CS4740
CLOUD COMPUTING

Transaction

Prof. Chang Lou, UVA CS, Spring 2024

CONTEXT

- We'll dive into more advanced topics: agreement, consensus..
- But before continuing DS, today we discuss how some DS challenges are solved in simpler, single-node systems.
  - Then we come back to distributed settings.

- Today: Transaction
  - How transaction helps solving key system challenges
  - Implement transaction: Part I

CHALLENGES OF DISTRIBUTED SYSTEMS

CHALLENGES OF DISTRIBUTED SYSTEMS

- Two common challenges of building a distributed system (e.g., database):
  - Handling **failures**: failures are inevitable but they create the potential for partial computations and correctness of computations after restart.
  - Handling **concurrency**: concurrency is vital for performance (e.g., I/O is slow so need to overlap with computation), but it creates races. Need to use some form of synchronization to avoid those.

They are not unique to distributed systems!

CHALLENGES OF SINGLE-NODE SYSTEMS

- Two common challenges of building a single-node system (e.g., database):
  - Handling **failures**: servers may crash or operations may abort anytime.
  - Handling **concurrency**: single-node systems handle concurrent requests to improve throughput.
- Let's start from what we left last class: RPC

EXAMPLE: FLIGHT BOOKING

– Client code:

```
x = server.getFlightAvailability(ABC, 123, date);   // read(ABC, 123, date)
if (x > 0)
    y = server.bookTicket(ABC, 123, date);          // write(ABC, 123, date)
server.putSeat(y, “aisle”);
```

FAILURE: ABORTION

Client 1

```
x = getSeats(ABC123);
y = getSeats(ABC789);
write(x-1, ABC123);

write(y+1, ABC789);
```

← crash/abort!

FAILURE: ABORTION

Client 1

```
x = getSeats(ABC123);
y = getSeats(ABC789);
write(x-1, ABC123);

write(y+1, ABC789);
```

crash/abort!

new ticket was booked but old ticket was not returned!

Client 1

```
x = getSeats(ABC123);
  // x = 10
if(x > 1)
  x = x – 1;
write(x, ABC123);
```

Client 2

```
x = getSeats(ABC123);
if(x > 1) //x = 10


  x = x – 1;
write(x, ABC123);
```

At Server: seats = 10

seats = 9

seats = 9

CONCURRENCY: 1. LOST UPDATE PROBLEM

Client 1

```
x = getSeats(ABC123);
  // x = 10
if(x > 1)
  x = x – 1;
write(x, ABC123);
```

Client 2

```
x = getSeats(ABC123);
if(x > 1) //x = 10


  x = x – 1;
write(x, ABC123);
```

At Server: seats = 10

C1's or C2's update was lost!

seats = 9

seats = 9

Client 1

```
x = getSeats(ABC123);
y = getSeats(ABC789);
write(x-5, ABC123);
  // ABC123 = 5 now

write(y+5, ABC789);
```

Client 2

```
x = getSeats(ABC123);
y = getSeats(ABC789);
  // x = 5, y = 15
print("Total:" x+y);
  // Prints "Total: 20"
```

At Server:
ABC123 = 10
ABC789 = 15

CONCURRENCY: 2. INCONSISTENT RETRIEVAL PROBLEM

Client 1

```
x = getSeats(ABC123);
y = getSeats(ABC789);
write(x-5, ABC123);
  // ABC123 = 5 now

write(y+5, ABC789);
```

Client 2

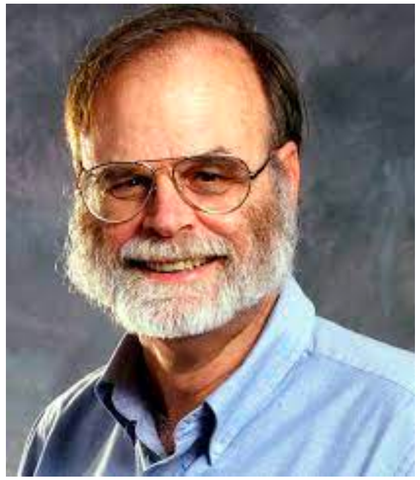
```
x = getSeats(ABC123);
y = getSeats(ABC789);
  // x = 5, y = 15
print("Total:" x+y);
  // Prints "Total: 20"
```

At Server:
ABC123 = 10
ABC789 = 15

C2's sum is the wrong value! Should have been "Total: 25"

TRANSACTION

- Turing-award-winning idea.
- Abstraction provided to programmers that encapsulates a unit of work against a database.
- Guarantees that the unit of work is executed atomically in the face of failures and is isolated from concurrency.



Jim Gray (1944-2012)

TRANSACTION API

– Simple but very powerful:

```
txID = Begin()          // Starts a transaction. Returns a unique ID for the
                        // transaction.
```

```
outcome= Commit(txID)  // Attempts to commit a transaction; returns
                       // whether or not the commit was successful. If
                       // successful, all operations in the transaction
                       // have been applied to the DB. If unsuccessful,
                       // none of them has been applied.
```

```
Abort(txID)             // Cancels all operations of a transaction and erases
                        // their effects on the DB. Can be invoked by the
                        // programmer or by the database engine itself.
```

SEMANTICS

- By wrapping a set of accesses in a transaction, the database can hide failures and concurrency under meaningful guarantees.
- One such set of guarantees is ACID:
  - Atomicity: Either all operations in the transaction will complete successfully (commit outcome), or none of them will (abort outcome), regardless of failures.
  - Isolation: A transaction's behavior is not impacted by the presence of concurrently executing transactions.
  - Durability: The effects of committed transactions survive failures.

hide failures

hide concurrency

EXAMPLE

Transaction T1

```
begin
x = getSeats(ABC123);
y = getSeats(ABC789);
write(x-1, ABC123);
write(y+1, ABC789);

commit
```

EXAMPLE

Transaction T1

```
begin
x = getSeats(ABC123);
y = getSeats(ABC789);
write(x-1, ABC123);
write(y+1, ABC789);

commit
```

Atomicity: both writes succeed, or neither

EXAMPLE

Transaction T1

```
begin
x = getSeats(ABC123);
y = getSeats(ABC789);
write(x-1, ABC123);
write(y+1, ABC789);

commit
```

Atomicity: both writes succeed, or neither

Durability: writes are persisted and recoverable after failures

EXAMPLE

Transaction T1

```
begin
x = getSeats(ABC123);
y = getSeats(ABC789);
write(x-1, ABC123);
write(y+1, ABC789);

commit
```

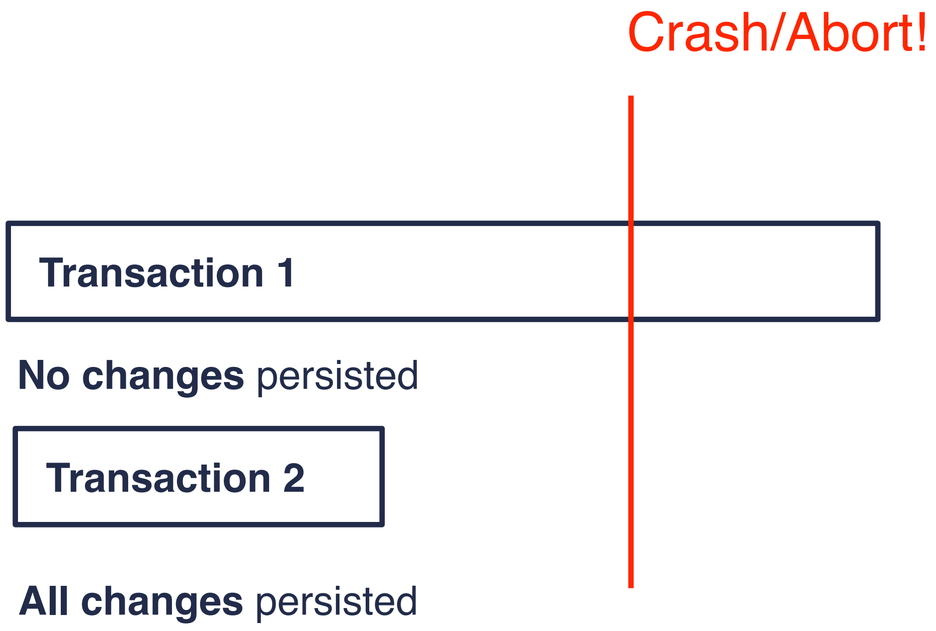
Atomicity: both writes succeed, or neither

Durability: writes are persisted and recoverable after failures

Isolation: T1 don't get affected by other transactions

HOW TO IMPLEMENT TRANSACTIONS?

- Atomicity and Durability
  - Key mechanism: **write-ahead logging**
- Isolation



HOW TO MAKE UPDATES DURABLE

Transaction T1

```
begin
x = getSeats(ABC123);
y = getSeats(ABC789);
write(x-1, ABC123);
write(y+1, ABC789);

commit
```

x = 10

y = 10

Disk: x = 9

Disk: y = 11

HOW TO MAKE UPDATES DURABLE

– Write updates to disk!

Transaction T1
begin
x = getSeats(ABC123);
y = getSeats(ABC789);
write(x-1, ABC123); → write to disk!
write(y+1, ABC789); → write to disk!

commit

x = 10

y = 10

Disk: x = 9

Disk: y = 11

HOW TO MAKE UPDATES ATOMIC

– Write updates to disk!

Transaction T1
begin
x = getSeats(ABC123);
y = getSeats(ABC789);
write(x-1, ABC123);

write(y+1, ABC789);
commit

x = 10

y = 10

write to disk!

crash!

Disk: x = 9

HOW TO MAKE UPDATES ATOMIC

– Write updates to disk!

Transaction T1
begin
x = getSeats(ABC123);
y = getSeats(ABC789);
write(x-1, ABC123);

write(y+1, ABC789);
commit

x = 10

y = 10

write to disk!

crash!

Disk: x = 9

now x needs to be reverted to old value (UNDO)! But how do we know?

HOW ABOUT WRITING DATA TO DISK WHEN COMMITTING

— Write updates to disk!

Transaction T1

```
begin
x = getSeats(ABC123);
y = getSeats(ABC789);
write(x-1, ABC123);

write(y+1, ABC789);
commit
```

crash!

write to disk!

x = 10

y = 10

HOW ABOUT WRITING DATA TO DISK WHEN COMMITTING

— Write updates to disk!

Transaction T1
begin
x = getSeats(ABC123);
y = getSeats(ABC789);
write(x-1, ABC123);

write(y+1, ABC789);
commit

crash!

write to disk!

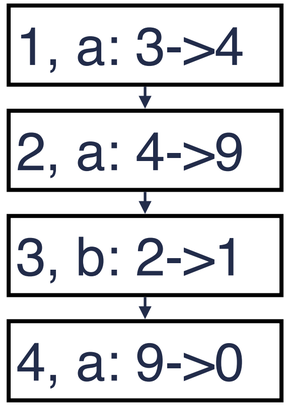
x = 10

y = 10

writing large pages directly to disk is very slow, which blocks the return of "commit"

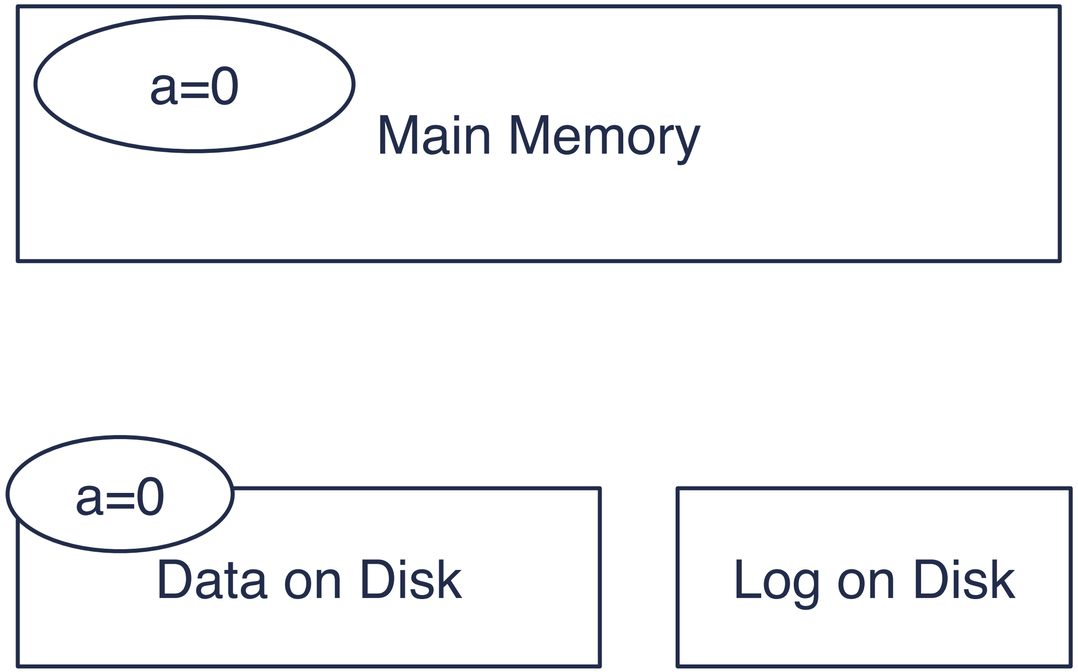
BASIC IDEA: LOGGING

- Record UNDO/REDO information, for every update, in a log
  - Sequential writes to log (put in one a separate disk)
  - Minimal diff written to log, so multiple updates fit in a single log page
- Log: An ordered list of UNDO/REDO actions
  - Log record: <XID, location, old data, new data>

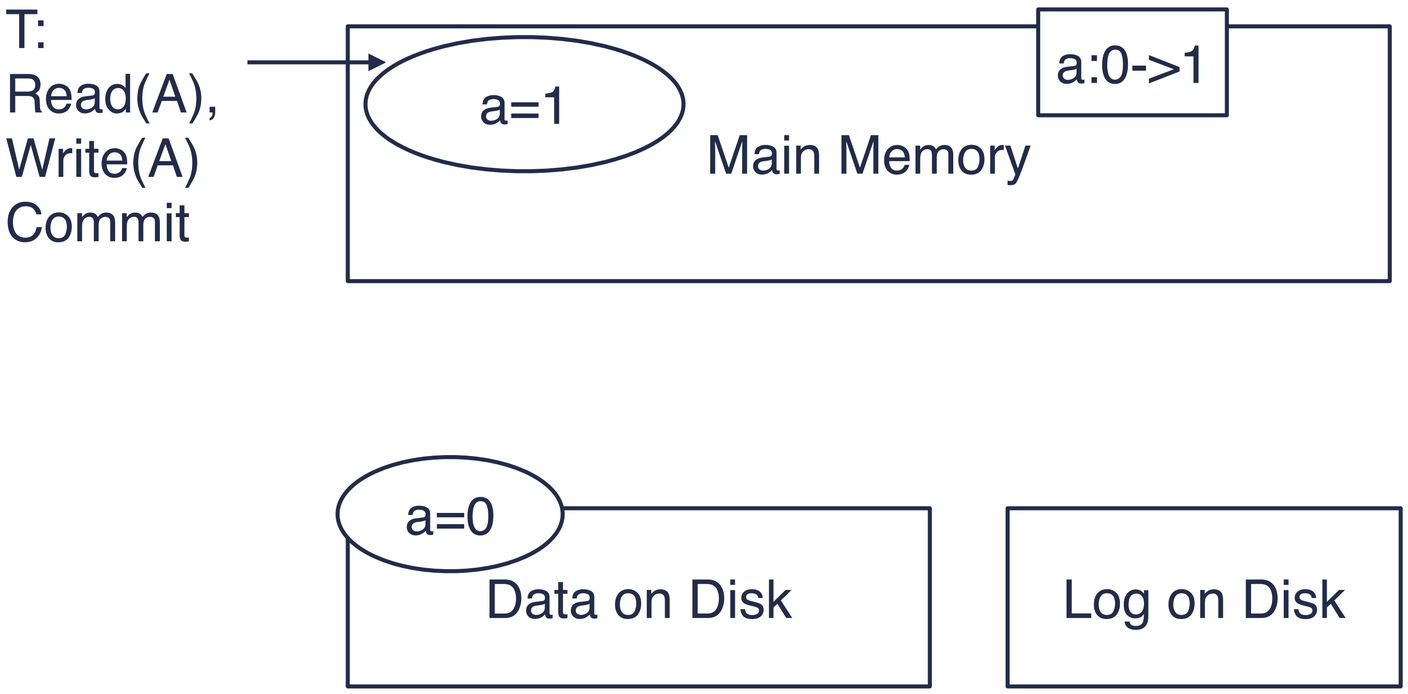


logs

WHY SIMPLE LOGGING NOT WORKING?

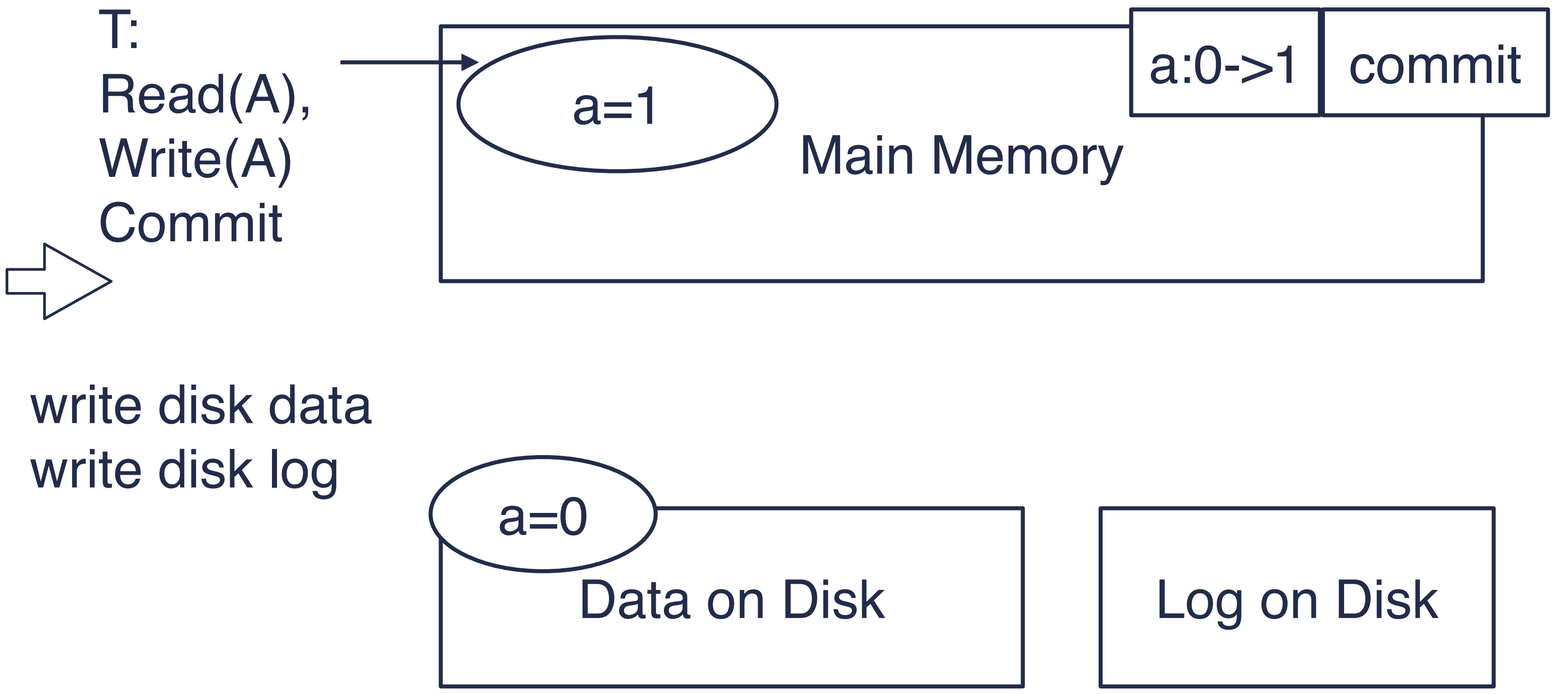


WHY SIMPLE LOGGING NOT WORKING?



Option1: committing **before** we've written either data or log to disk…

WHY SIMPLE LOGGING NOT WORKING?

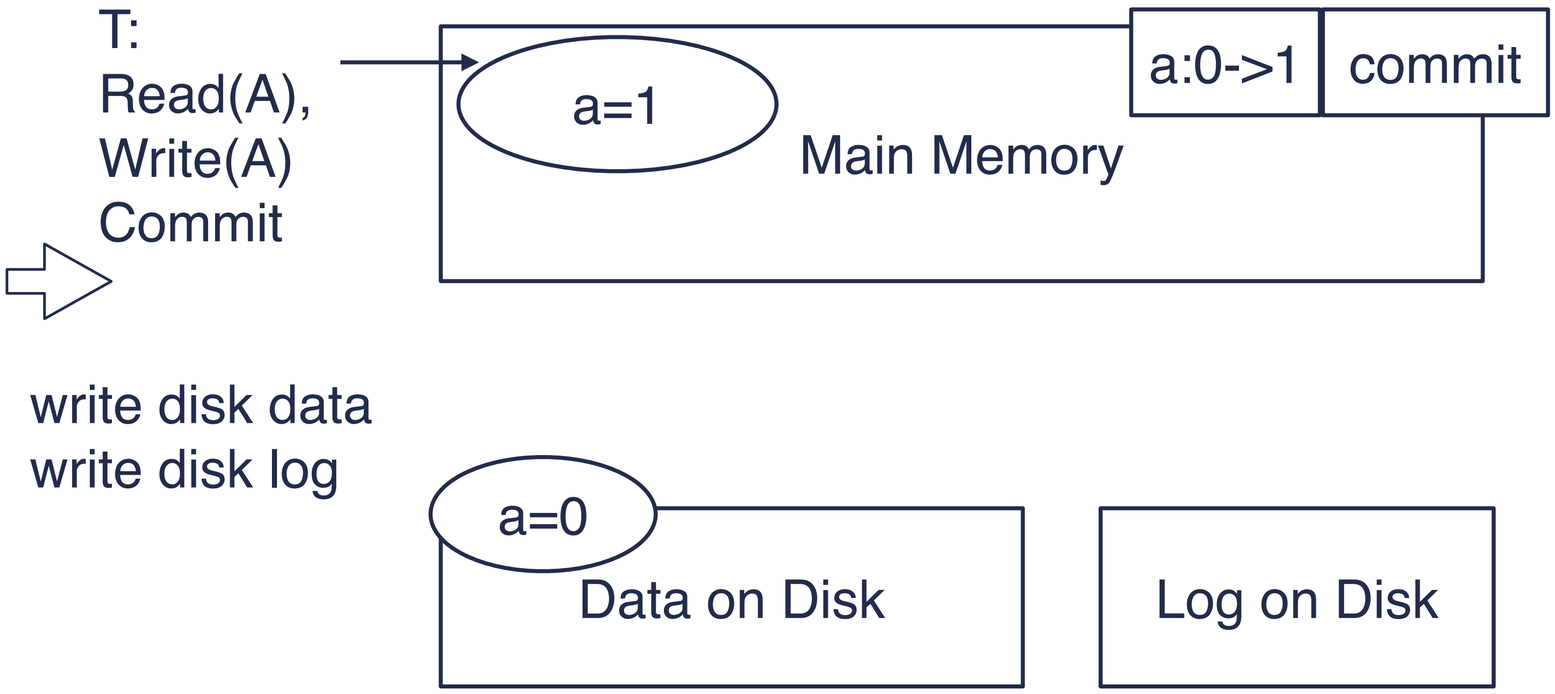


Option1: committing **before** we've written either data or log to disk…

crash!

what happened?

WHY SIMPLE LOGGING NOT WORKING?



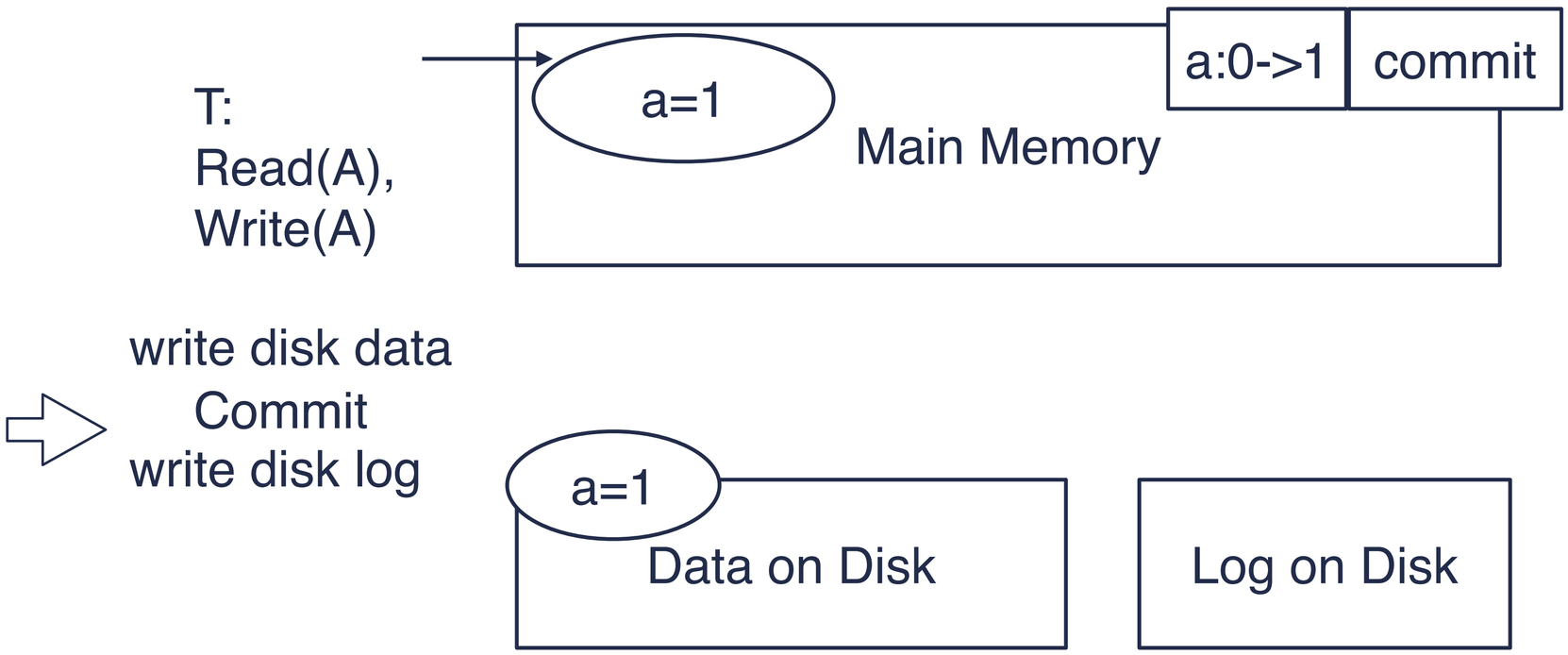
Option1: committing **before** we've written either data or log to disk…

crash!

what happened?

Lost T's update!

WHY SIMPLE LOGGING NOT WORKING?

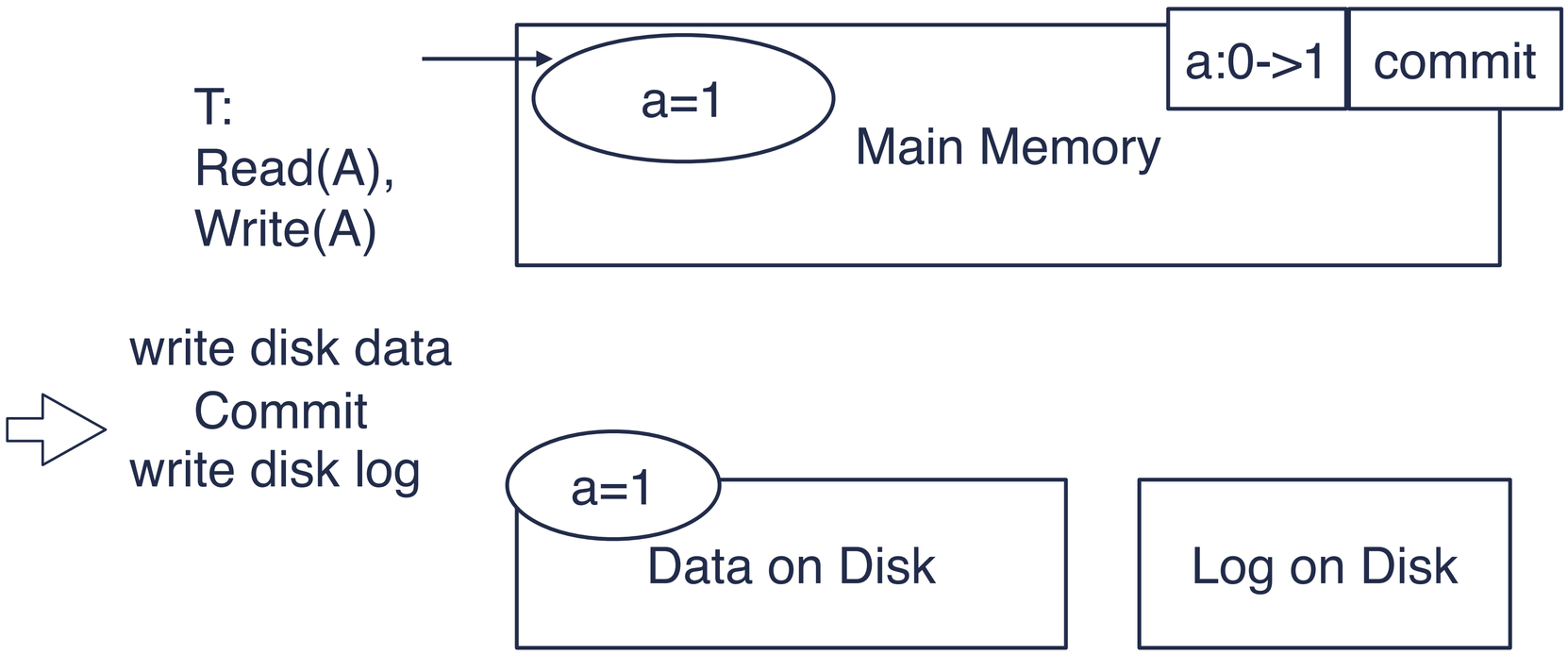


Option2: committing **after** we've written data but **before** we've written log to disk…

crash!

what happened?

WHY SIMPLE LOGGING NOT WORKING?



Option2: committing **after** we've written data but **before** we've written log to disk…

crash!

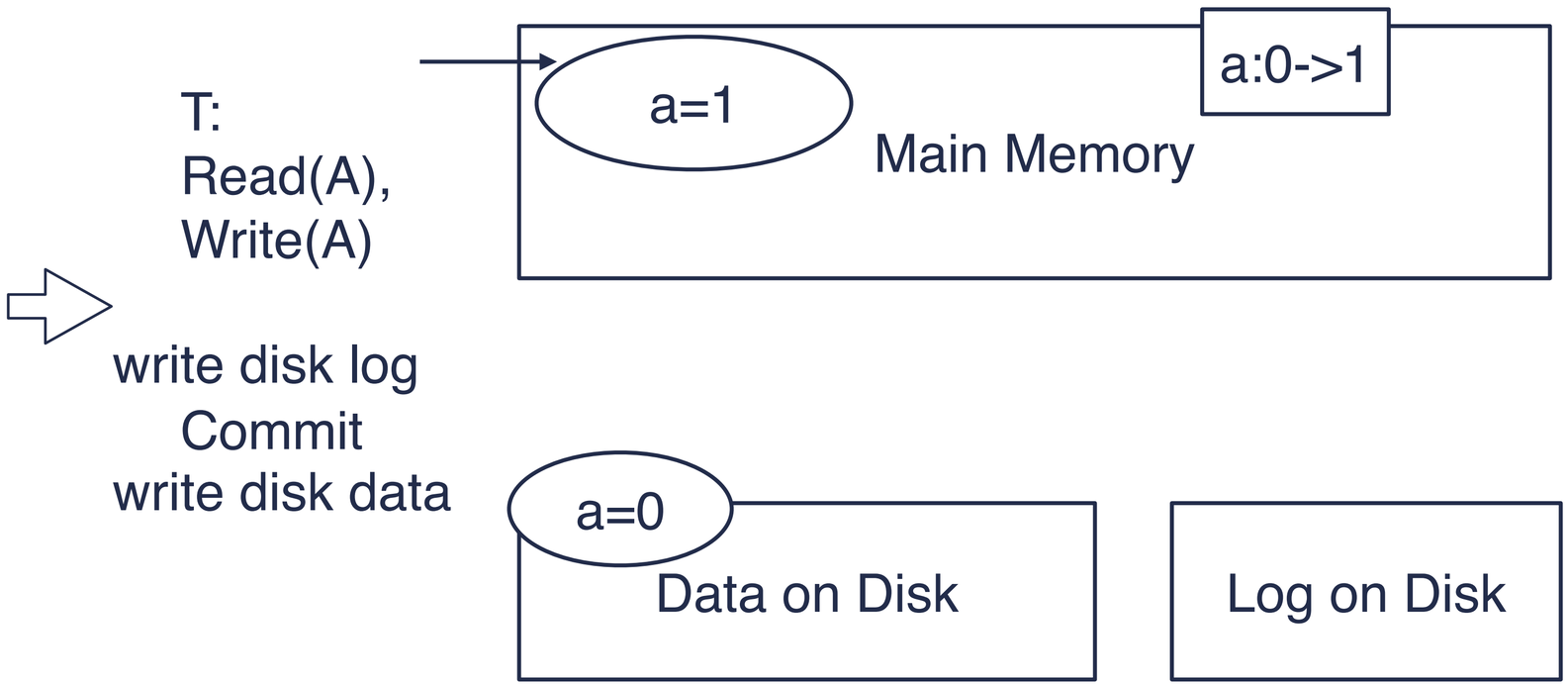
what happened?

How do we know whether T was committed??

WRITE-AHEAD LOGGING (WAL)

- The Write-Ahead Logging Protocol:
  - Write logs to disk and return to the client
  - Write the data to disk asynchronously

WRITE-AHEAD LOGGING (WAL)

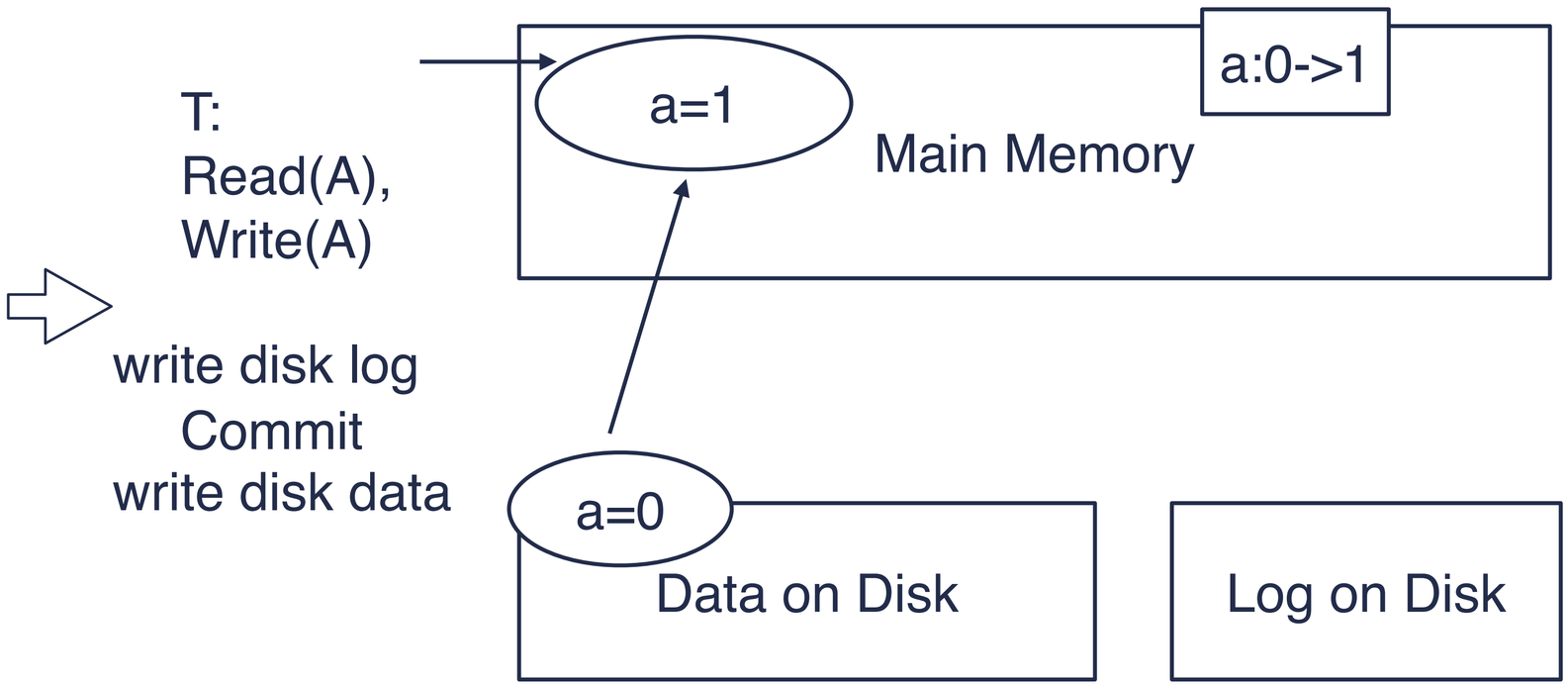


WAL: committing **after** we've written log to disk but **before** we've written data to disk

crash!

what happened?

WRITE-AHEAD LOGGING (WAL)



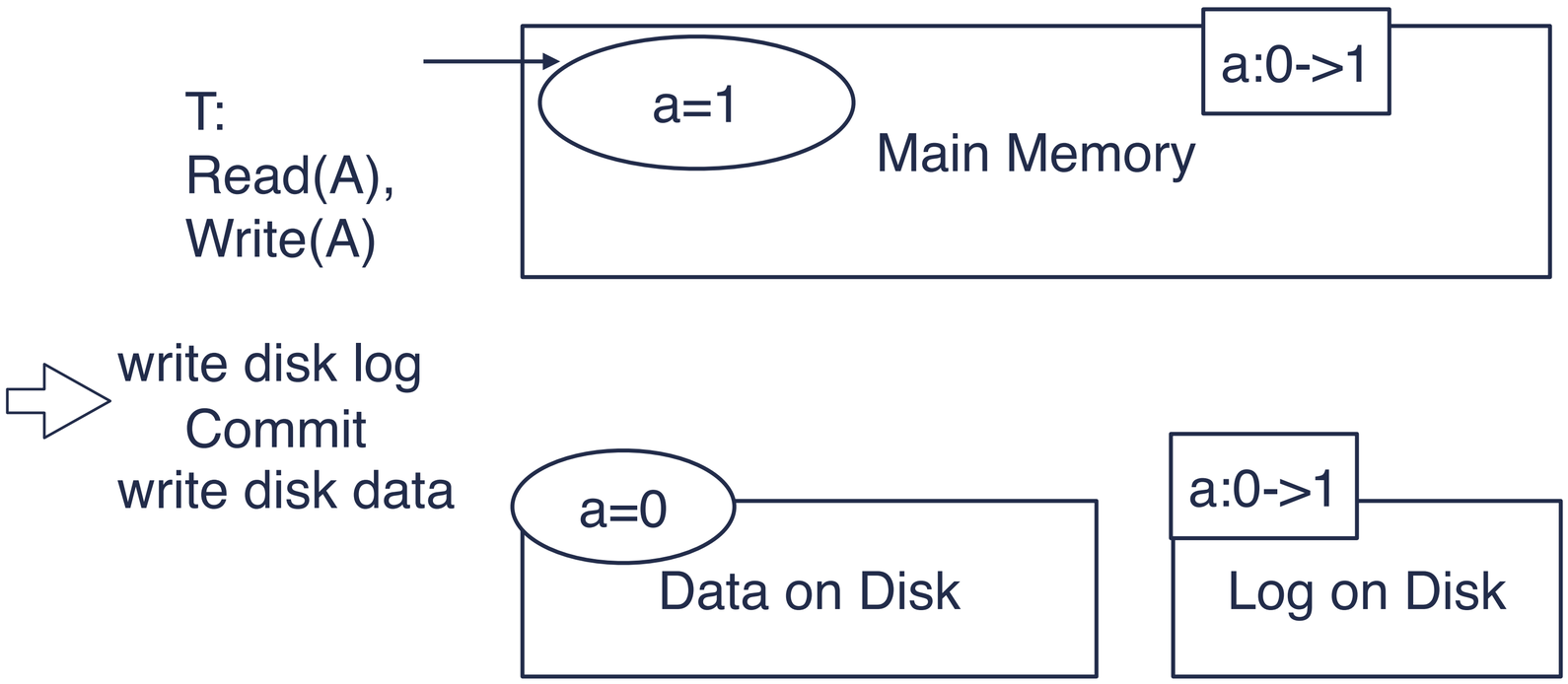
WAL: committing **after** we've written log to disk but **before** we've written data to disk

crash!

what happened?

read data from disk

WRITE-AHEAD LOGGING (WAL)

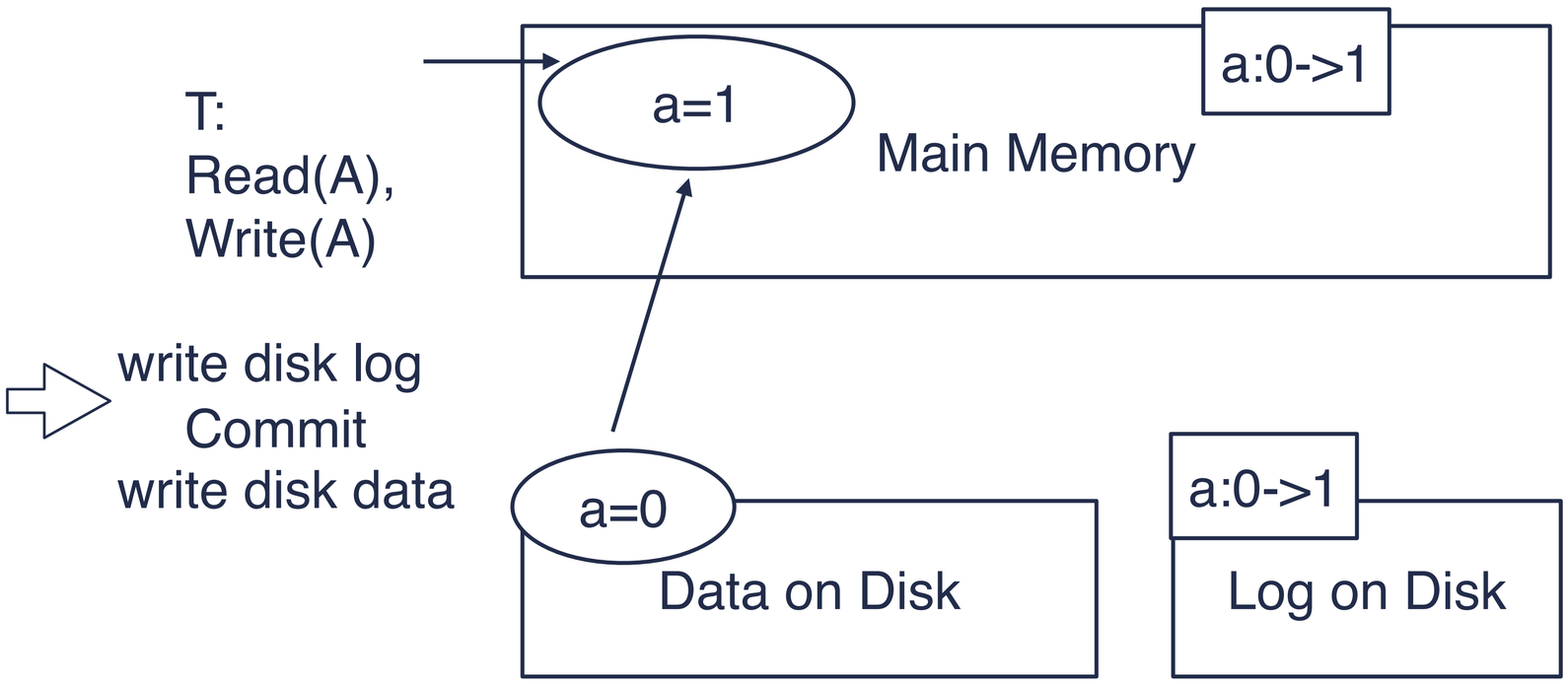


WAL: committing **after** we've written log to disk but **before** we've written data to disk

crash!

what happened?

WRITE-AHEAD LOGGING (WAL)



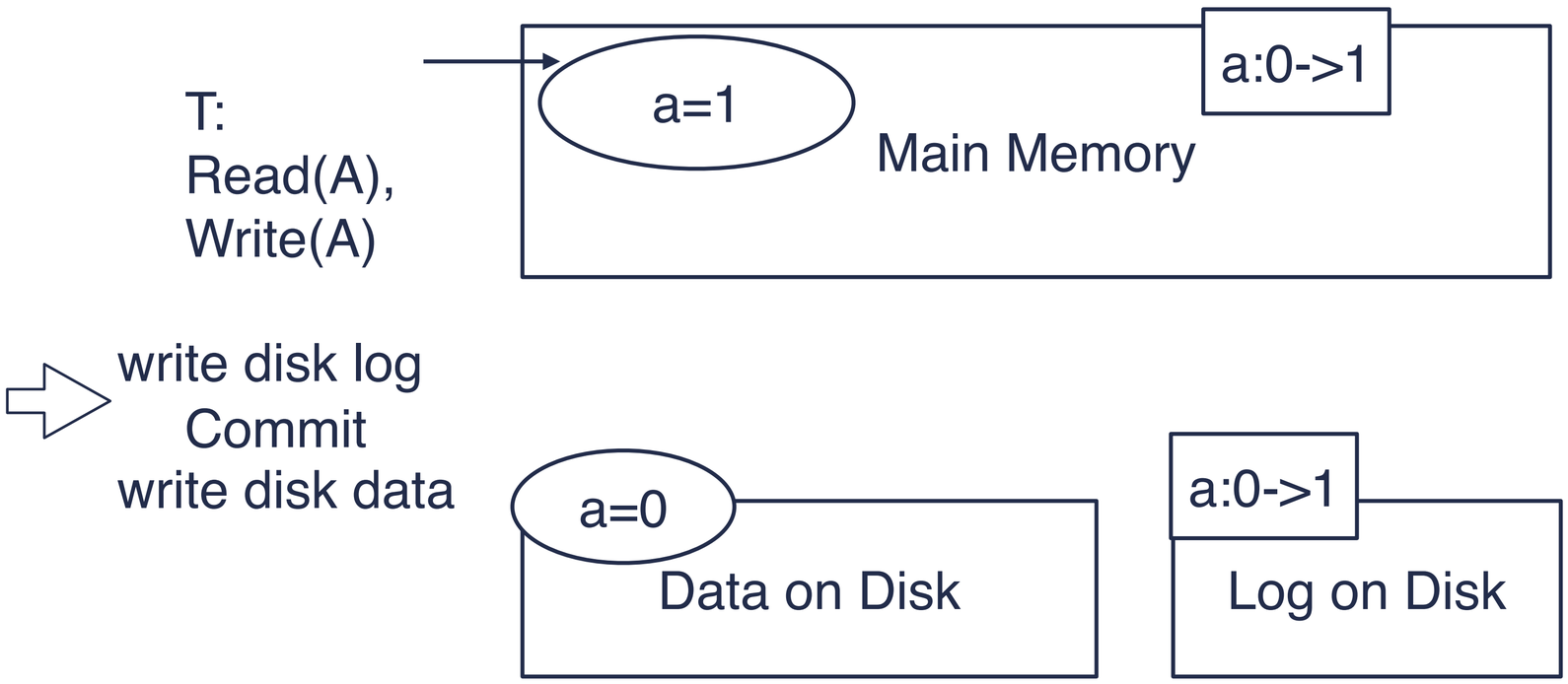
WAL: committing **after** we've written log to disk but **before** we've written data to disk

crash!

what happened?

read data from disk

WRITE-AHEAD LOGGING (WAL)

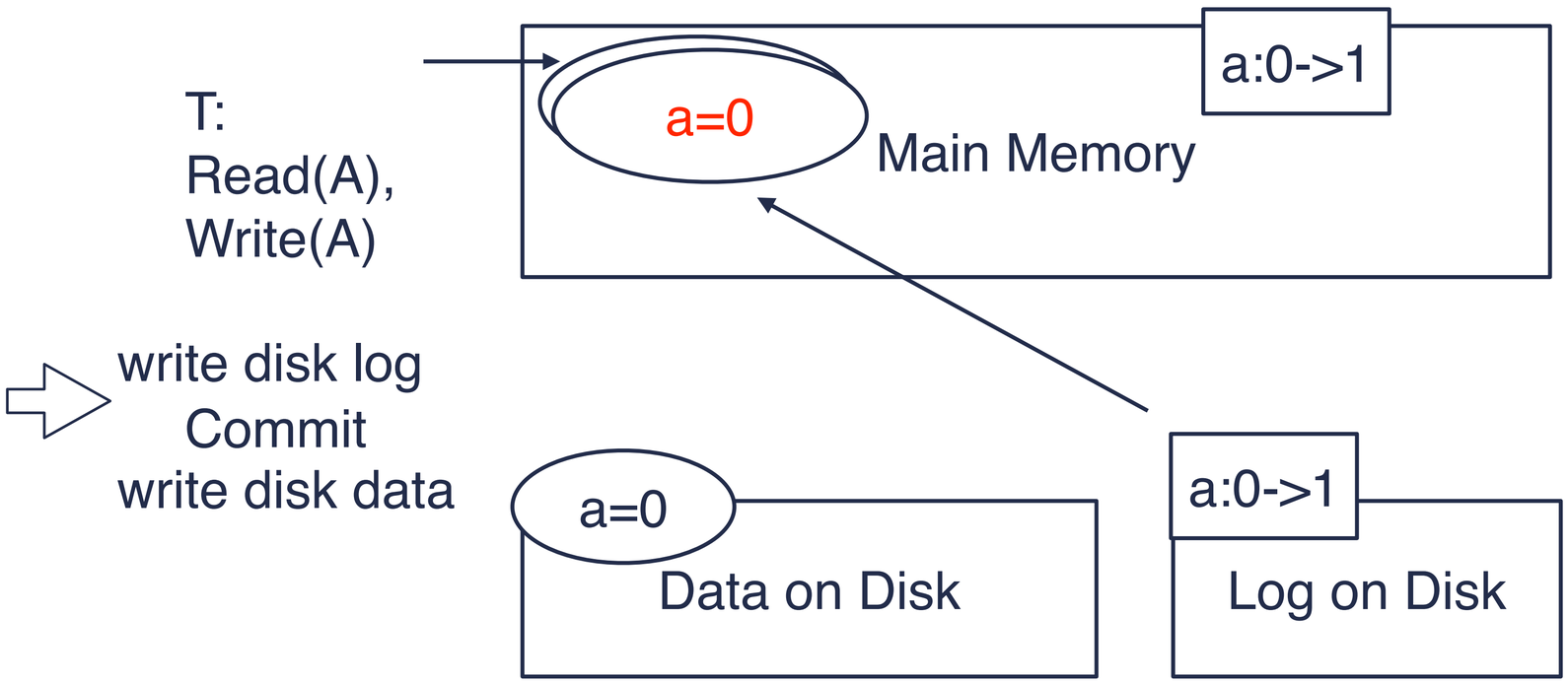


WAL: committing **after** we've written log to disk but **before** we've written data to disk

abort!

what happened?

WRITE-AHEAD LOGGING (WAL)



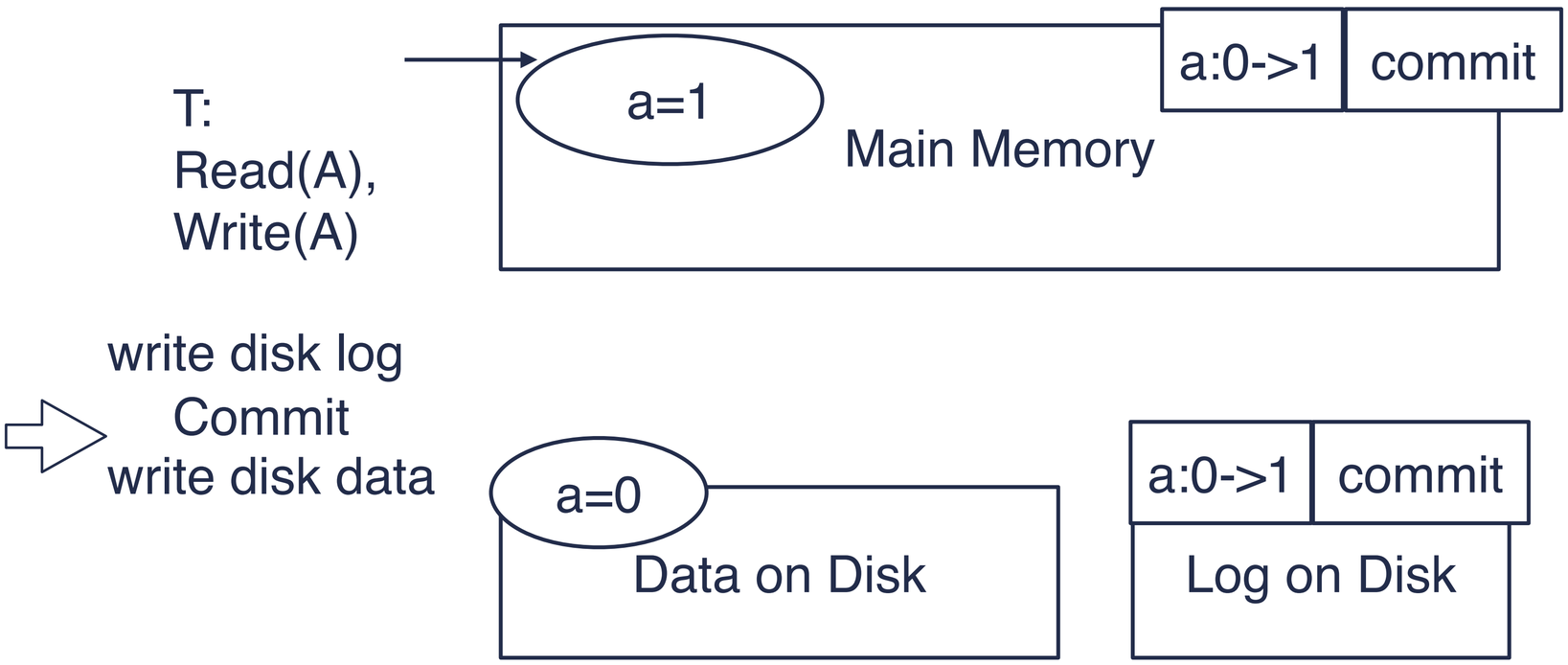
WAL: committing **after** we've written log to disk but **before** we've written data to disk

abort!

what happened?

Just UNDO the log

WRITE-AHEAD LOGGING (WAL)

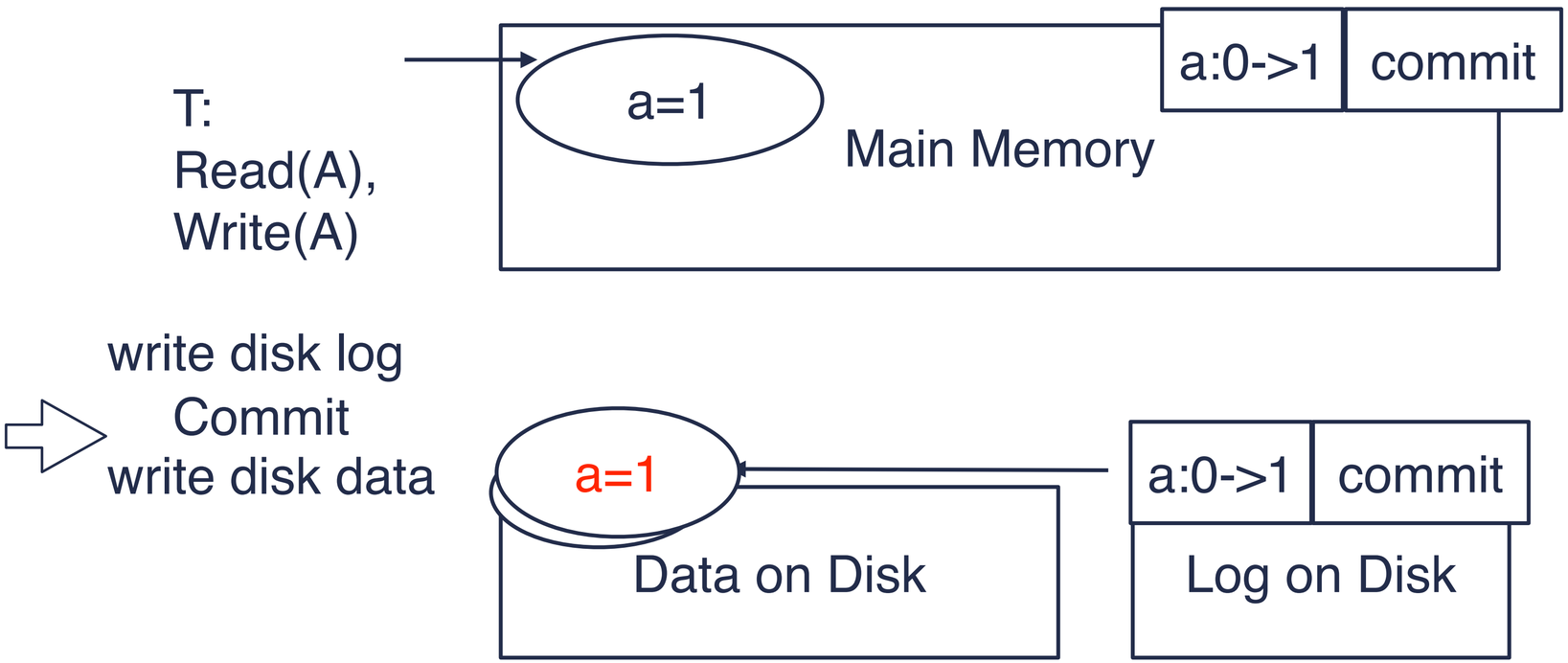


WAL: committing **after** we've written log to disk but **before** we've written data to disk

crash!

what happened?

WRITE-AHEAD LOGGING (WAL)



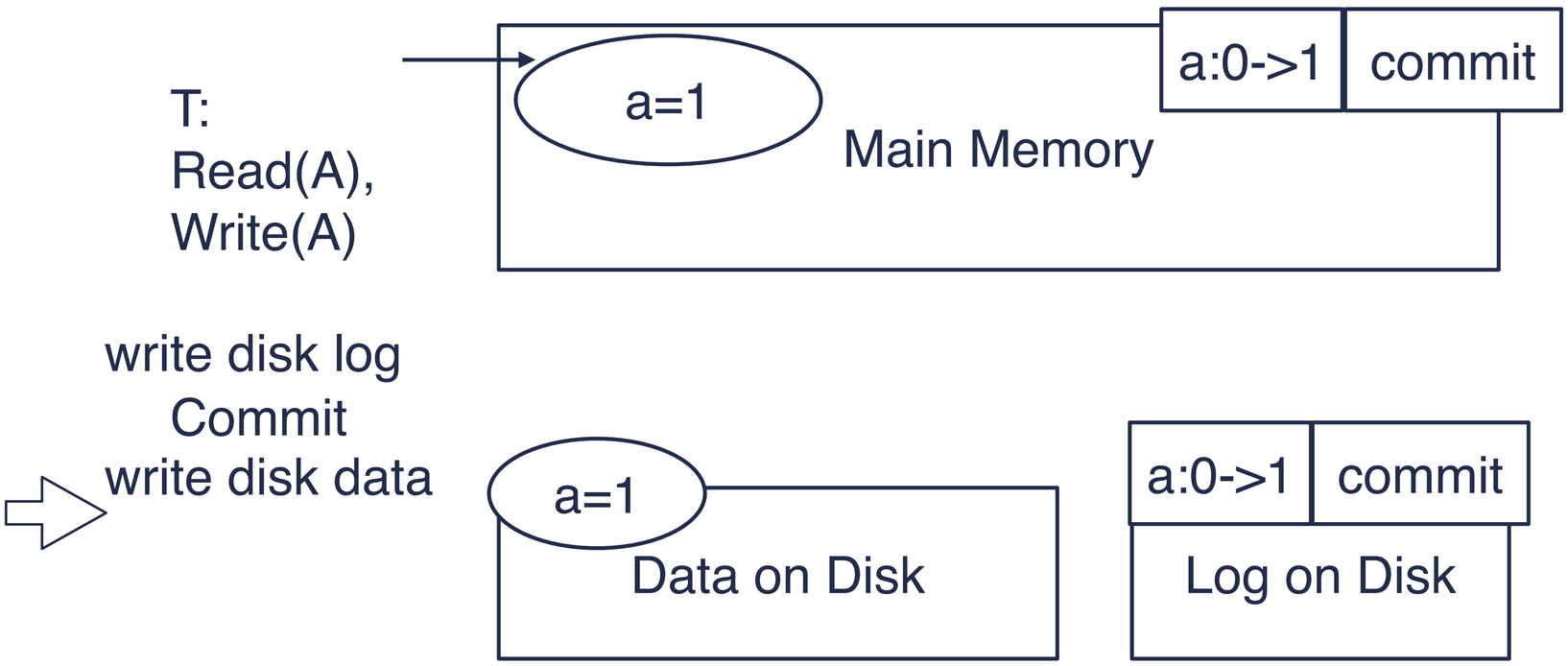
WAL: committing **after** we've written log to disk but **before** we've written data to disk

crash!

what happened?

Just REDO the log

WRITE-AHEAD LOGGING (WAL)

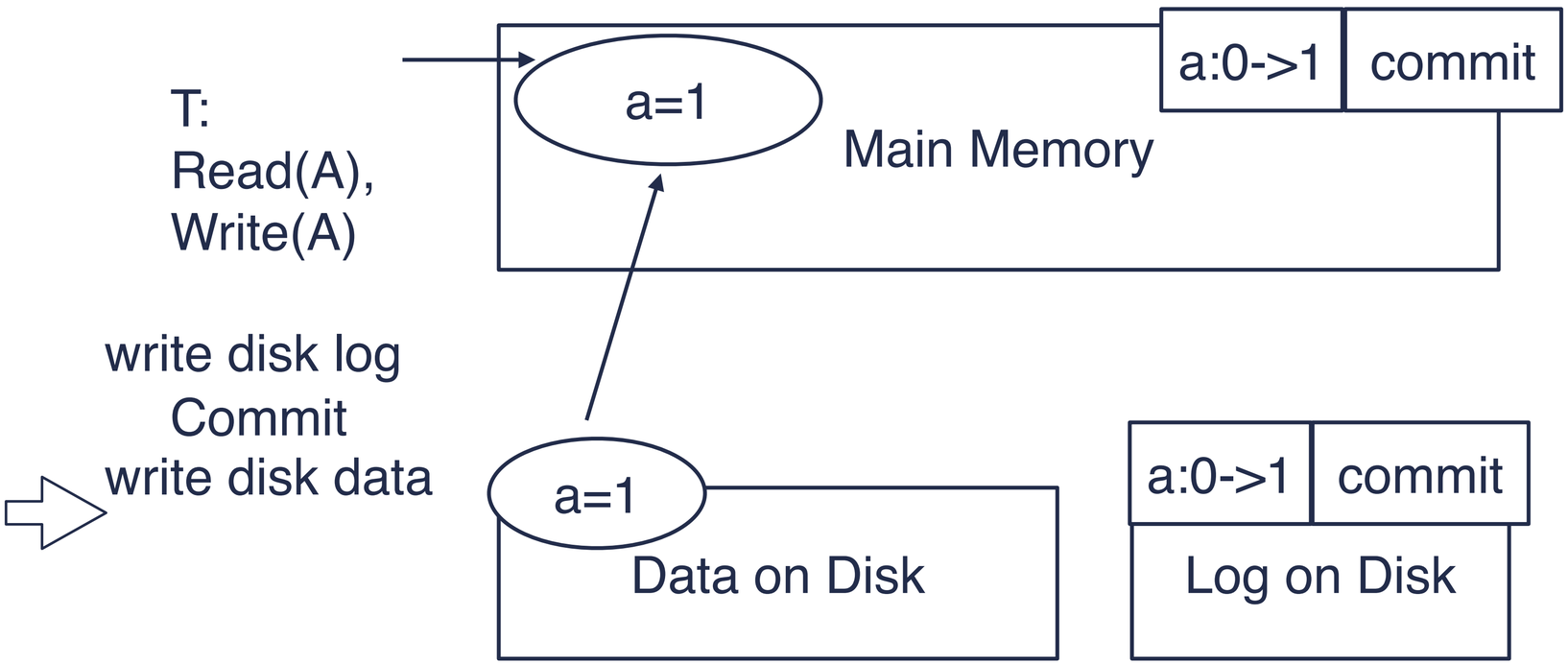


WAL: committing **after** we've written log to disk but **before** we've written data to disk

crash!

what happened?

WRITE-AHEAD LOGGING (WAL)



WAL: committing **after** we've written log to disk but **before** we've written data to disk

crash!

what happened?

read data from disk

RECOVERING FROM SIMPLE FAILURES

- e.g., system crash
  - For now, assume we can read the log
- “Analyze” the log
- Redo all (usually) transactions (forward)
  - Repeating history!
- Undo uncommitted transactions (backward)

WHY WRITE-AHEAD LOGGING (WAL)

- The Write-Ahead Logging Protocol:
  - Must enforce the log record for an update before the corresponding data page gets to disk (where the name of protocol comes from)
  - Must write all log records for a Xact before commit
- #1 guarantees Atomicity
- #2 guarantees Durability

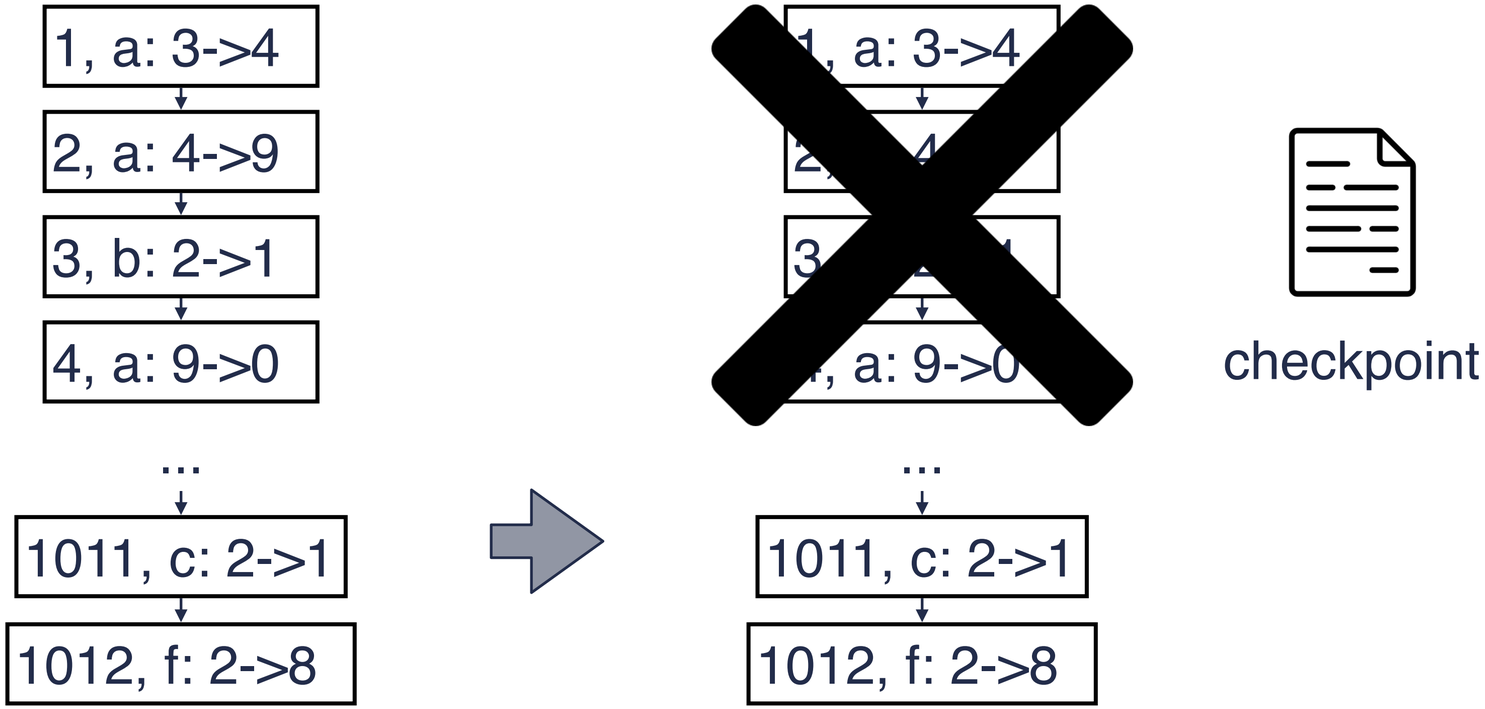
THE PERFORMANCE OF WAL

THE PERFORMANCE OF WAL

- Why performance is good?
  - This decouples writing a transaction's dirty pages to database on disk from committing the transaction.
  - We only need to write its corresponding log records.
  - If a txn updates a 100 tuples stored in 100 pages, we only need to write 100 log records (which could be a few pages) instead of 100 dirty pages.

CHECKPOINT

– Can start analysis with last checkpoint



TAKEAWAYS

- Systems need dealing with failures and concurrency
- Transactions provide Atomicity, Durability, Isolation
  - How? **Write-Ahead Logging**
- Next class: Transaction (contd.)
  - We'll discuss concurrency control techniques such as **locking**

ACKNOWLEDGEMENT

THIS COURSE IS DEVELOPED HEAVILY BASED ON COURSE MATERIALS SHARED BY PROF. INDRANIL GUPTA, PROF. ROBERT MORRIS, PROF. MICHAEL FREEDMAN, PROF. KYLE JAMIESON, PROF. WYATT LLOYD AND PROF. ROXANA GEAMBASU. MANY APPRECIATIONS FOR GENEROUSLY SHARING THEIR MATERIALS AND TEACHING INSIGHTS.

THIS SLIDES INCLUDES CONTENTS FROM TAMAL TANU BISWAS' SLIDES FOR ROCHESTER DATABASE SYSTEMS