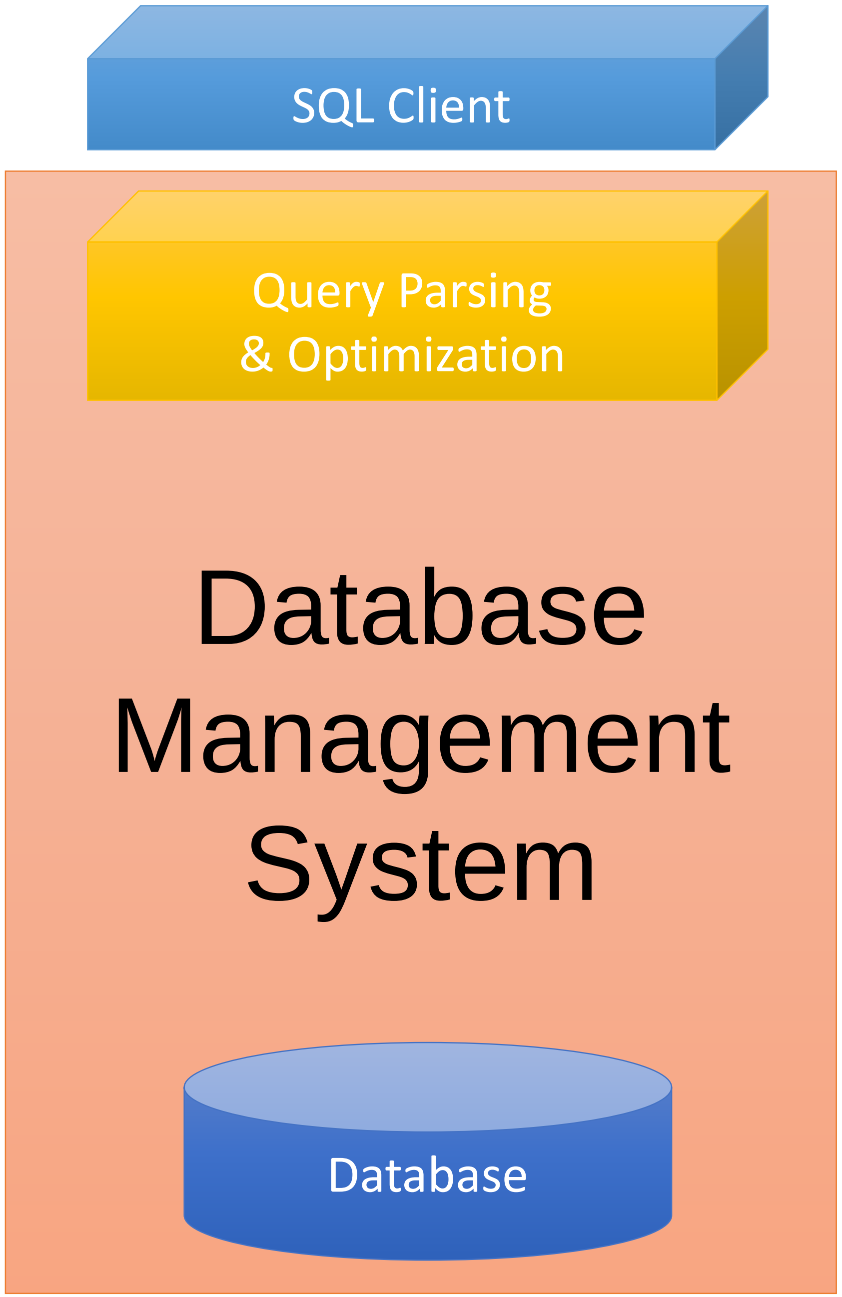
Architecture of a DBMS

Parse, check, and verify the SQL expression

```
SELECT s.sid, s.sname, r.bid
FROM Sailors s, Reserves r
WHERE s.sid = r.sid AND s.age > 30
```

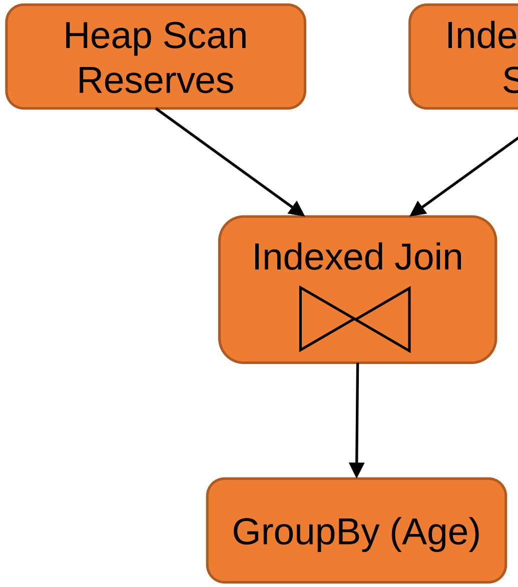
and translate into an efficient relational query plan

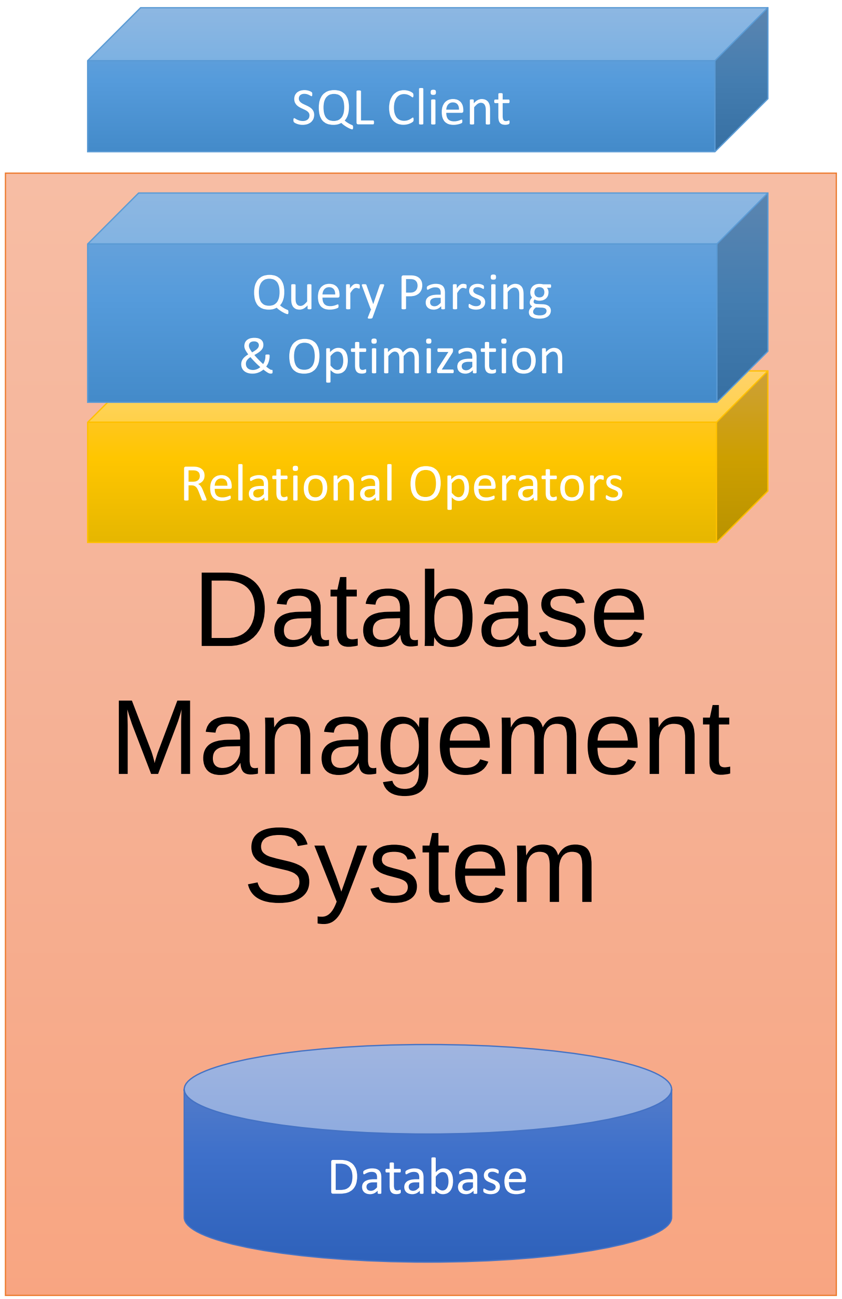




Architecture of a DBMS

Execute the dataflow by operating on **records** and **files**

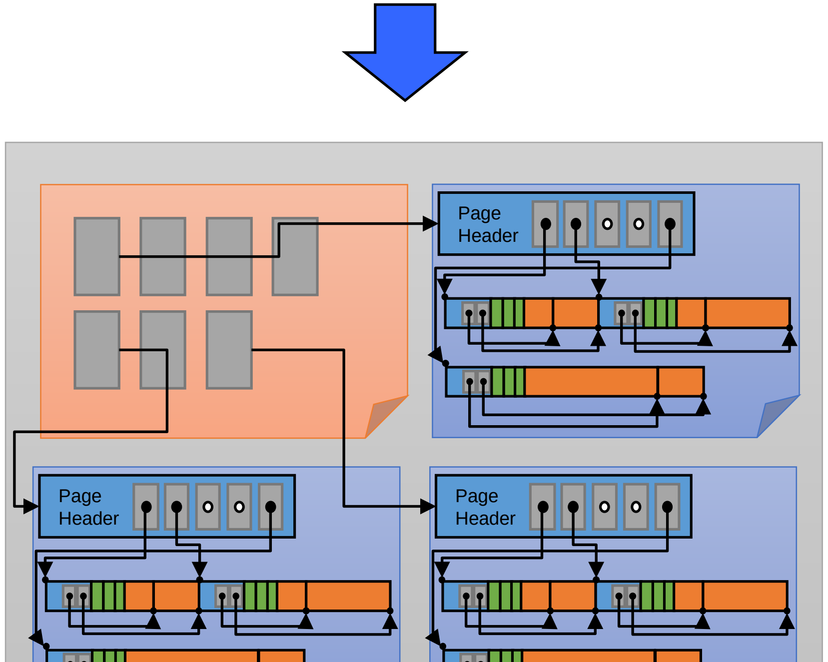


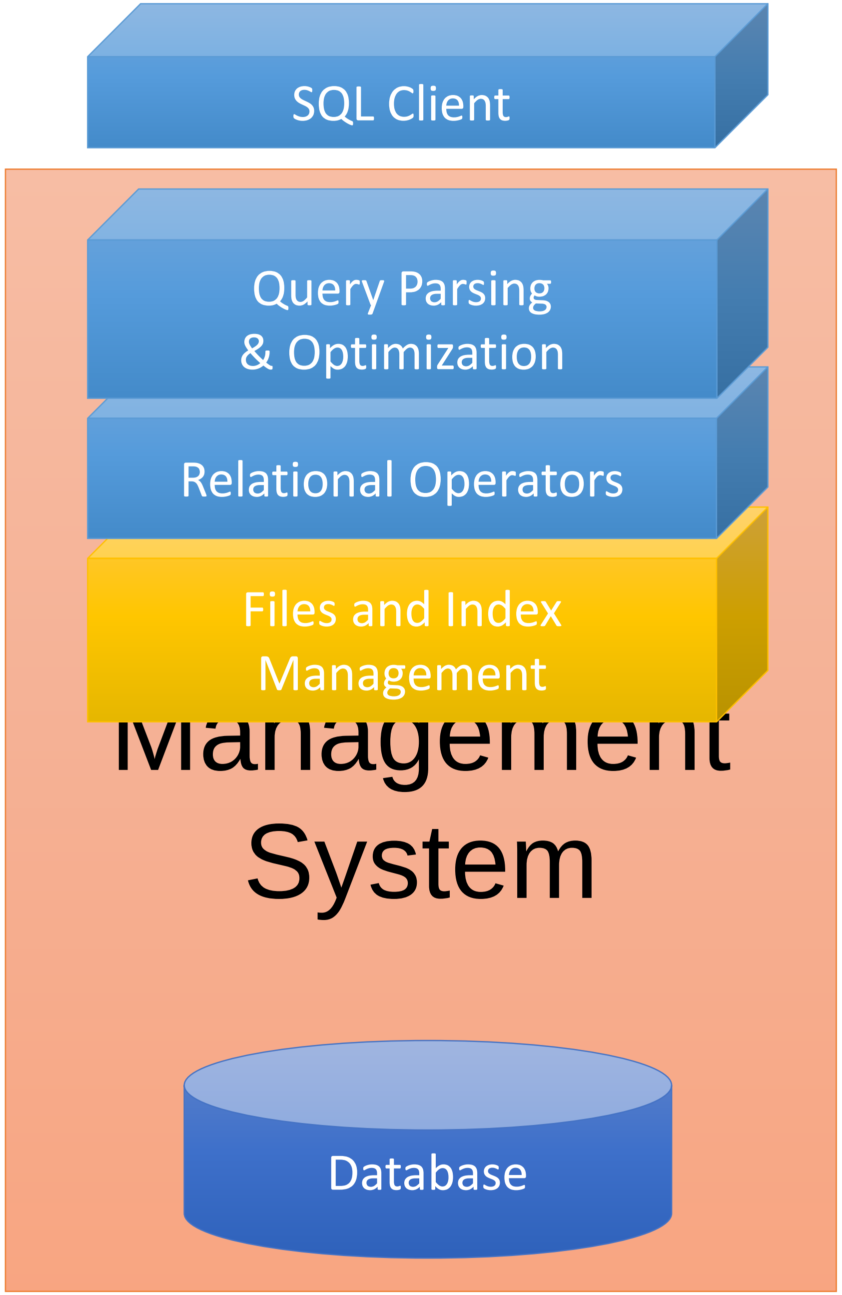


Architecture of a DBMS

Organizing tables and records as groups of pages in a logical file.

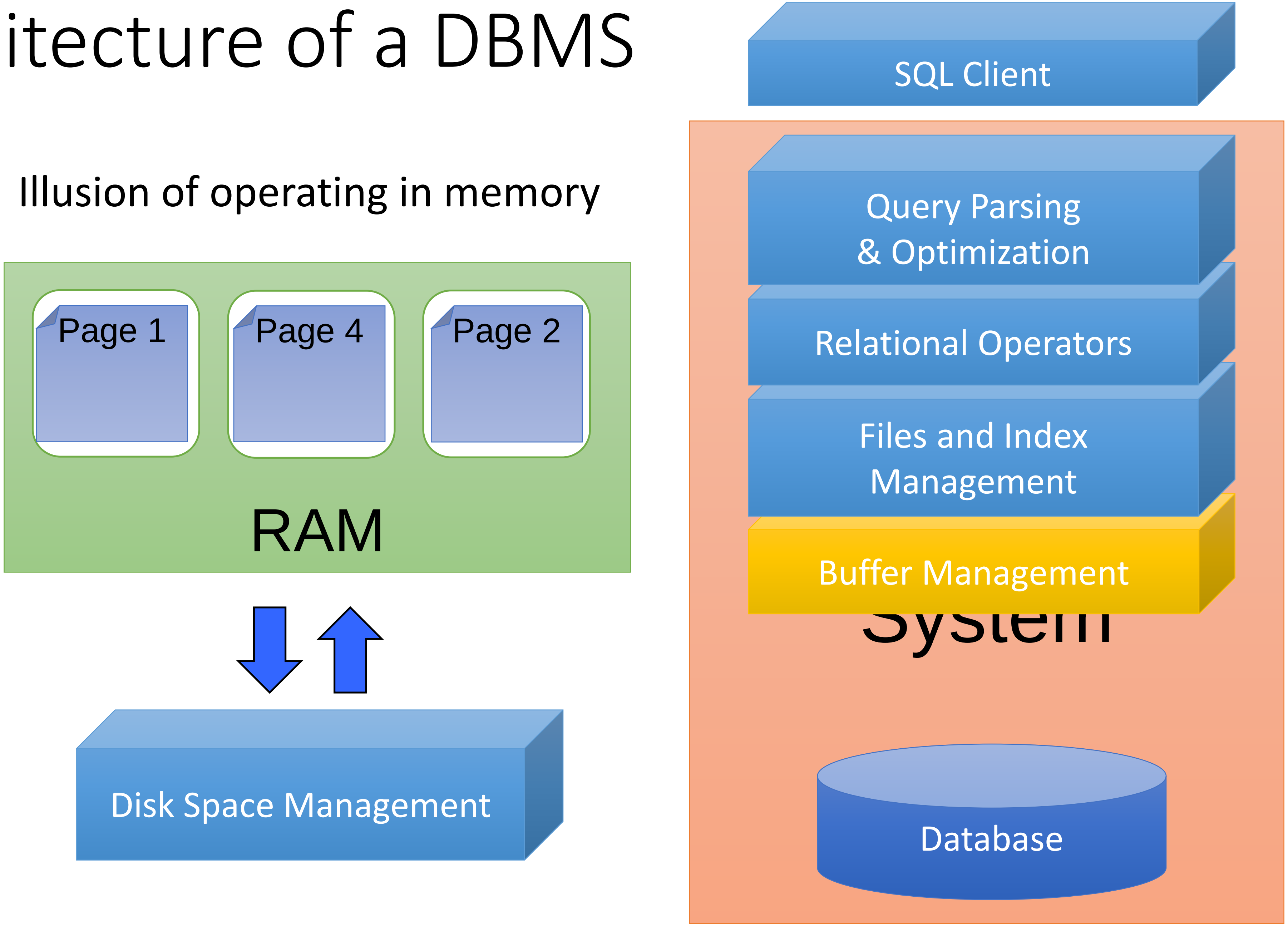
Name	Addr	Sex	Age	Zip
Bob	Harmon	M	32	94703
Alice	Mabel	F	33	94703
Jose	Chavez	M	31	94110
Jane	Chavez	F	30	94110





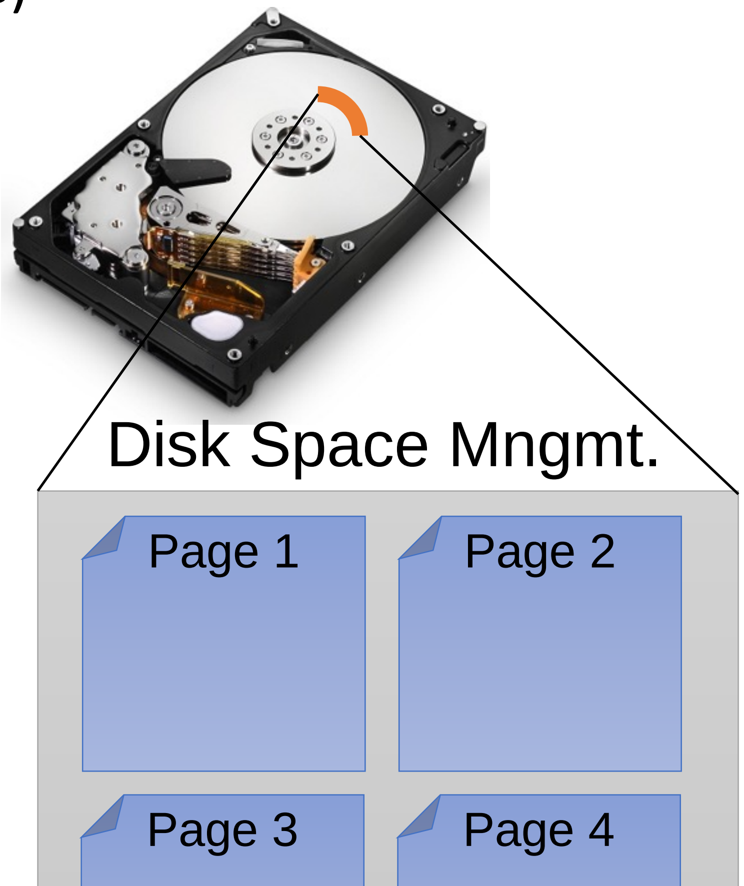
Architecture of a DBMS

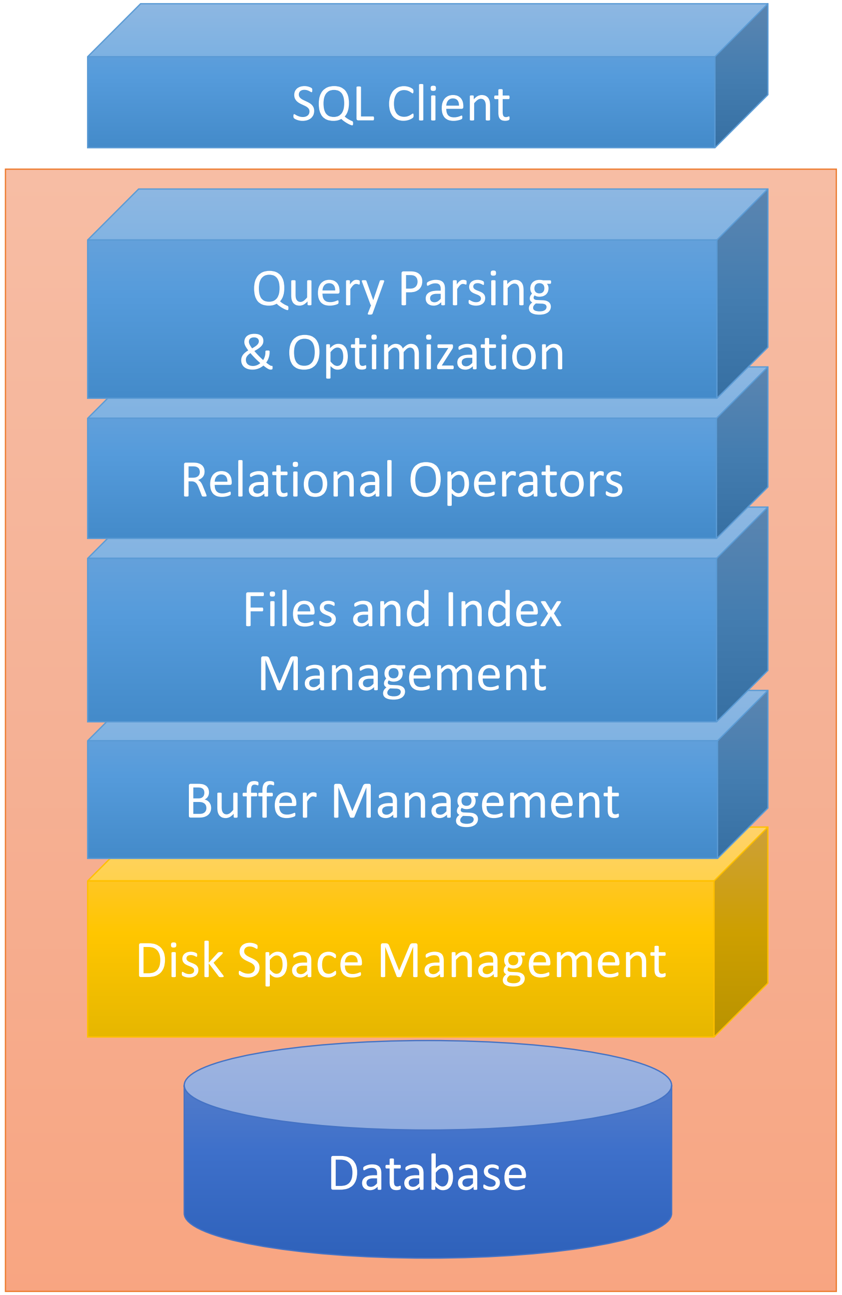
Illusion of operating in memory



Architecture of a DBMS

Translates page requests into physical bytes on one or more device(s)





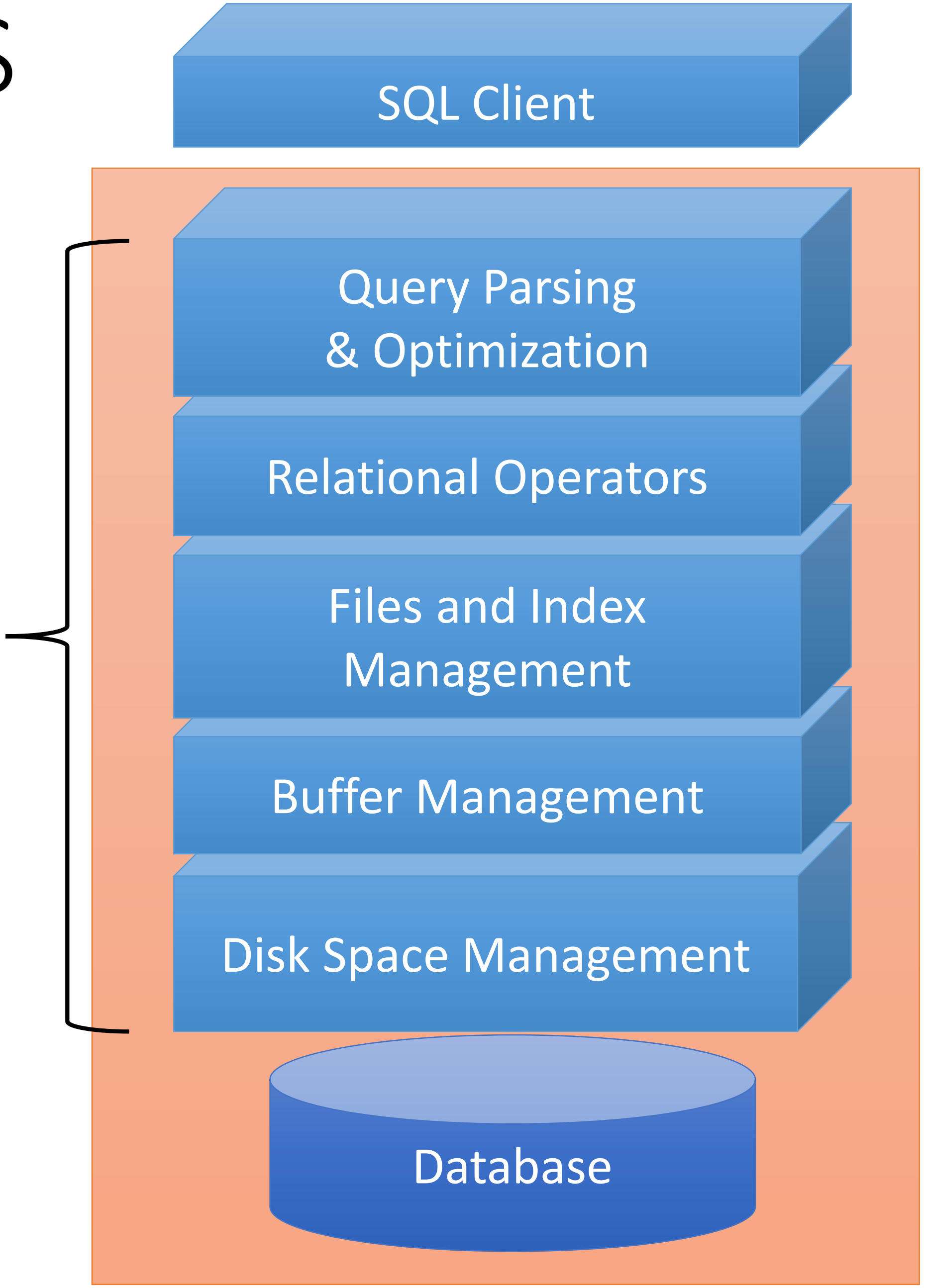
Architecture of a DBMS

Organized in layers

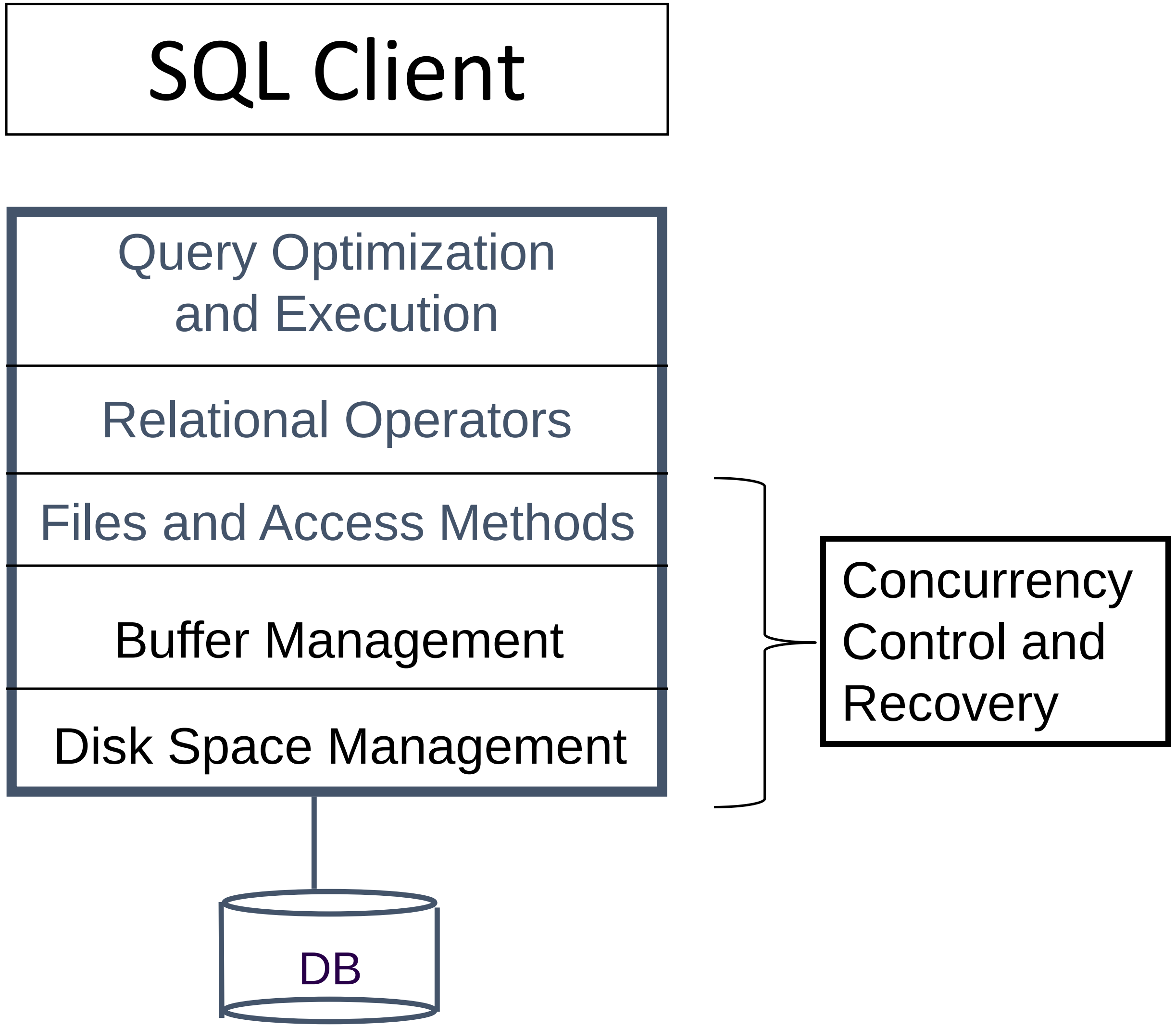
Each layer abstracts the layer below

- Manage complexity
- Perf. Assumptions

Example of **good** systems design



Block diagram of a DBMS



Goals for this and next lecture

- **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

Today’s Lecture

1. Transactions

2. Properties of Transactions: ACID

3. Logging

1. Transactions

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

```
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
```

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

Transaction Properties: ACID

- Atomic
  - State shows either all the effects of txn, or none of them
- Consistent
  - Txn moves from a state where integrity holds, to another where integrity holds
- Isolated
  - Effect of txns is the same as txns running one after another (ie looks like batch mode)
- Durable
  - 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

ACI<u>D</u>: <u>D</u>urability

- 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…

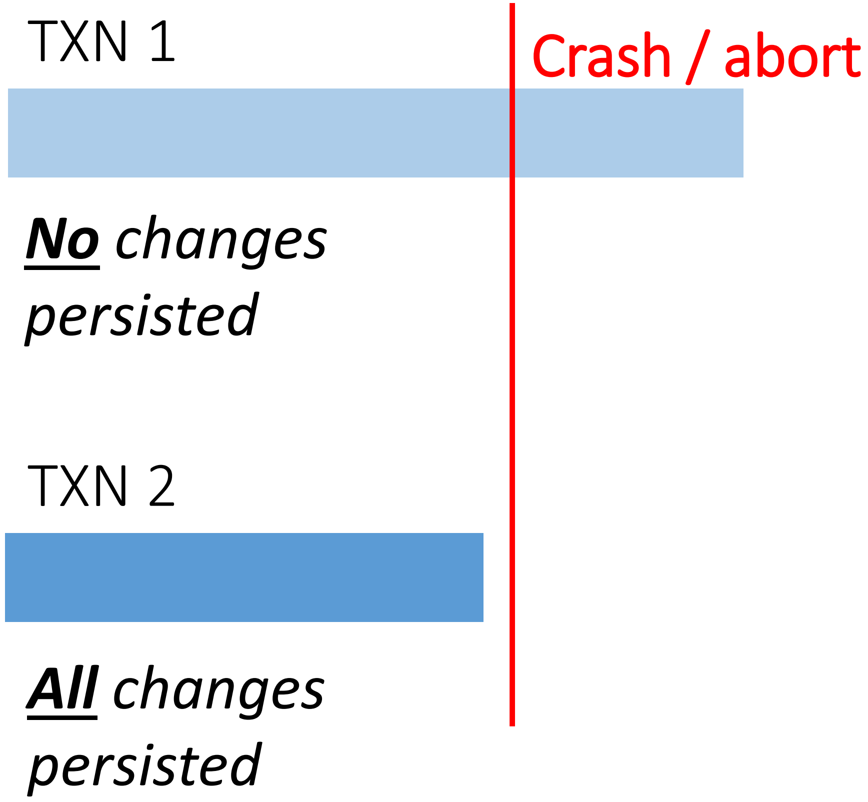
ACID is an extremely important & successful paradigm, but still debated!

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!

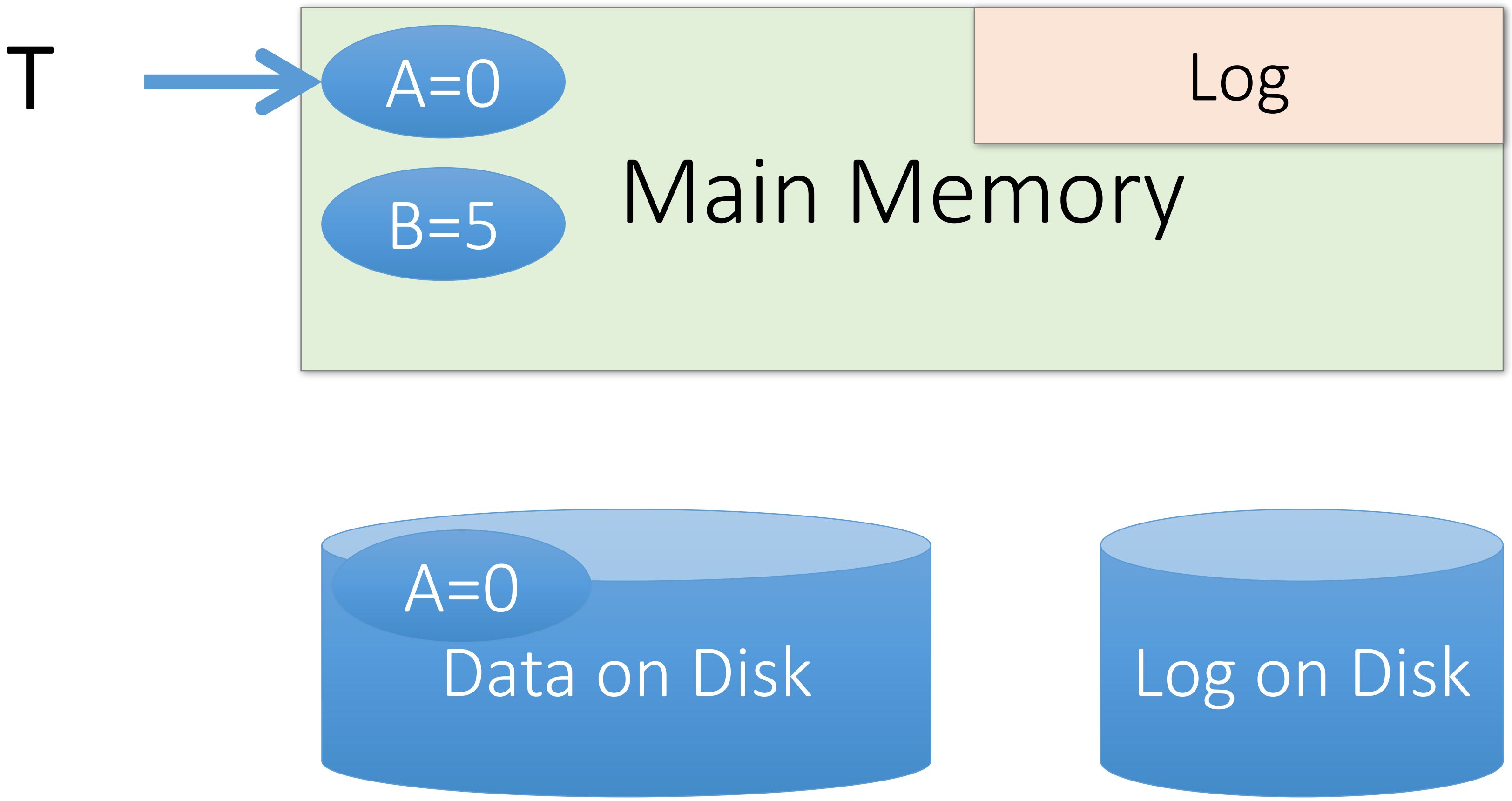
3. Atomicity & Durability via Logging

What you will learn about in this section

1. Logging: An animation of commit protocols

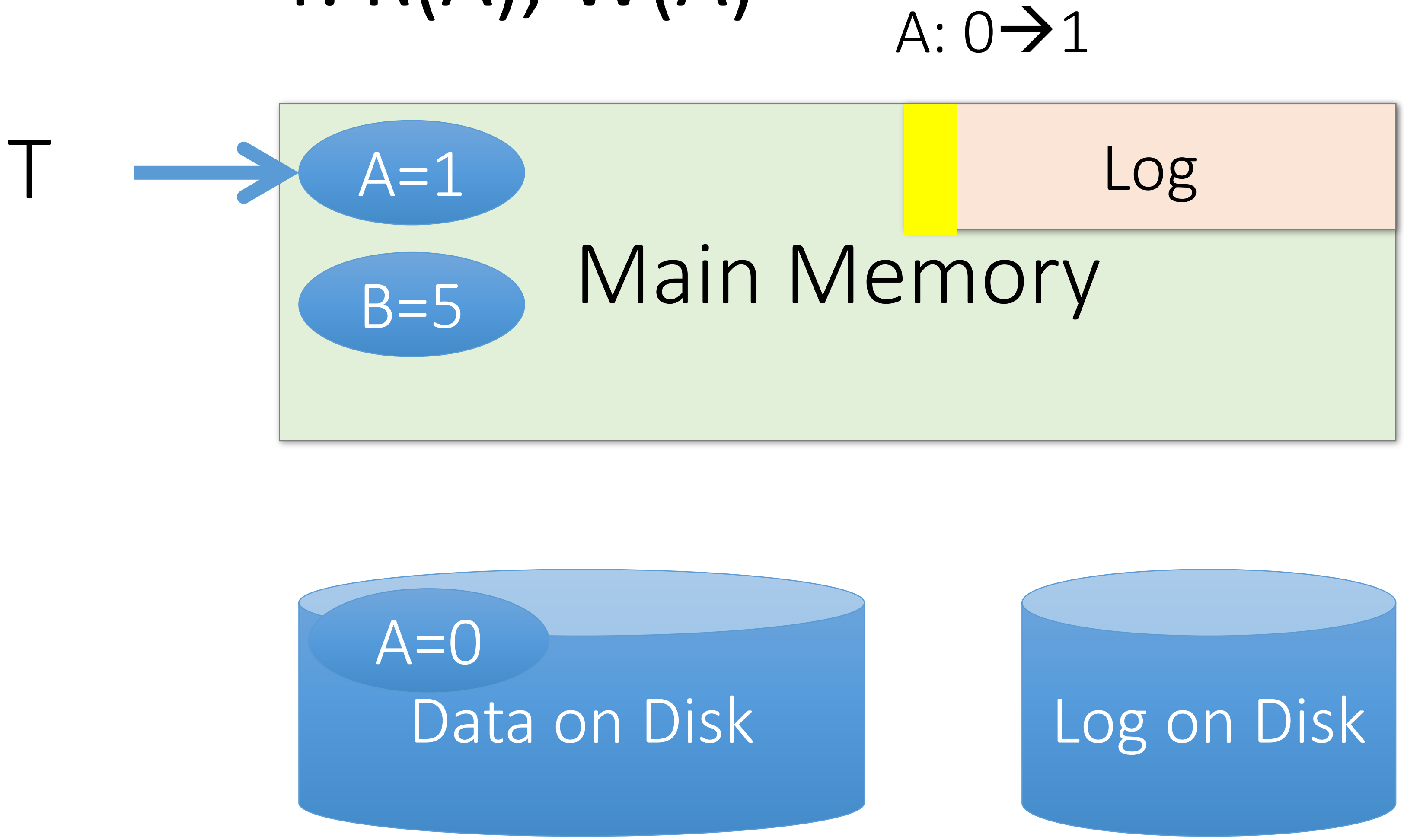
A picture of logging

T: R(A), W(A)



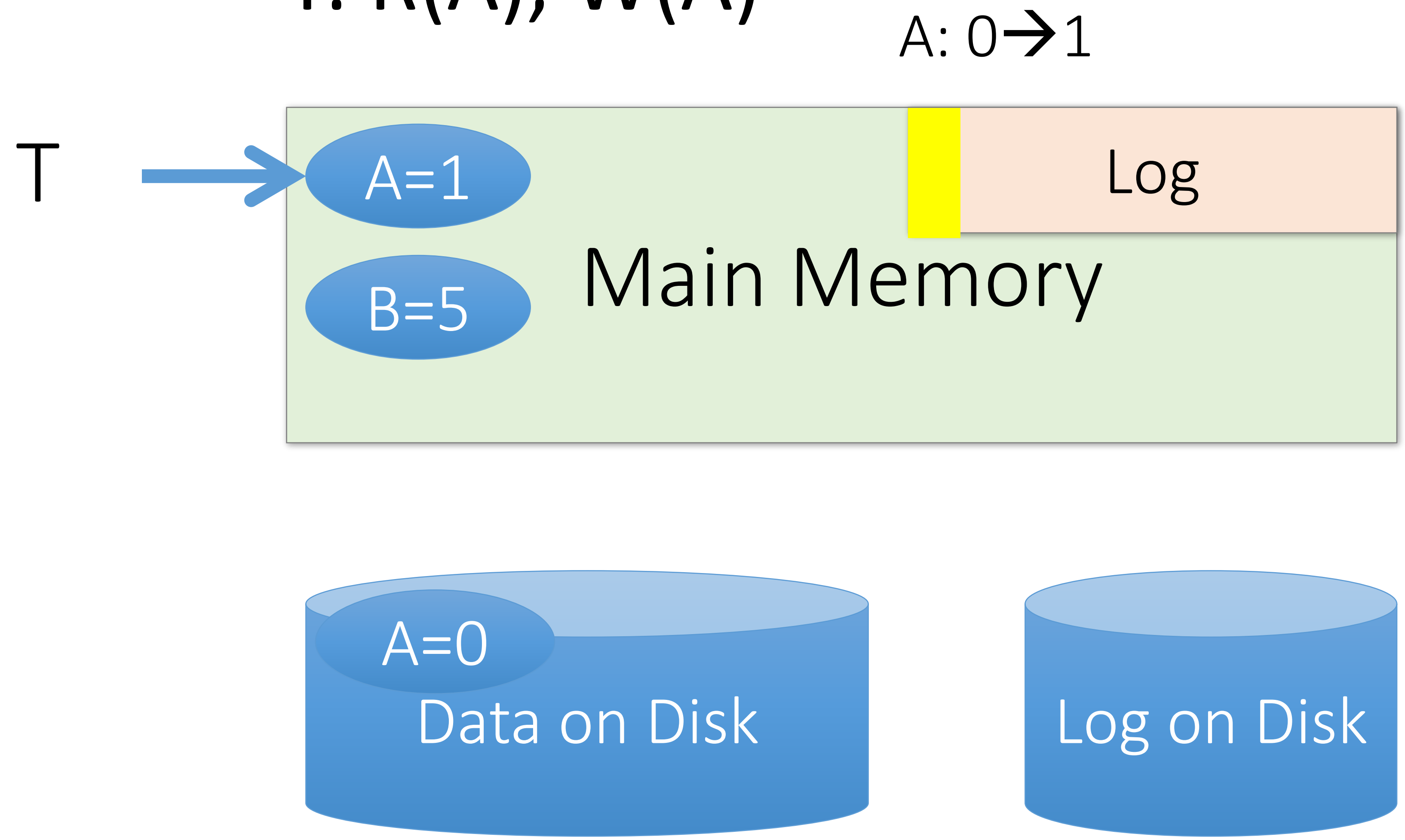
A picture of logging

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

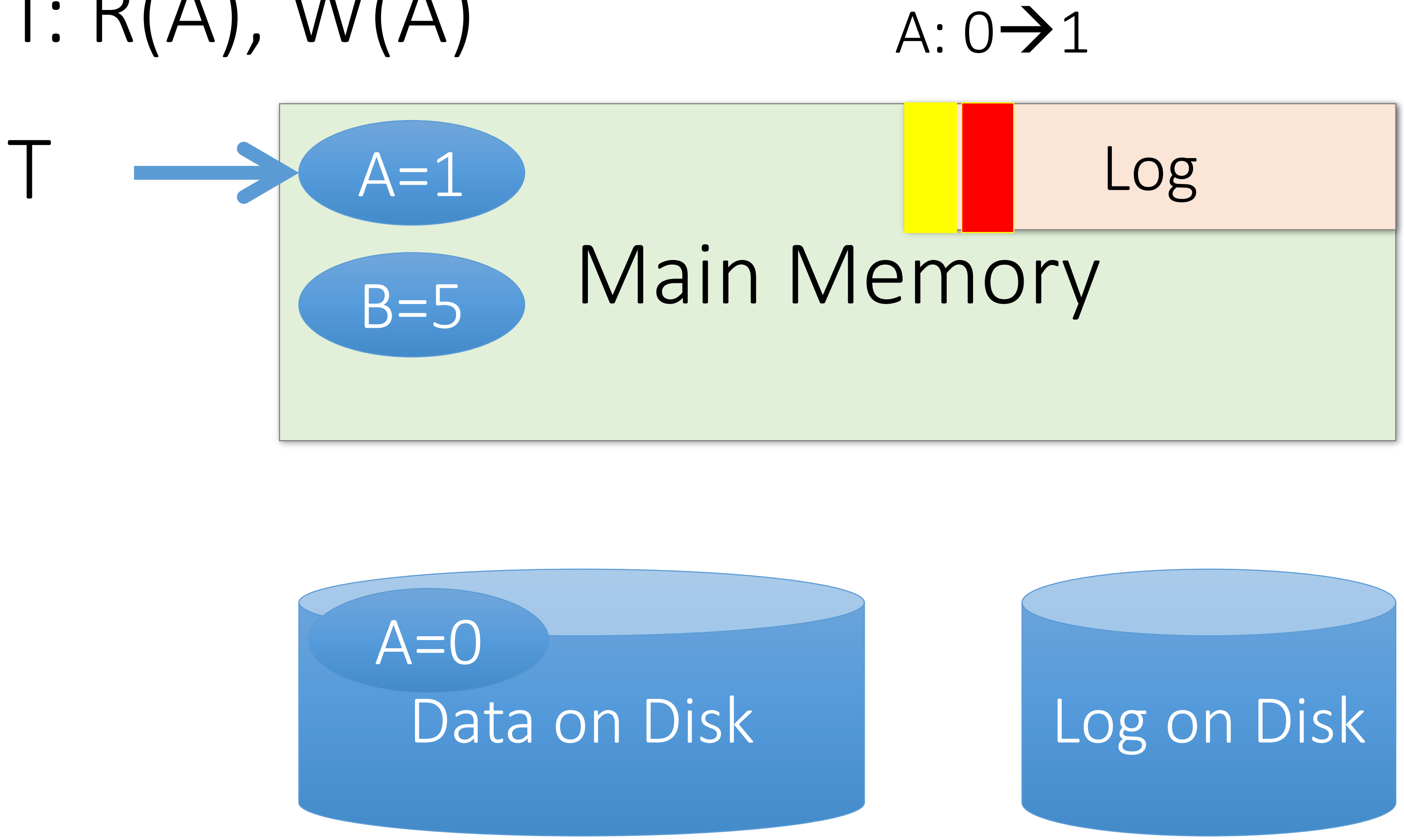
2. All log records up to last update from this TX are FORCED

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…

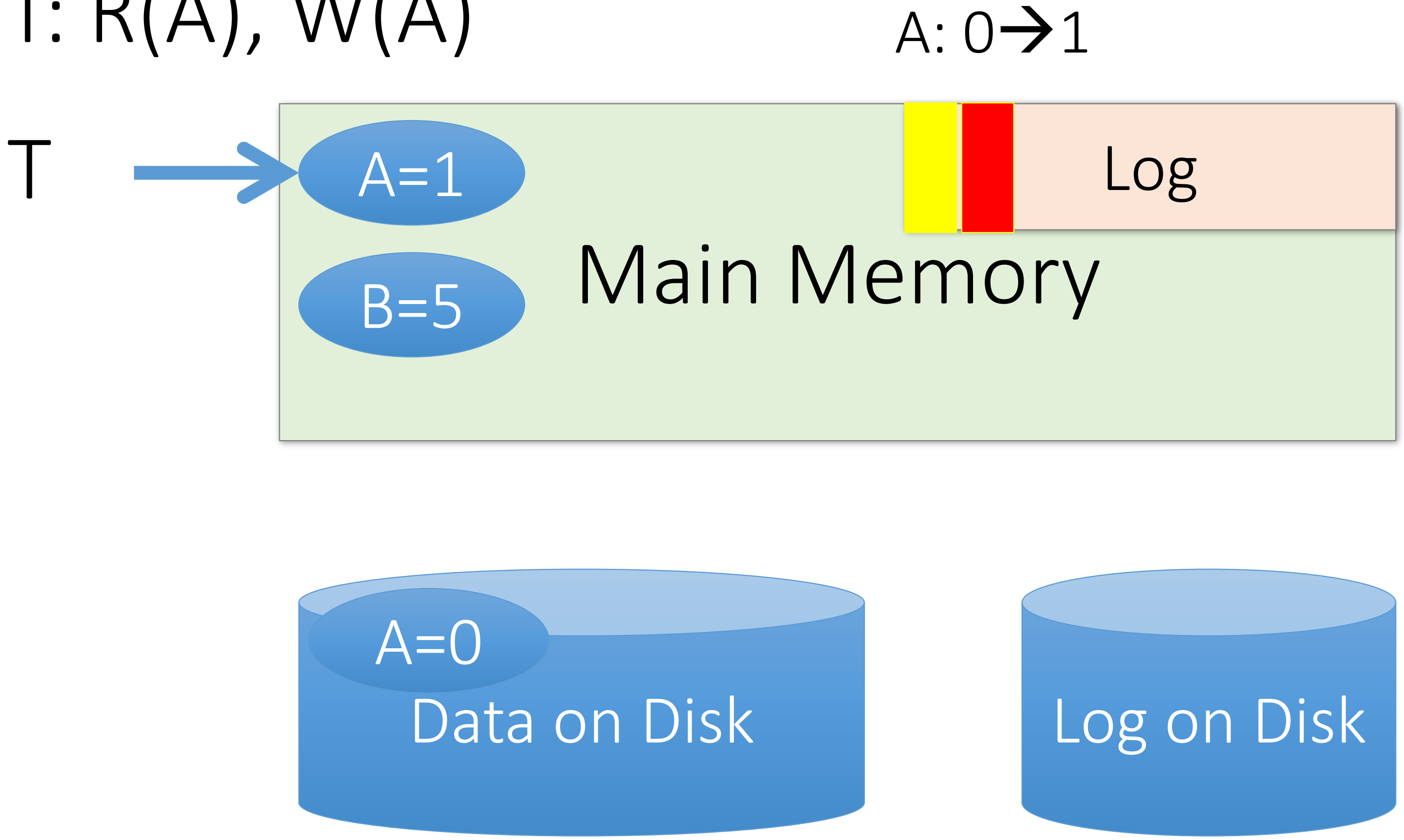
OK, Commit!

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!

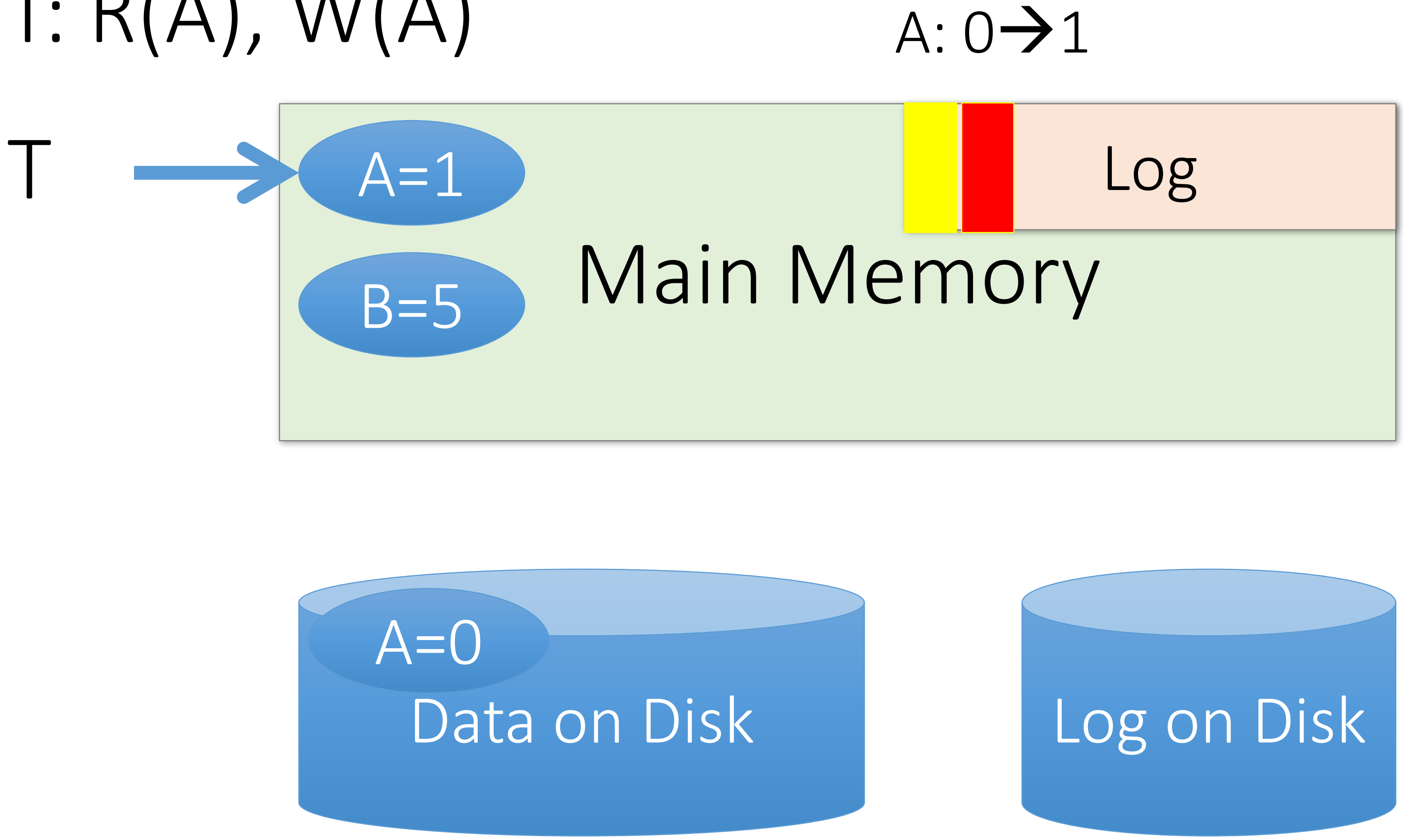
If we crash now, is T durable? Yes! Except…

How do we know whether T was committed??

Improved Commit Protocol (WAL)

Write-ahead Logging (WAL) Commit Protocol

T: R(A), W(A)



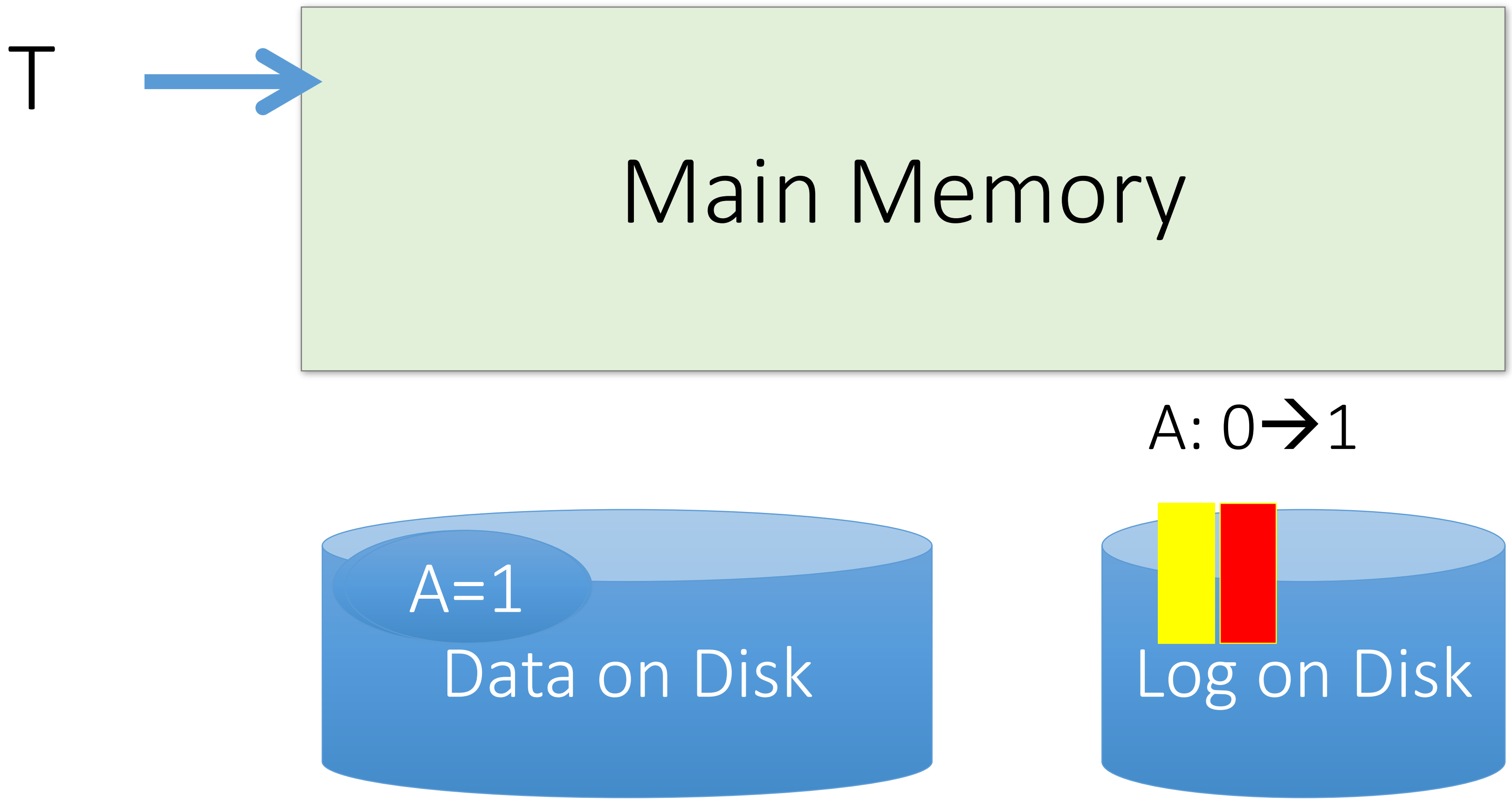
This time, let's try committing ***after*** **we've written log to disk but** ***before*** **we've written data to disk**... this is WAL!

OK, Commit!

If we crash now, is T durable?

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

→ <u>Atomicity</u>

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

→ <u>Durability</u>

I.e. transaction is not committed until all of its log records— including its "commit" record—are on the stable log.

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…