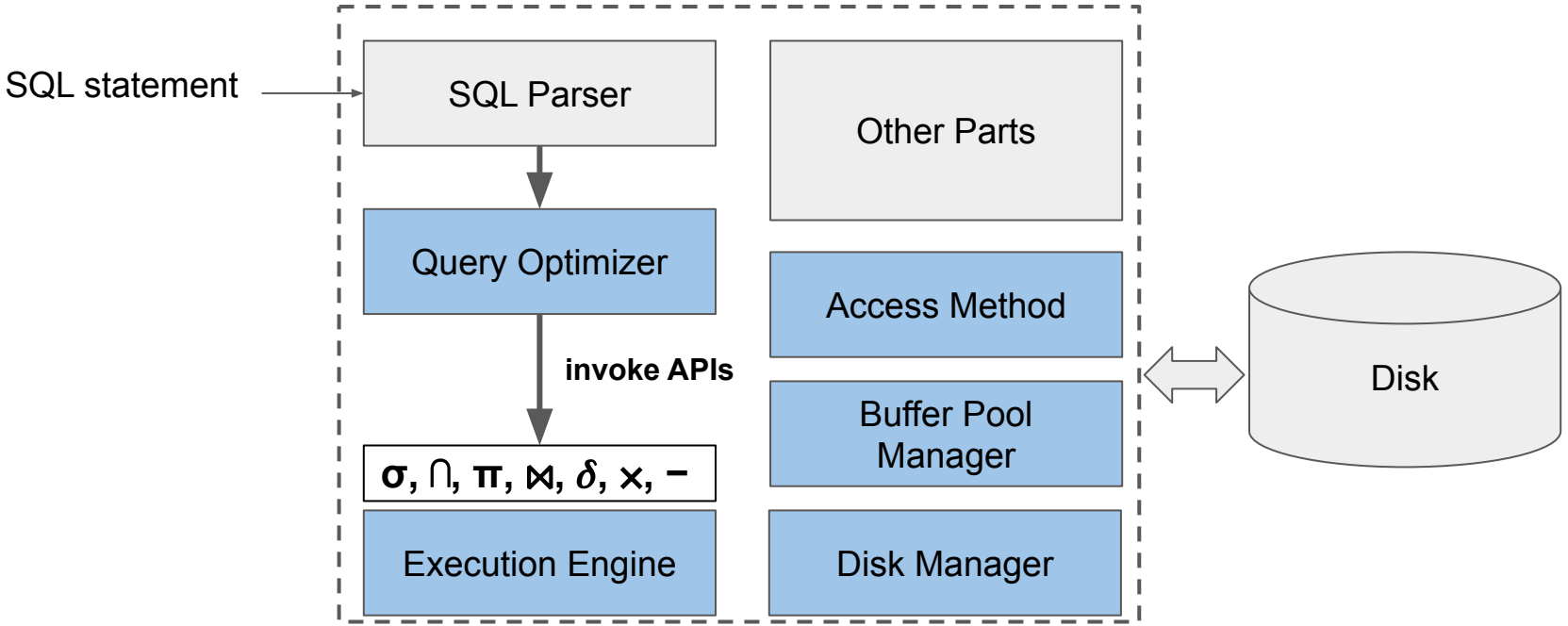
Databases and Big Data

Transactions 1

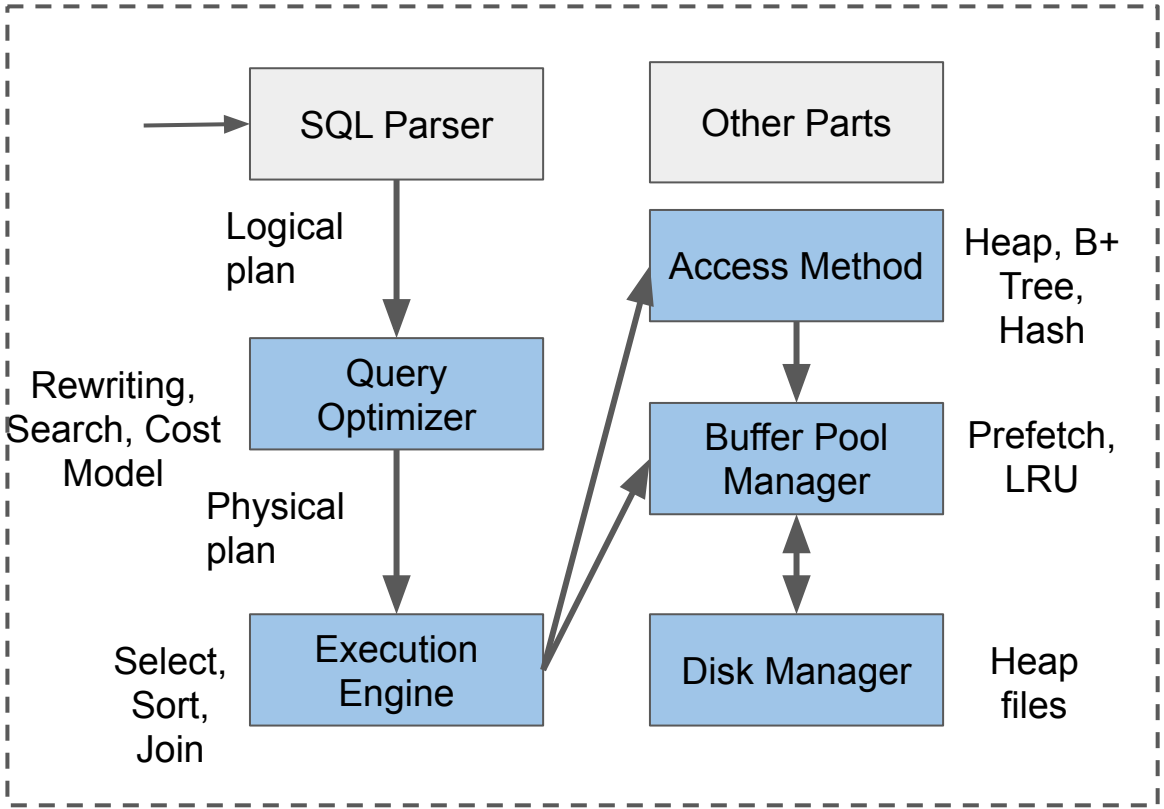
Recap

Need to know what the boxes do

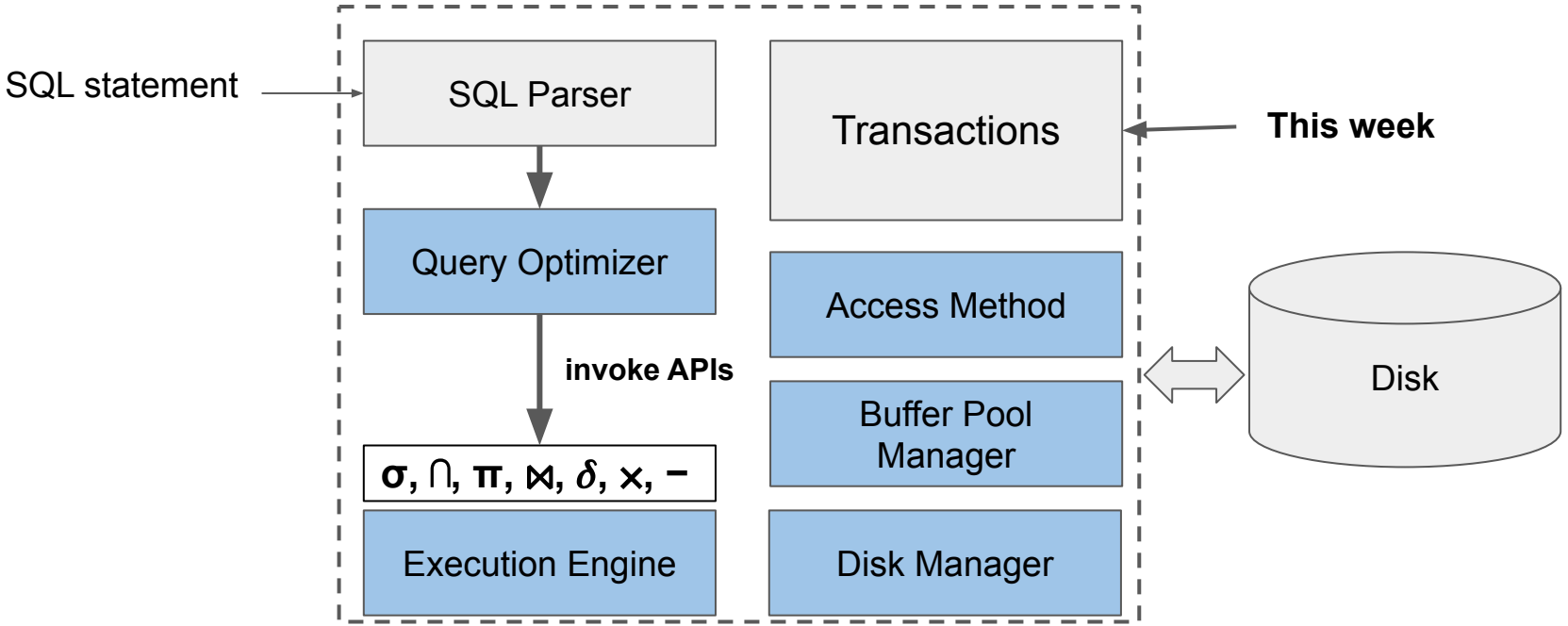


So Far

- How a read query
  - Executed
  - Made efficient



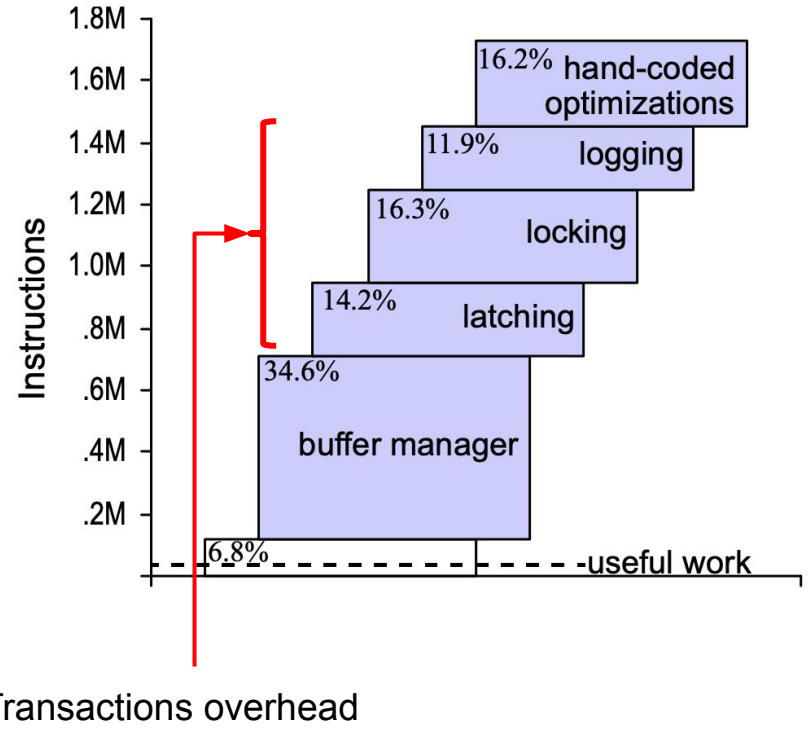
Schedule



How to support **write** operations under **failures** and **concurrency**

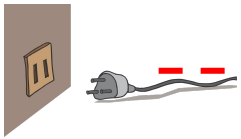
Transactions

- Hugely powerful abstraction
- Gives you **correctness**
- But significant overhead
- Don’t need it when
  - Read only
  - *Don’t care about correctness*



How To Lose Your Job Immediately

- Boss
  - Move some data to our Secret database
- You (didn't know about transactions)
  - Sure thing! Run this code.



```
insert into Secret
  select * from ProductionDB
  where recipient = "Anh";

delete from ProductionDB
where recipient = "Anh";
```

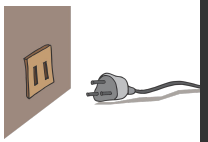
insert into secret
--> ok
delete
--> not ok

Not all operations are executed!

but the secret is still in productionDB

How To Lose Your Job Immediately

- Boss
  - Load this precious data in
- You (didn't know about transactions)
  - Sure thing! Run this code.



```
load data local infile "precious.csv"
into table ProductionDb
fields terminated by ','
lines terminated by '\n';
```

Not everything is loaded into a database

Not all operations are executed!

How To Rob A Bank

- Someone from the bank show you this database implementation


```
dispense(amount):
  old_balance = db.account.read("XYZ")
  new_balance = old_balance - amount
  db.account.write("XYZ", new_balance)
  dispense_cash(amount)
```

How To Rob A Bank

10

```
dispense(amount):
  old_balance = db.account.read("XYZ")
  new_balance = old_balance - amount
  db.account.write("XYZ", new_balance)
  dispense_cash(amount)
```

XYZ	100

10

```
dispense(amount):
  old_balance = db.account.read("XYZ")
  new_balance = old_balance - amount
  db.account.write("XYZ", new_balance)
  dispense_cash(amount)
```

XYZ	90

XYZ	80

How To Rob A Bank

When the local state is not reflective of true state.

XYZ	100

```
dispense(amount):
  old_balance = db.account.read("XYZ")
  new_balance = old_balance - amount
  db.account.write("XYZ", new_balance)
  dispense_cash(amount)
```

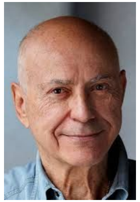
XYZ	90

XYZ	100

```
dispense(amount):
  old_balance = db.account.read("XYZ")
  new_balance = old_balance - amount
  db.account.write("XYZ", new_balance)
  dispense_cash(amount)
```

XYZ	90

XYZ	100



```
dispense(amount):
  old_balance = db.account.read("XYZ")
  new_balance = old_balance - amount
  db.account.write("XYZ", new_balance)
  dispense_cash(amount)
```

XYZ	90

How To Rob A Bank

- This did happen!

Concurrent write operations interfere with each other!

ars TECHNICA

BIZ & IT TECH SCIENCE POLICY CARS GAMING & CULTURE STORE FORUMS

FASTER THAN MINING —

How the "world's first Bitcoin bank" was robbed blind

Attacker overwhelmed the system, then withdrew $600,000 worth of bitcoins.

JON BRODKIN - 3/5/2014, 4:28 PM

Flexcoin, the self-proclaimed "world's first Bitcoin bank," was robbed by attackers who took advantage of a flaw in the bank's code for transferring bitcoins.

As reported yesterday, Flexcoin shut down after an attacker made off with 896 bitcoins, the equivalent of about $600,000. The company has since posted a more thorough explanation of just how it was robbed on its home page:

> The attacker logged into the flexcoin front end from IP address 207.12.89.117 under a newly created username and deposited to address 1DSD3B3uS2wGZjZAwa2dqQ7M9v7Ajw2iLy
>
> The coins were then left to sit until they had reached 6 confirmations.
>
> The attacker then successfully exploited a flaw in the code which allows transfers between flexcoin users. By sending thousands of simultaneous requests, the attacker was able to "move" coins from one user account to another until the sending account was overdrawn, before balances were updated.
>
> This was then repeated through multiple accounts, snowballing the amount, until the attacker withdrew the coins. (Here and Here.)

Transaction

- Two problems with write operations

Problem	**Caused by**
Not all operations are executed!	Failure
Concurrent write operations interfere with each other!	Concurrency

- Transaction:
  - A sequence of operations, executed together as one unit
  - Without the above two problems

Transaction

- Transaction Abstraction

BEGIN

[Statements]

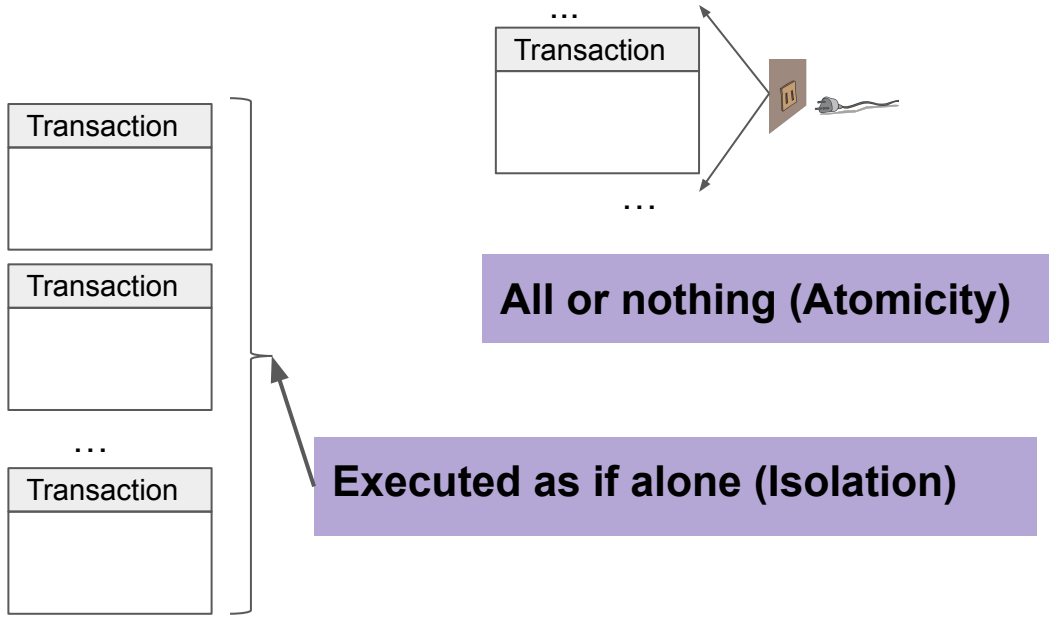
COMMIT

```
BEGIN
insert into Secret
  select * from ProductionDB
  where recipient = "Anh";
delete from ProductionDB
  where recipient = "Anh";
COMMIT
```

```
BEGIN
    old_balance = db.account.read("XYZ")
    new_balance = old_balance - amount
    db.account.write("XYZ", new_balance)
    dispense_cash(amount)
COMMIT
```

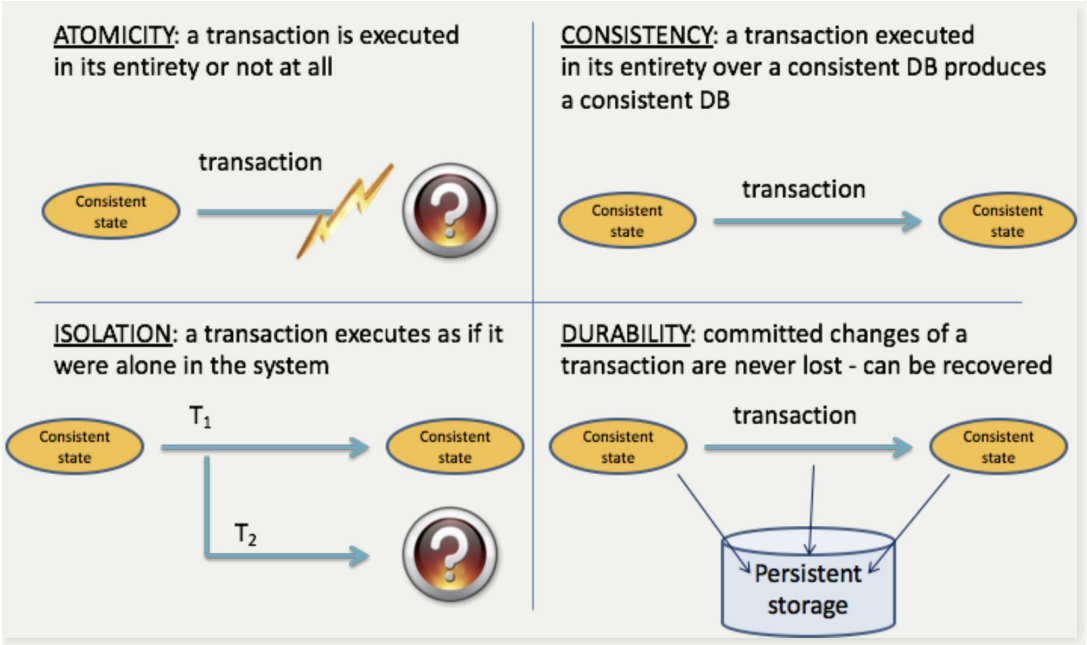
Transaction

Transaction = A sequence of operations, executed together as **one indivisible unit**



Transaction

- ACID
  - You may see this around
  - In this course, we care about A and I



Transaction

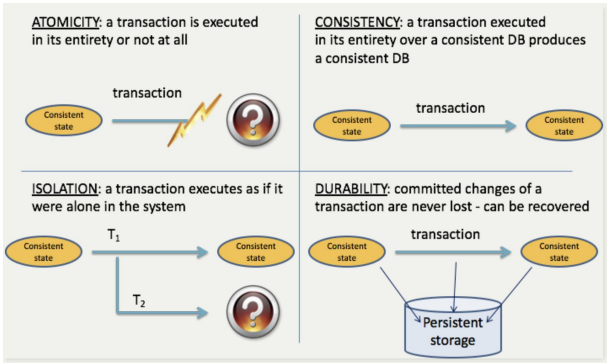
- Strawman solution (shadow file):
  - Before BEGIN, make a copy of database
  - Changes applied to the copy
  - When COMMIT, *rename* the copy to the master
  - [Seen something similar in Linux?]
- ACID?
  - Yes, assuming that *rename* is atomic
- But it's not the way DBMS does it

BEGIN

[Statements]

COMMIT

- High performance cost of making a copy of database
- Assuming rename is atomic --> so rename cannot fail



Atomicity = All or Nothing

- Question: how to make a transaction all-or-nothing?
- Challenges:
  - Multiple operations/statements
  - There are failures

```
old_balance = db.account.read("XYZ")
new_balance = old_balance - amount
db.account.write("XYZ", new_balance)
dispense_cash(amount)
```

NO

YES

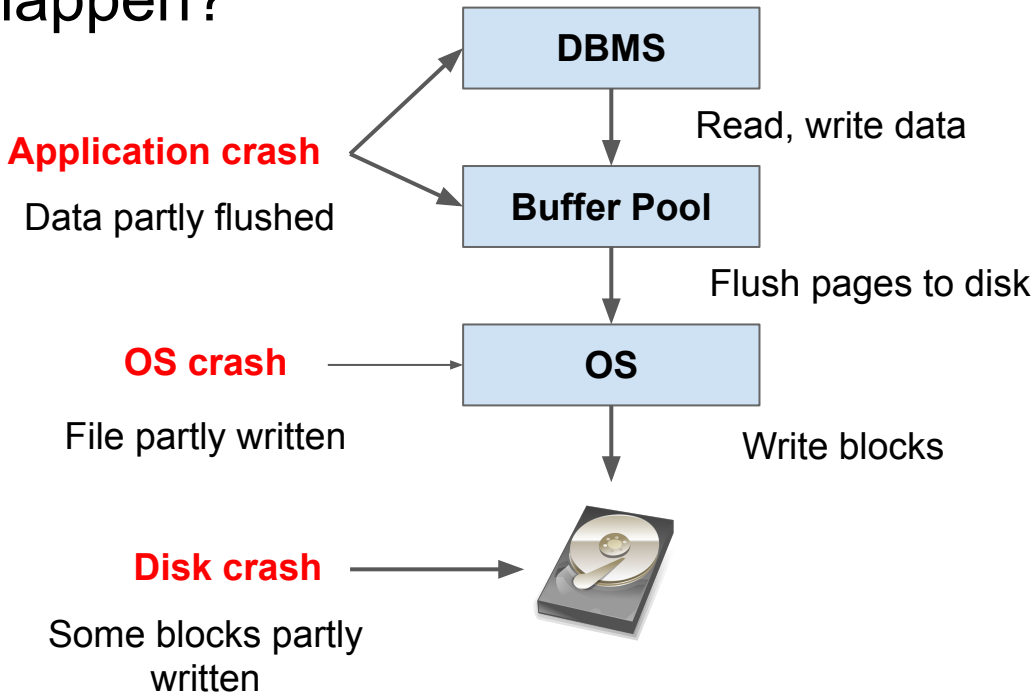
```
entry:
  # Turn on page size extension for 4Mbyte
  movl    %cr4, %eax
  orl     $(CR4_PSE), %eax
  movl    %eax, %cr4
  # Set page directory
  movl    $(V2P_WO(entrypgdir)), %eax
  movl    %eax, %cr3
  # Turn on paging.
  movl    %cr0, %eax
  orl     $(CR0_PG|CR0_WP), %eax
  movl    %eax, %cr0

  # Set up the stack pointer.
  movl $(stack + KSTACKSIZE), %esp
```

Each of these statements is atomic

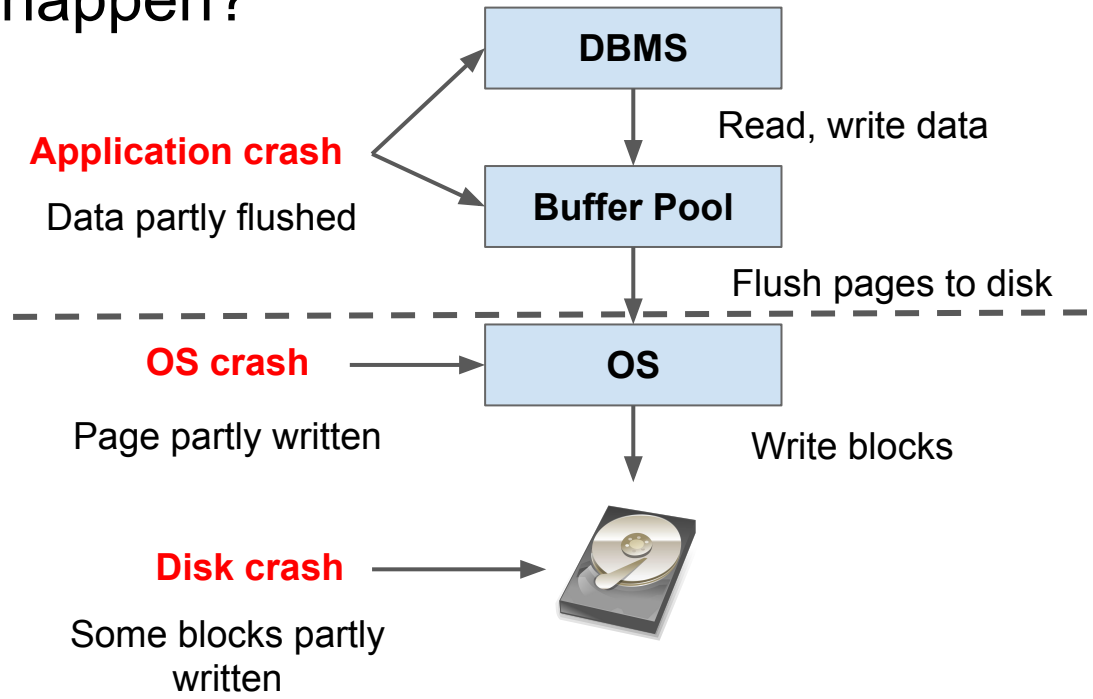
Failures

- Where crashes happen?



Failures

- Where crashes happen?

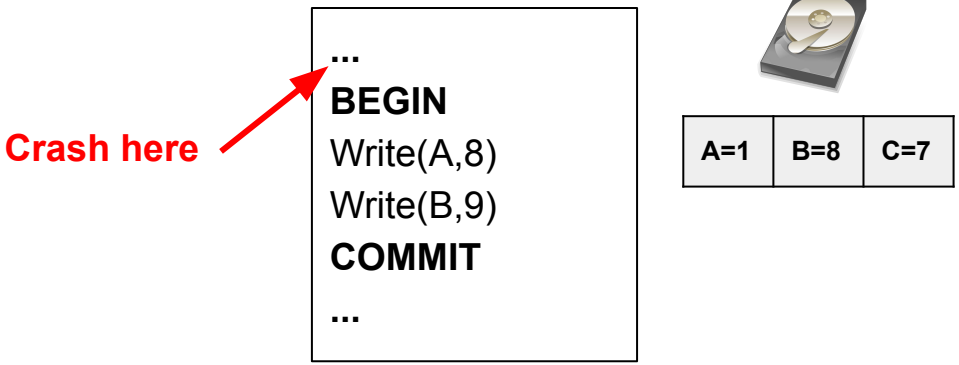


For this course, we assume page content on disk is not corrupted by OS crash

flush(page) is atomic

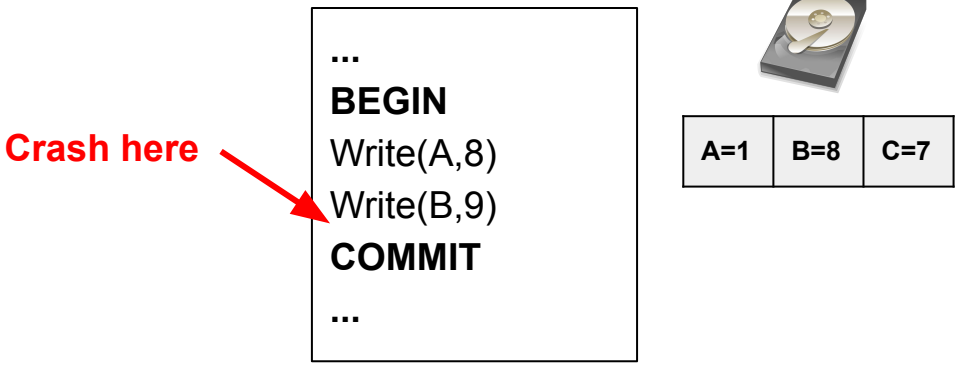
Atomicity

- Example



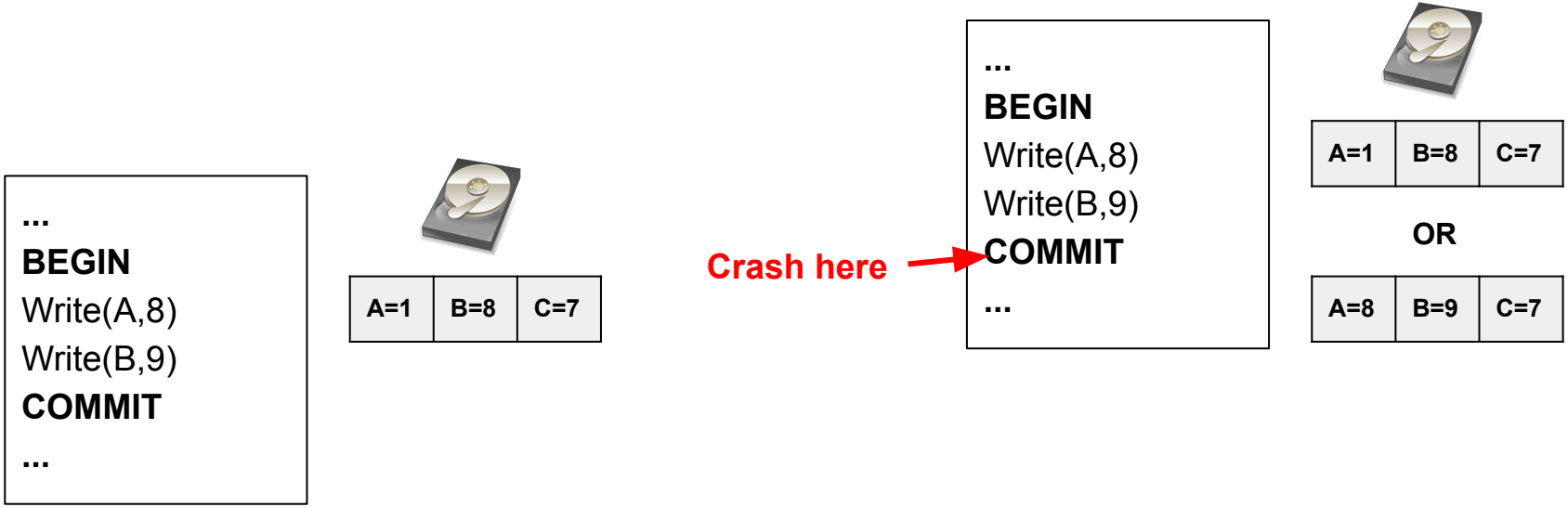
Atomicity

- Example



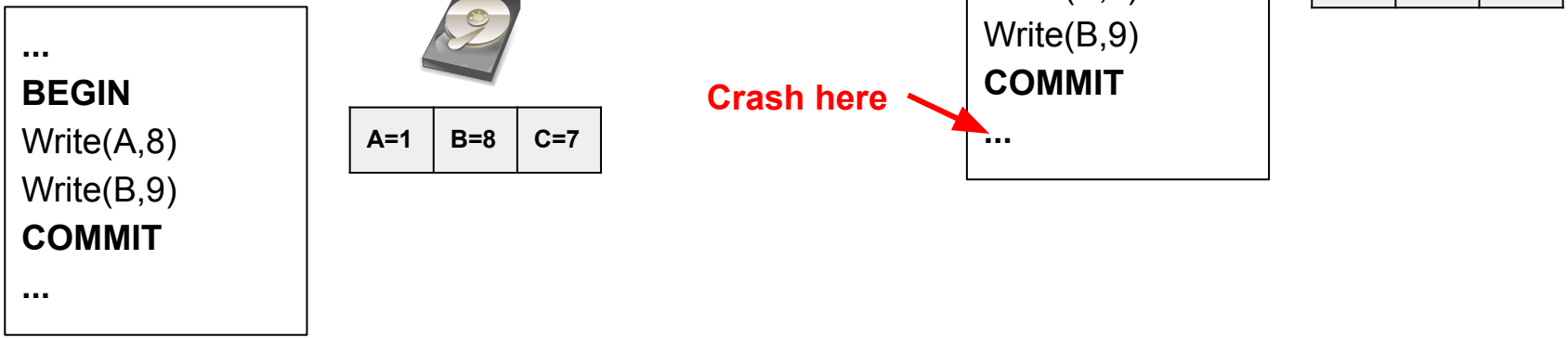
Atomicity

- Example



Atomicity

- Example



Atomicity

- Before COMMIT
  - Nothing is written to disk
- COMMIT = point of no return
  - If passed (return True), everything written to disk
  - If failed (return False), nothing written to disk
  - If crashed, either all or nothing.

```
...
BEGIN
Write(A,8)
Write(B,9)
COMMIT
...
```

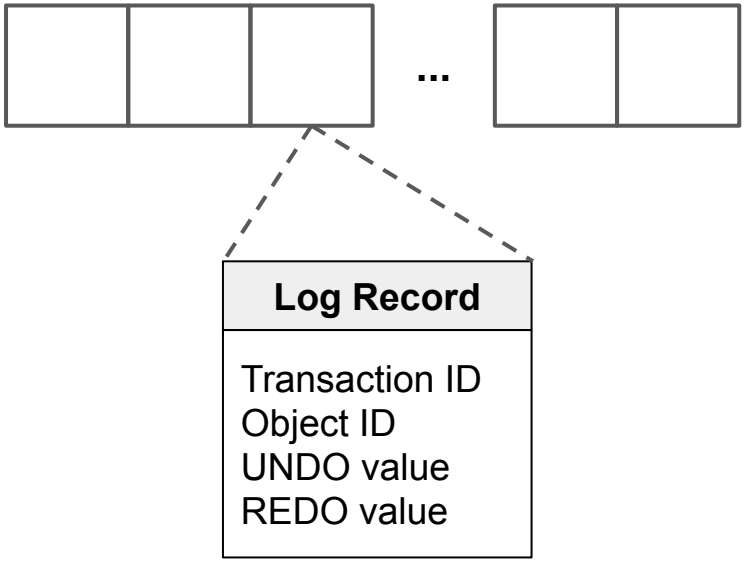
Atomicity

- Write Ahead Logging
  - During a transaction, record changes to a log file
  - The log file is on disk
  - Log file contains information to recover after crash
- Main idea:
  - DBMS stages the log file in memory
  - On COMMIT, dump the log file to disk
  - If successful, COMMIT is successful
  - **Only then** the data is updated on disk

WAL

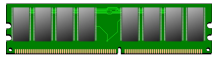
objectid -> A or B in prev example

- What's in the log



WAL

...
BEGIN
Write(A,8)
Write(B,9)
COMMIT
...



WAL

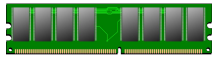
Buffer Pool

A=8	B=9	C=7

... | X | Y |

WAL

...
BEGIN
Write(A,8)
Write(B,9)
COMMIT
...



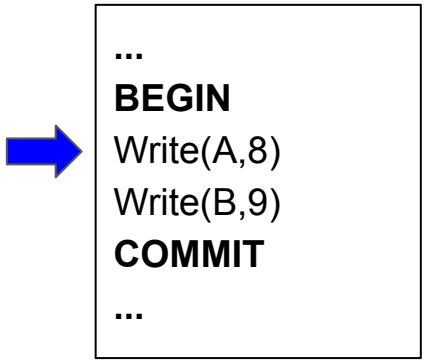
WAL
<T_1, BEGIN>

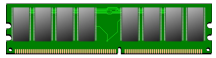
Buffer Pool

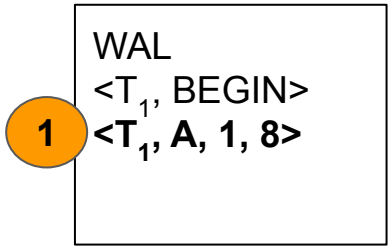
A=8	B=9	C=7

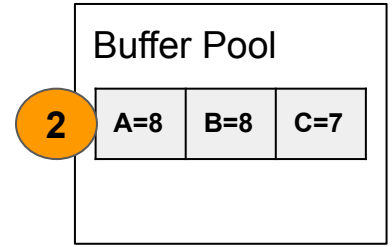
... | X | Y |

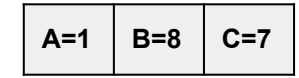
WAL


...
BEGIN
Write(A,8)
Write(B,9)
COMMIT
...



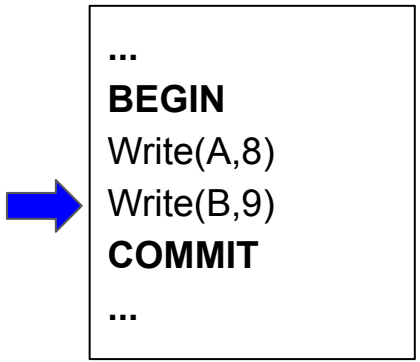

WAL
<T1, BEGIN>
1
<T1, A, 1, 8>

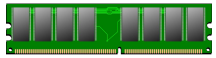

Buffer Pool
2
A=8
B=8
C=7


A=1
B=8
C=7


...
X
Y

WAL





WAL
$<T_1$, BEGIN>
$<T_1$, A, 1, 8>
$<T_1$, B, 8, 9>

1

Buffer Pool

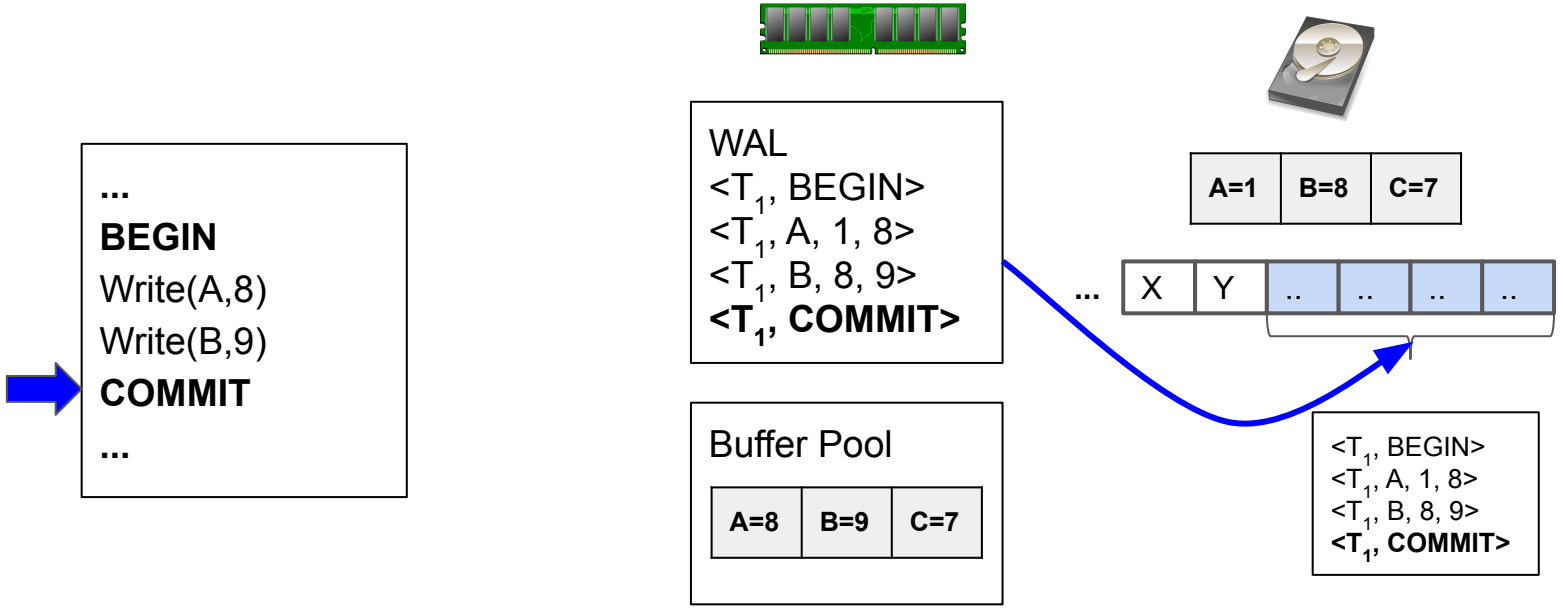
2

A=8	B=9	C=7

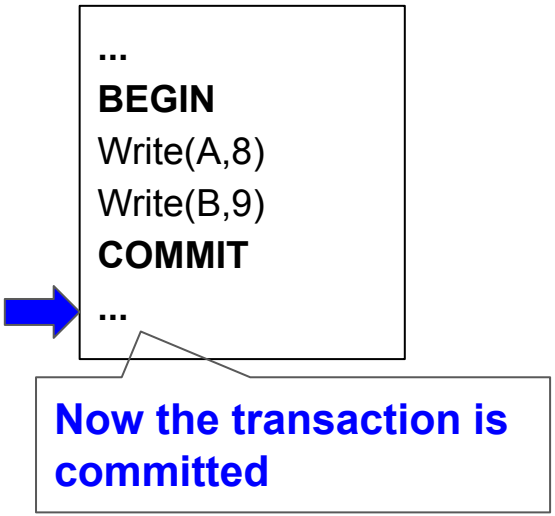
A=1	B=8	C=7

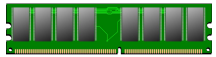
... | X | Y |

WAL


...
BEGIN
Write(A,8)
Write(B,9)
COMMIT
...
WAL
<T_1, BEGIN>
<T_1, A, 1, 8>
<T_1, B, 8, 9>
<T_1, COMMIT>
Buffer Pool
A=8
B=8
C=7
A=1
B=8
C=7
...
X
Y
..
..
..
..
<T_1, BEGIN>
<T_1, A, 1, 8>
<T_1, B, 8, 9>
<T_1, COMMIT>

WAL





WAL

Buffer Pool

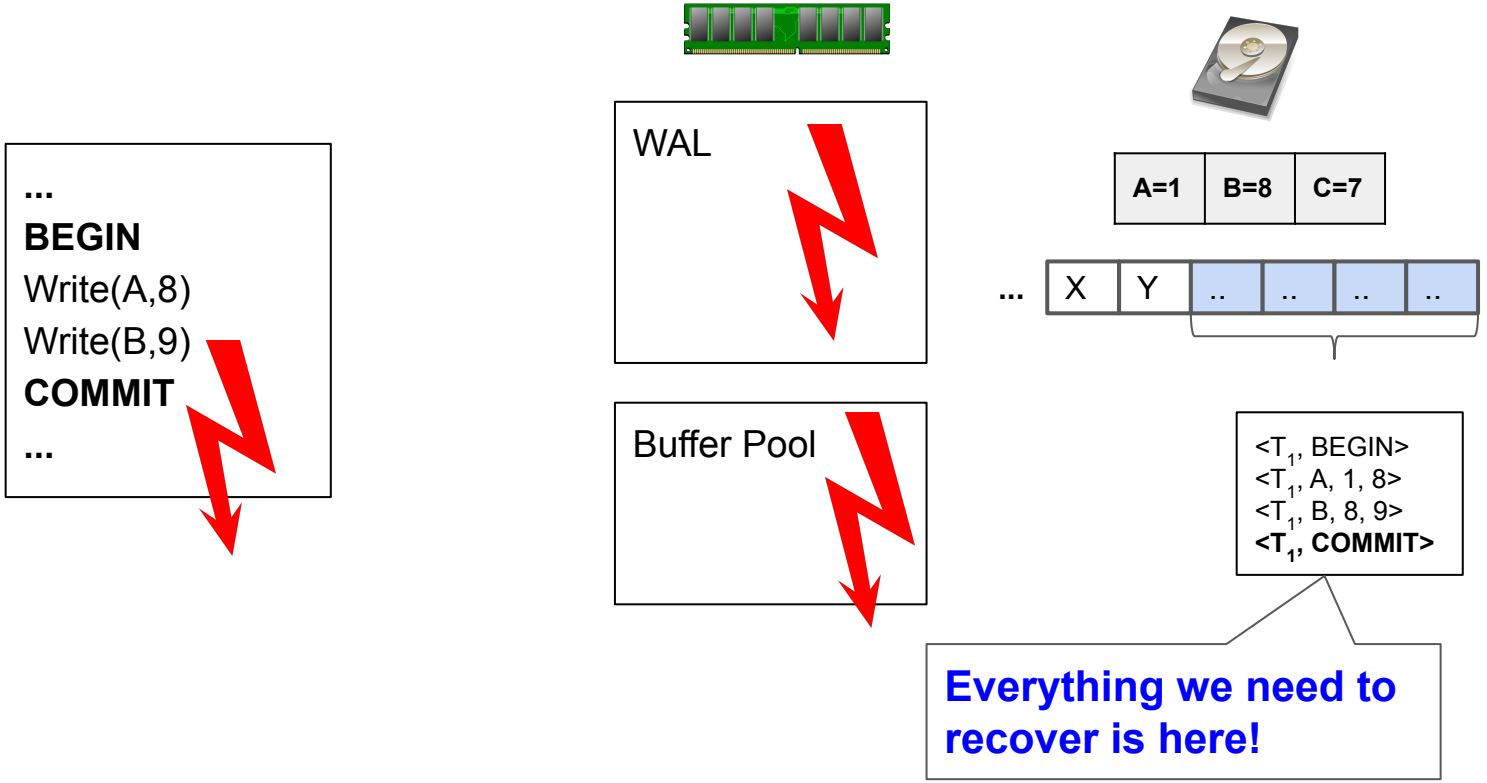
A=8	B=8	C=7

A=1	B=8	C=7

... | X | Y | .. | .. | .. | .. |

$<T_1$, BEGIN>
$<T_1$, A, 1, 8>
$<T_1$, B, 8, 9>
$<T_1$, COMMIT>

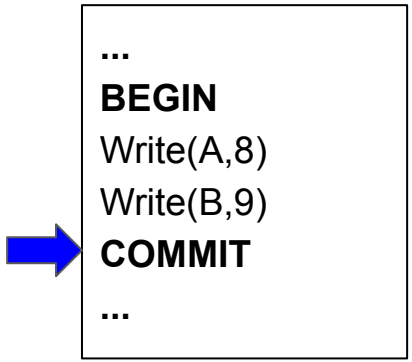
WAL

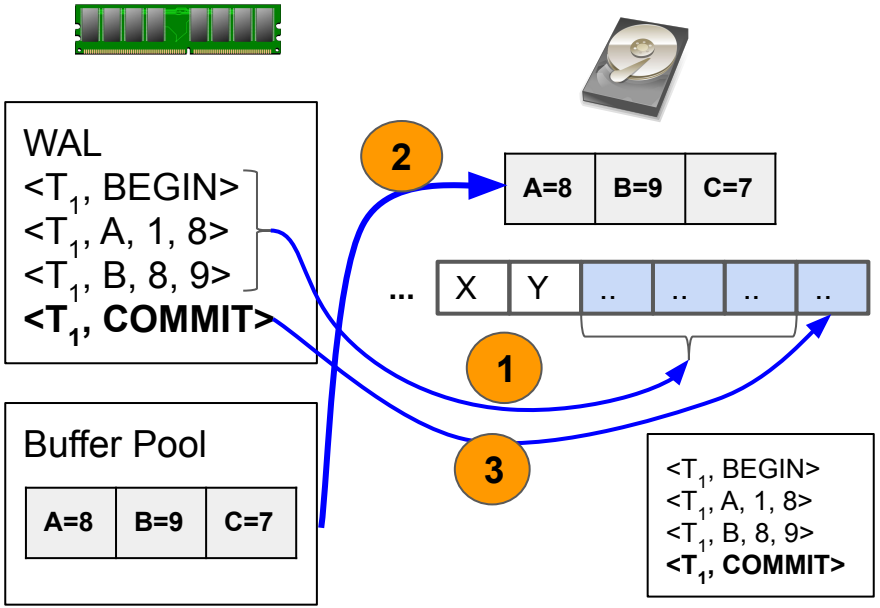

...
BEGIN
Write(A,8)
Write(B,9)
COMMIT
...
WAL
Buffer Pool
A=1
B=8
C=7
...
X
Y
..
..
..
..
<T1, BEGIN>
<T1, A, 1, 8>
<T1, B, 8, 9>
<T1, COMMIT>
Everything we need to recover is here!

WAL

- When is data updated on disk?
  - Before COMMIT returns
  - Called **UNDO logging**

While modifying the value in disk, we create corresponding records in the logs.
*So if there isn't a **<TX, COMMIT>** record at the end, we have to undo what we did.*

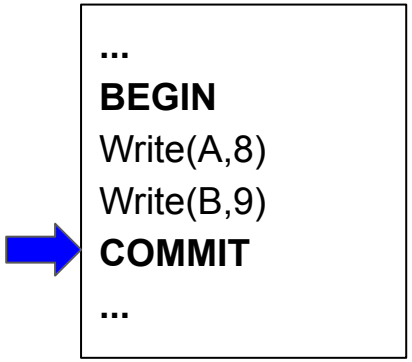


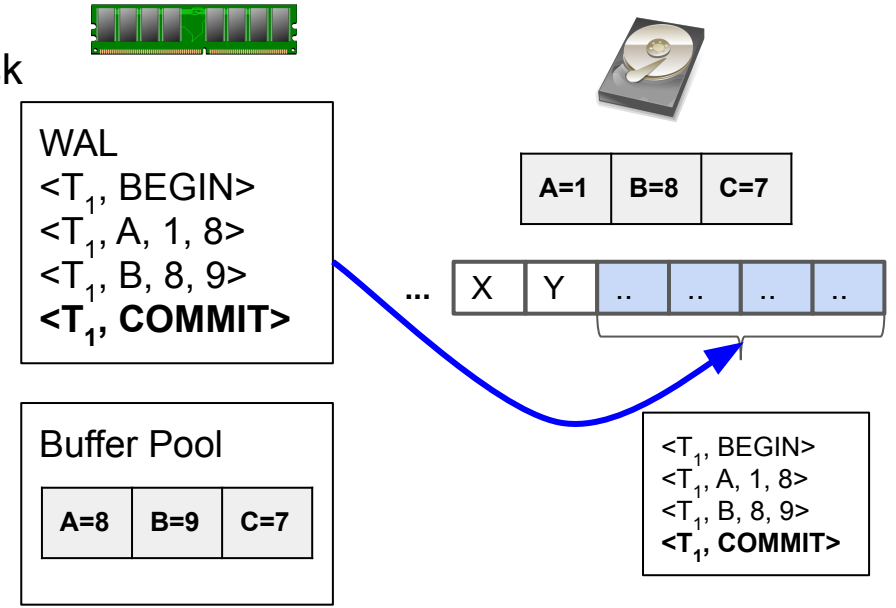


Begin -> Commit is 1 transaction
max undo 1 transaction

WAL

- When is data updated on disk?
  - After COMMIT returns
    - Write a FLUSHED record to disk
  - Called **REDO logging**





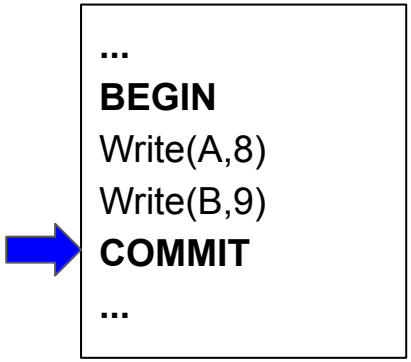
WAL

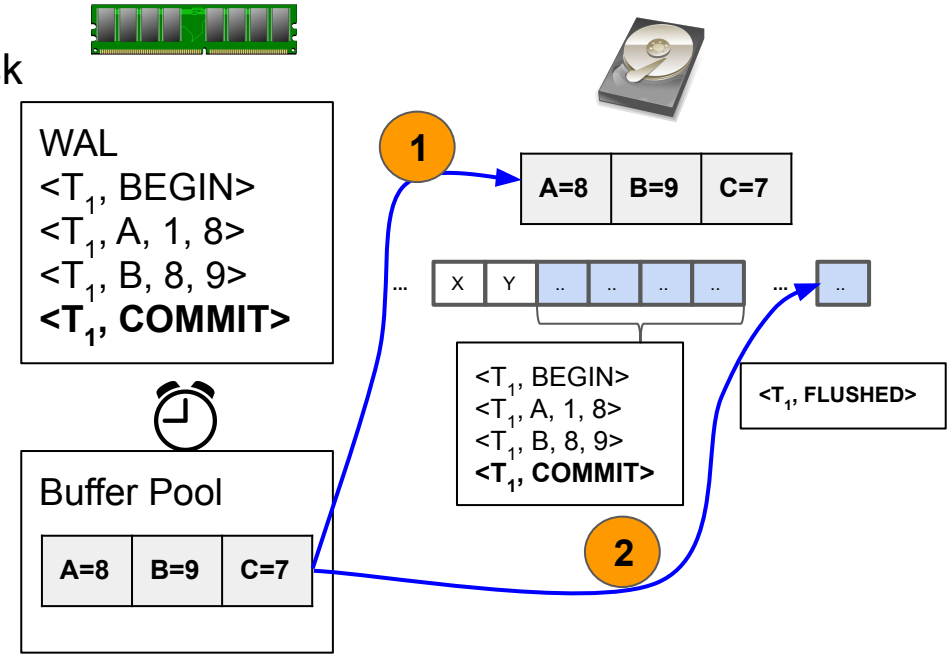
*You update the data to disk altogether after **COMMIT** returns. Then when it finishes, you write a **<TX, FLUSHED>** record.*

***COMMIT** -> user knows transaction was executed correctly. Even if there is some error now, we have to ensure that the DB state is the state user expects after the transaction happens*

***FLUSHED** -> Results are on disk - no need to redo anything*

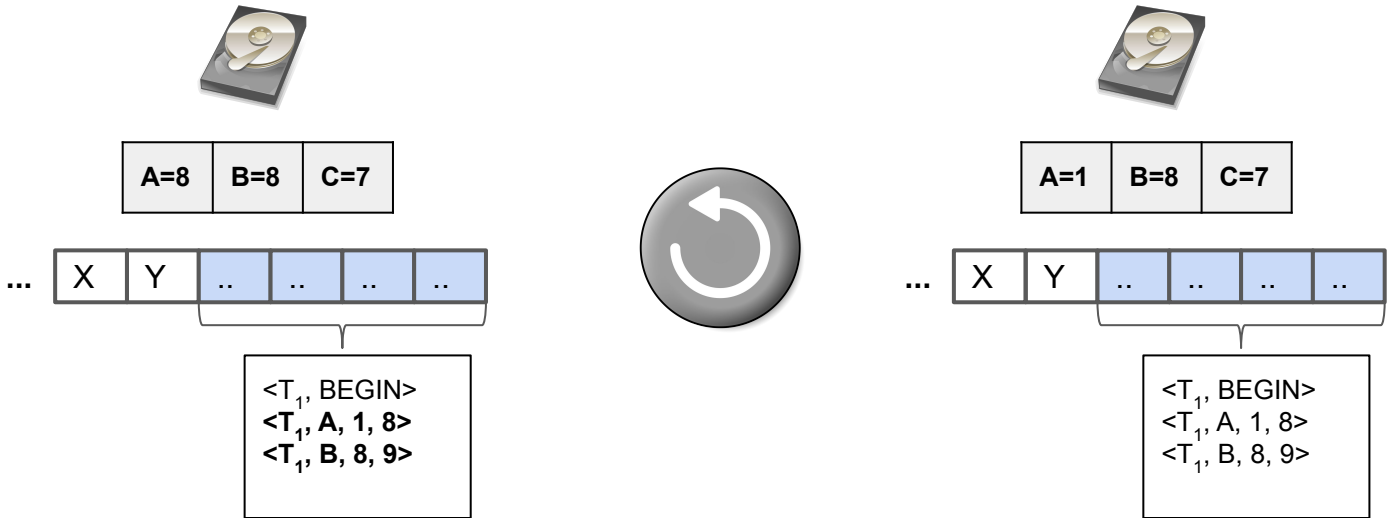
- When is data updated on disk?
  - After COMMIT returns
    - Write a FLUSHED record to disk
  - Called **REDO logging**





WAL

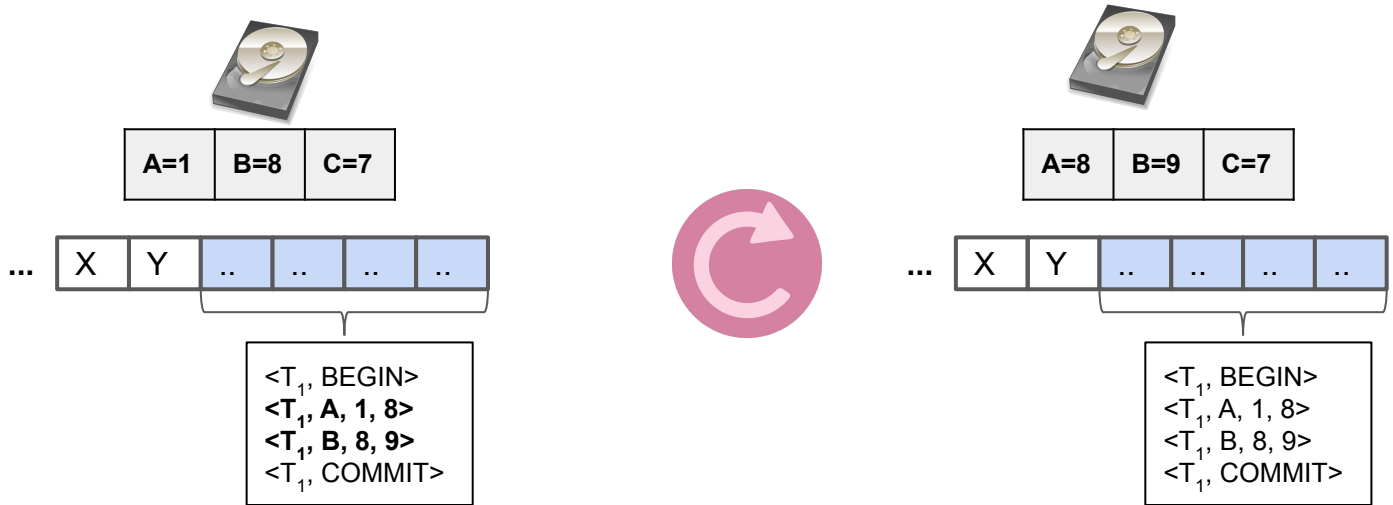
- Recovery with UNDO logging
  - Replay UNDO values for every transaction
    - Without <.., COMMIT> record



WAL

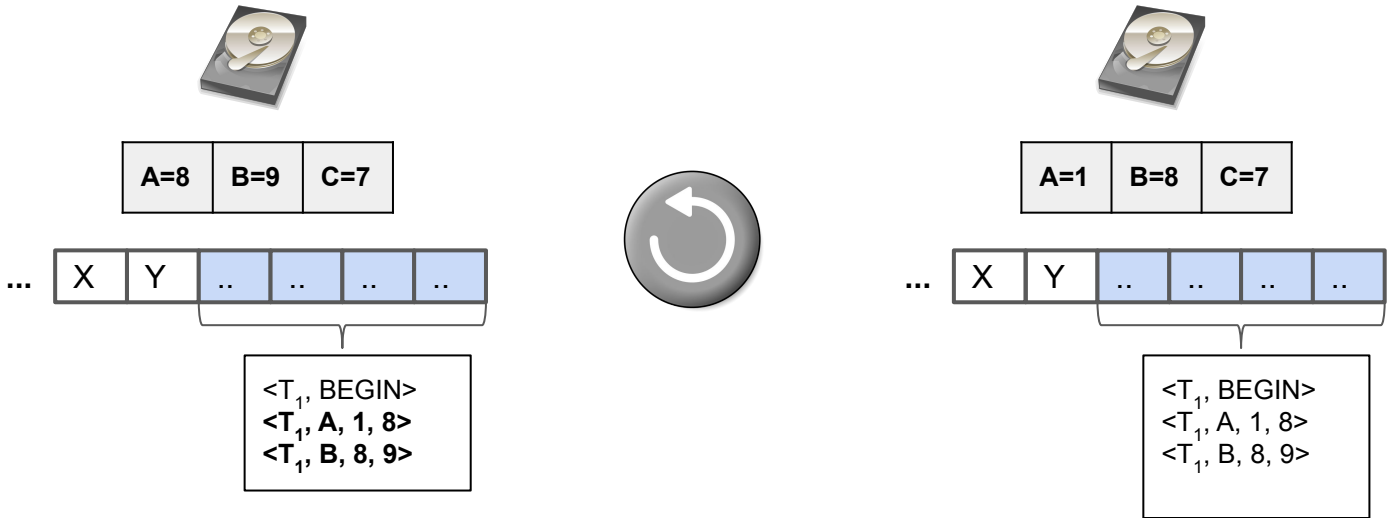
- Need to redo all (many) transaction
- During commit you don't pay the price to flush but during recovery you pay the price of many more transaction.
- Ignore transactions that do not have commit
- Scan from the end then start from where you stopped when you redo
Redo: *- If we dont see T1 Flush we have to redo*

- Recovery with REDO logging
  - Replay REDO values for every COMMITTED transaction
    - Ones with <.., COMMIT> records
    - But without <..., FLUSHED> record



WAL

- Recovery with UNDO logging
  - Replay UNDO values for every non-COMMITTED transaction
    - Ones without <.., COMMIT> records



Undo vs. Redo Logging

- Undo:
  - Slow COMMIT
- Redo:
  - Require large buffer
  - Slow recovery (may need to redo a lot)

Undo:
Need to wait for 2 steps
Redo:
- Avoid flushing so have to keep many things in buffer

Summary

- Transaction is powerful abstraction
- Atomicity & Isolation are important
- Write Ahead Logging for atomicity
  - Write changes to log files first
  - Write log files before update data on disk
  - REDO logging: update data **after** COMMIT record is flushed to disk
  - UNDO logging: update data **before** COMMIT record is flushed to disk