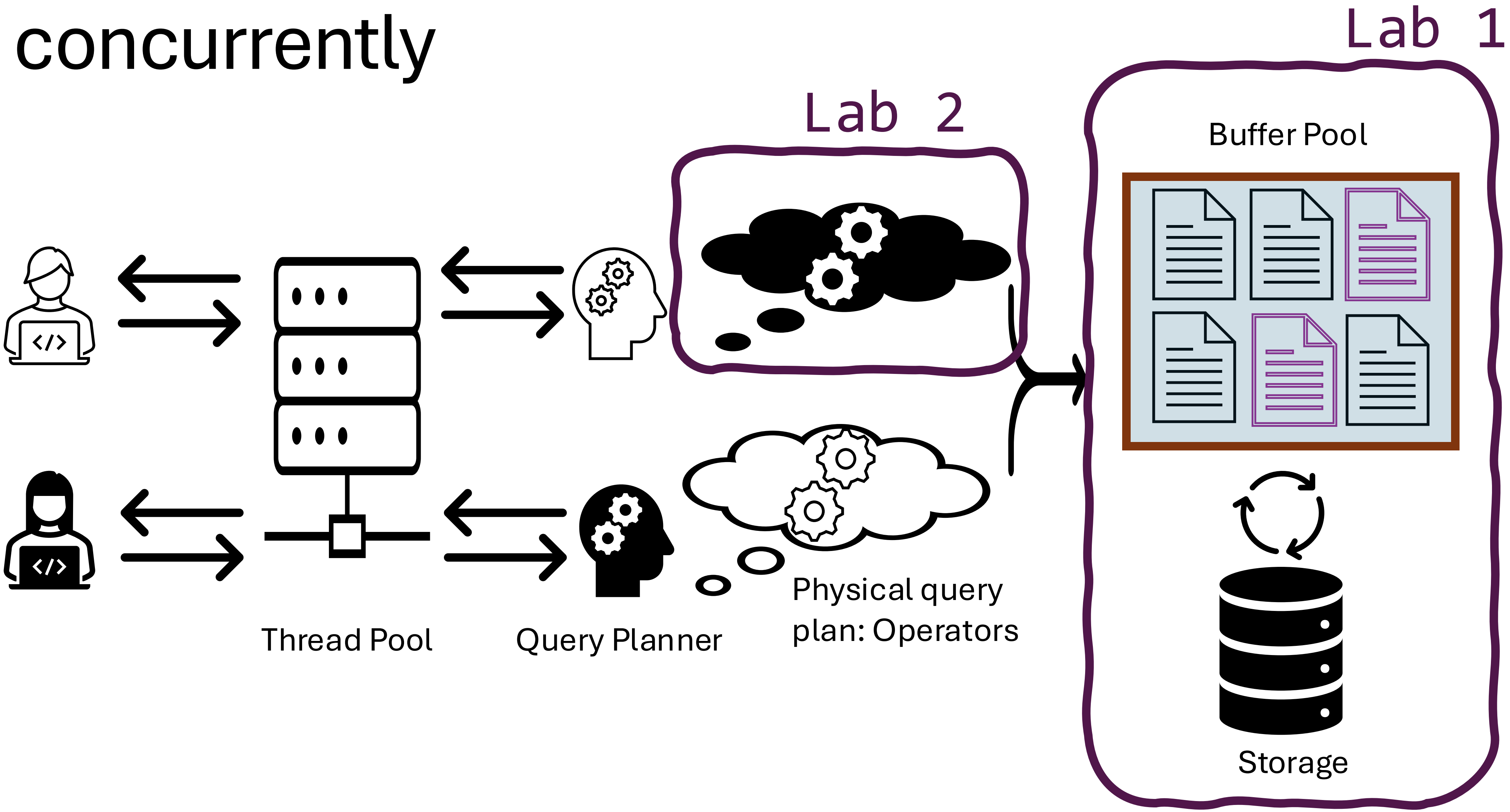
Lab 3 Bootcamp

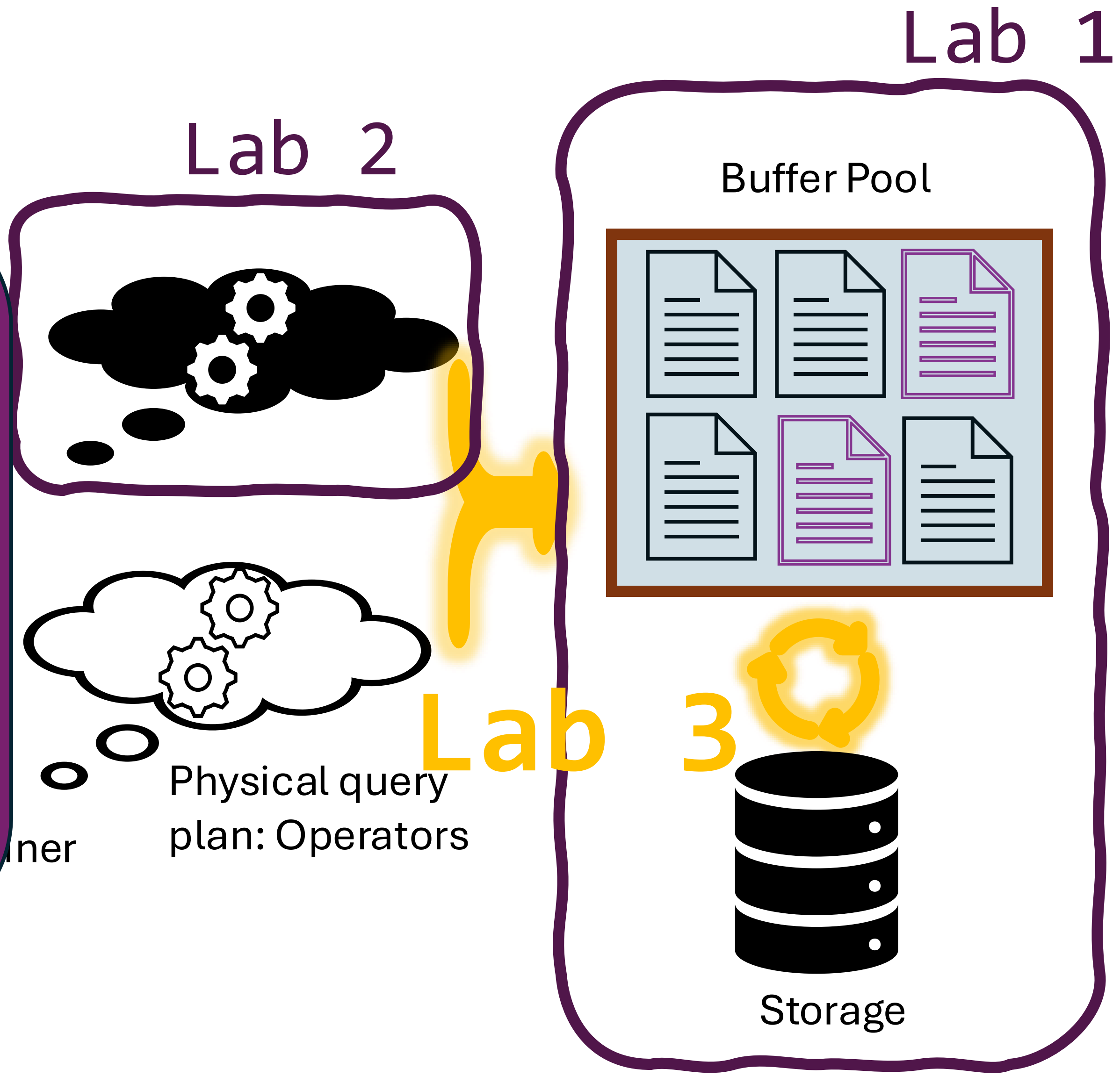
10/31/2024

Different threads are trying to execute queries concurrently



Different threads are trying to execute queries concurrently

Handled by starter code and test cases



Lab 1

Lab 2

Buffer Pool

ner

Physical query plan: Operators

Lab 3

Storage

ACID Properties of Transactions

- **A** tomicity – many actions look like one; “all or nothing”
- **C** onsistency – database preserves invariants
- **I** solation – concurrent actions don’t see each other’s results
- **D** urability – completed actions in effect after crash (“recoverable”)

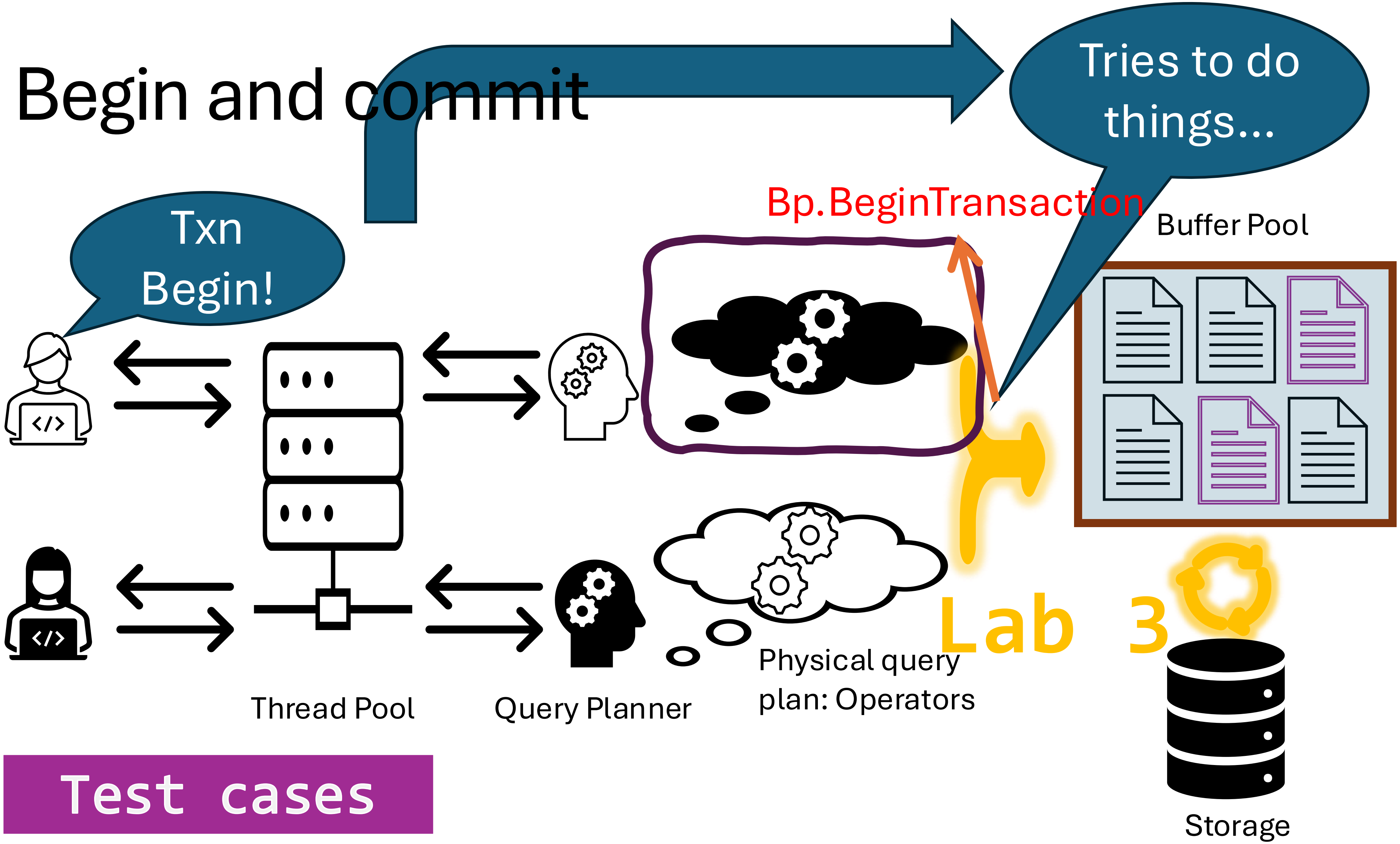
- **A**tomicity – many actions look like one; “all or nothing”
- In reality, actions take time!
  - To get atomicity, to prevent multiple actions from interfering with each other
  - I.e., are **I**solated – transactions do not interfere with each other

- **D**urability – recoverable into a state where no partial transactions are present after a crash
  - Usually implemented with Write-ahead Logs. ***We will not do this for lab 3 but you can try this for lab 4.***

Users view: Three transaction operations

- BEGIN TRANSACTION
  - Followed by SQL operations that (may or may not) modify database
- COMMIT: make the effects of the transaction durable
  - After COMMIT returns database guarantees results present even after crash
  - And results are visible to other transactions
- ABORT: undo all effects of the transaction

Begin and commit



Txn Begin!

Tries to do things...

Bp.BeginTransaction

Buffer Pool

Thread Pool

Query Planner

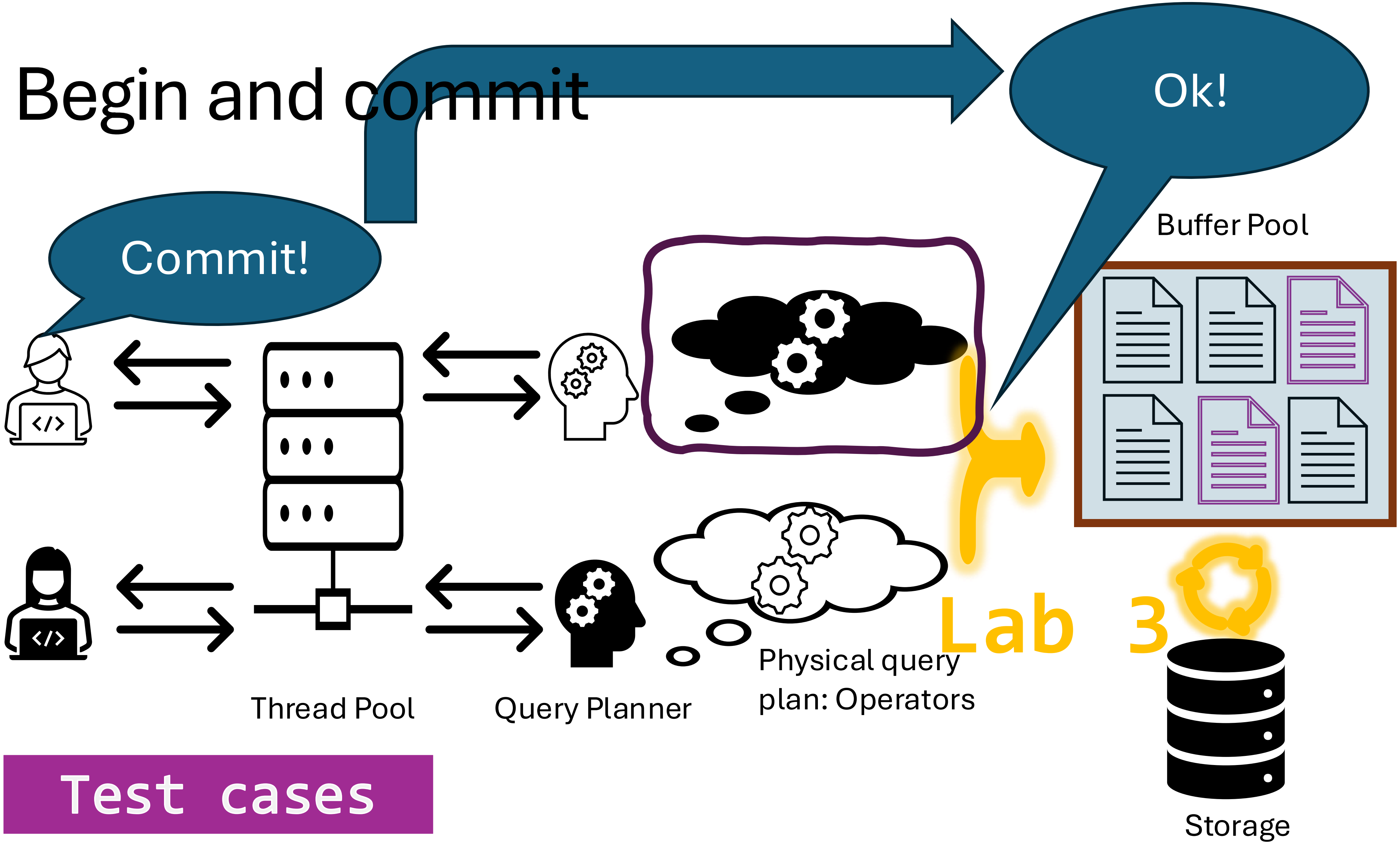
Physical query plan: Operators

Lab 3

Storage

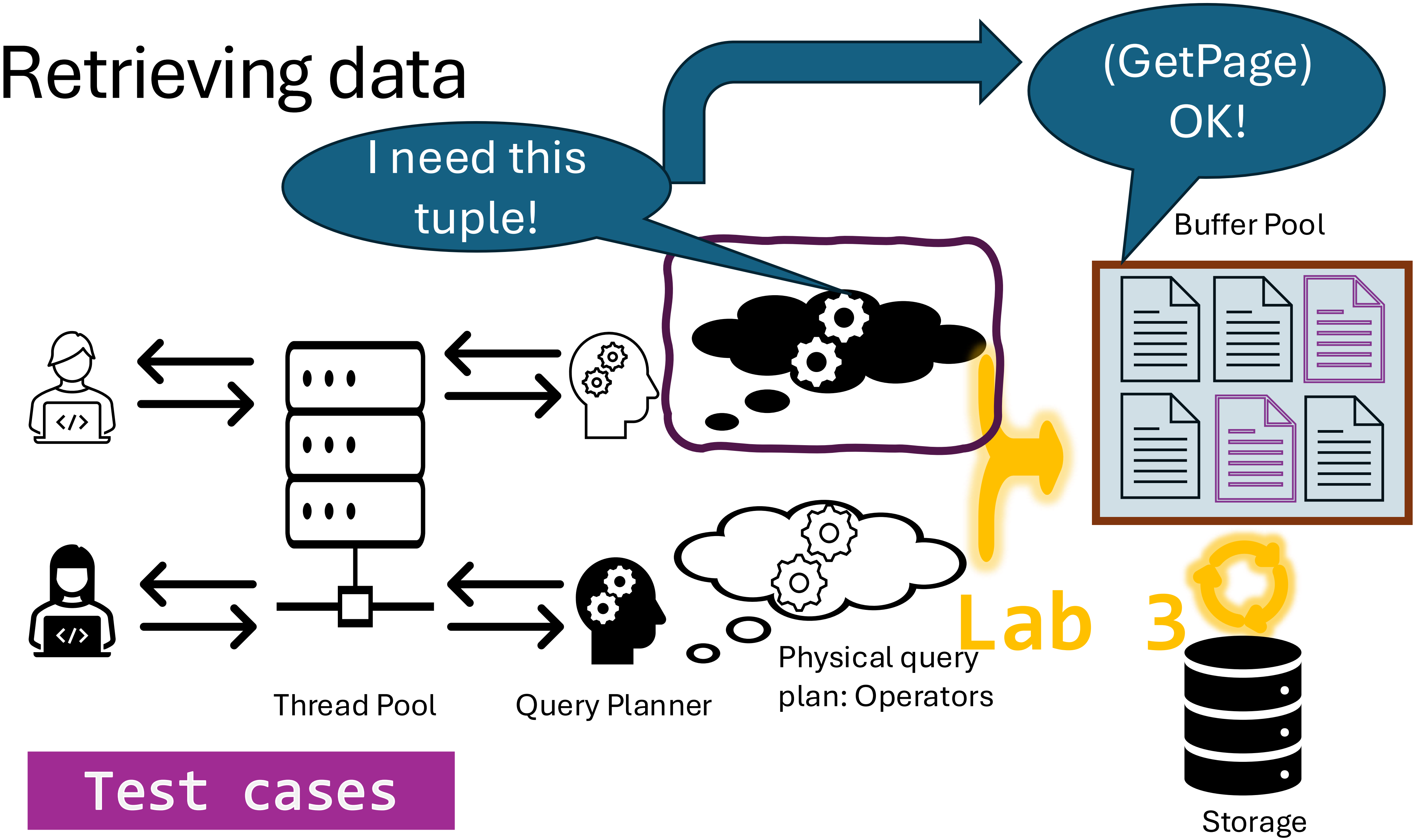
Test cases

Begin and commit



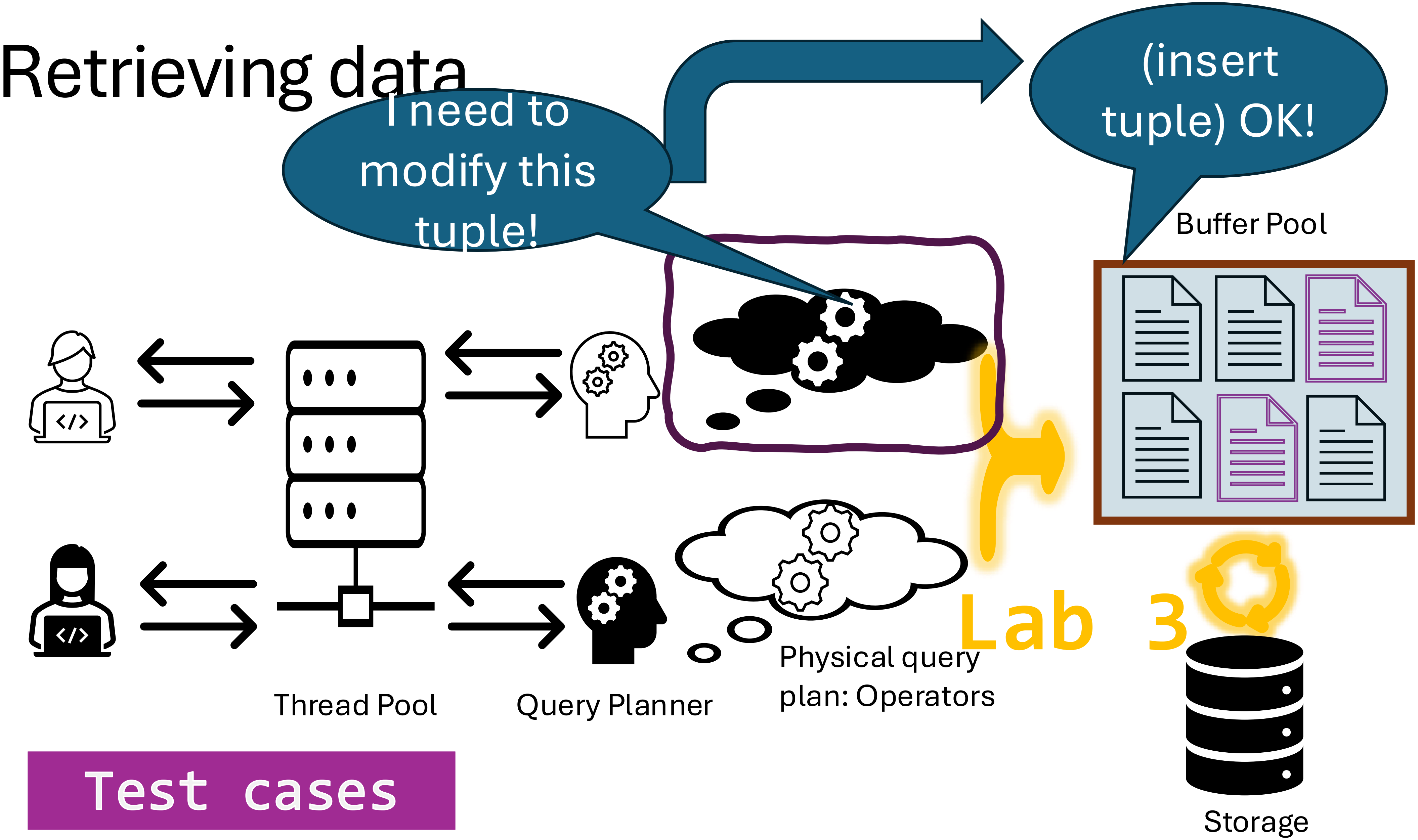
Test cases

Retrieving data

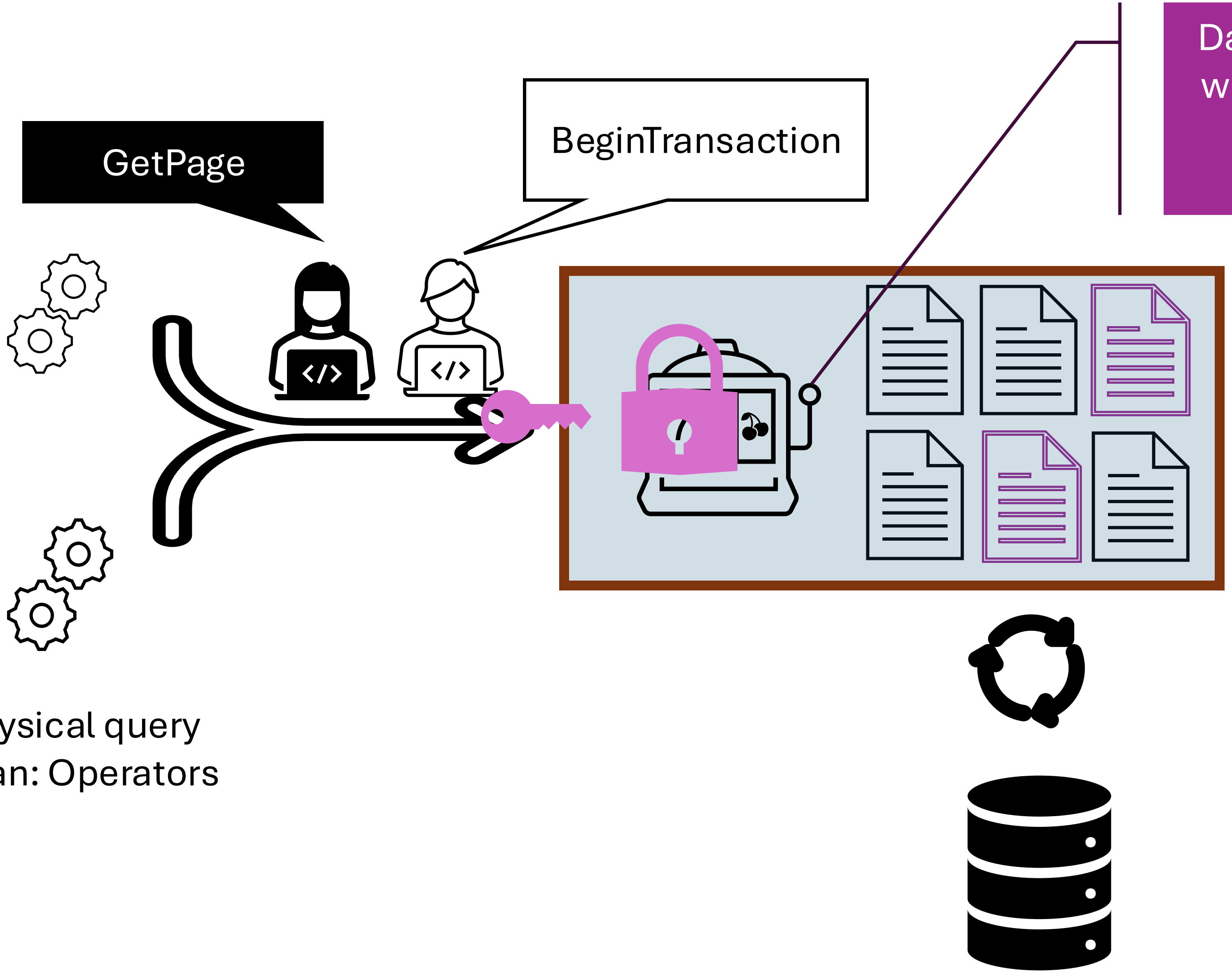


Test cases

Retrieving data



Test cases



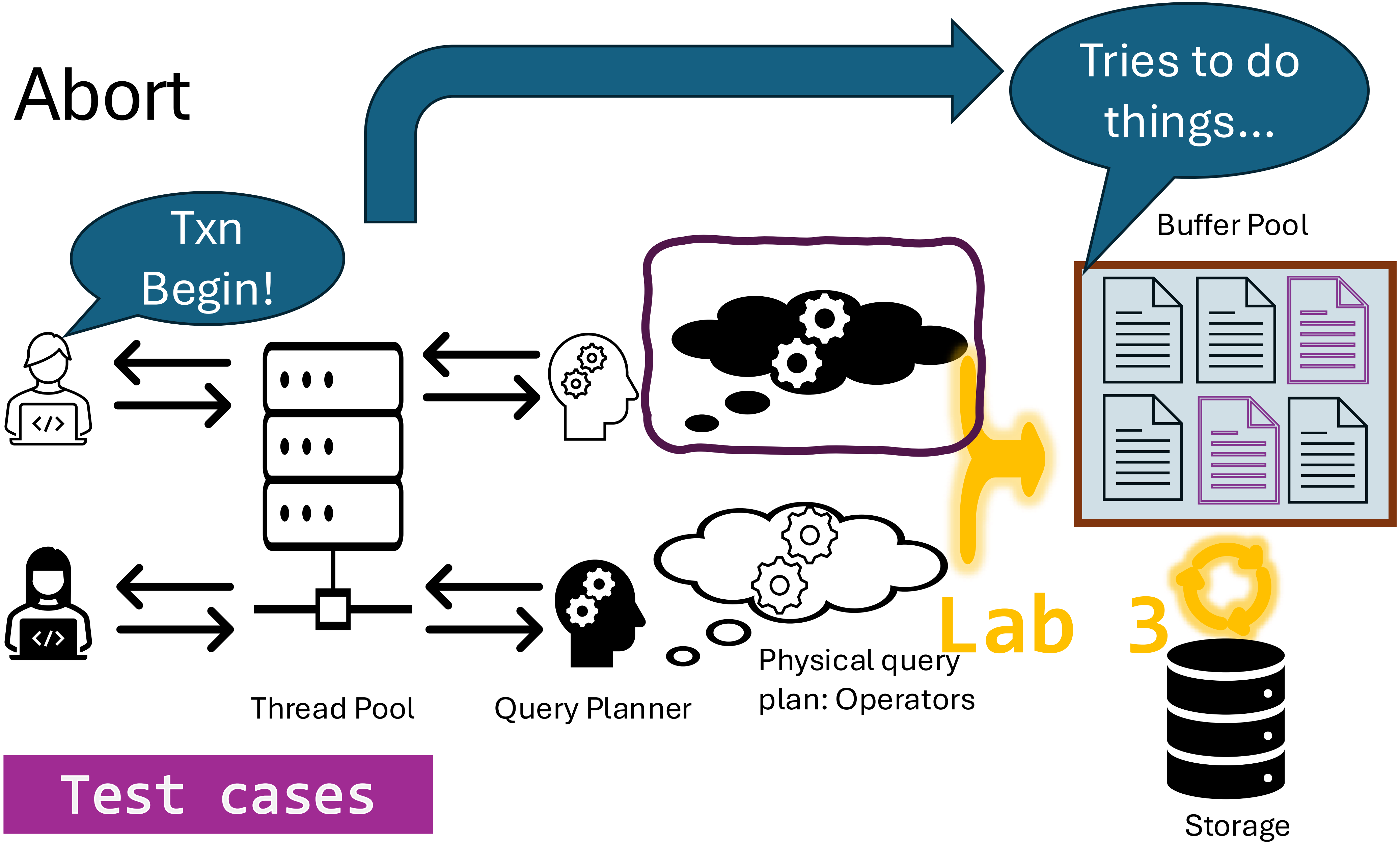
GetPage

BeginTransaction

Data structure that records what the buffer pool needs to record regarding transactions and locks

Physical query plan: Operators

Abort



Txn Begin!

Tries to do things...

Buffer Pool

Thread Pool

Query Planner

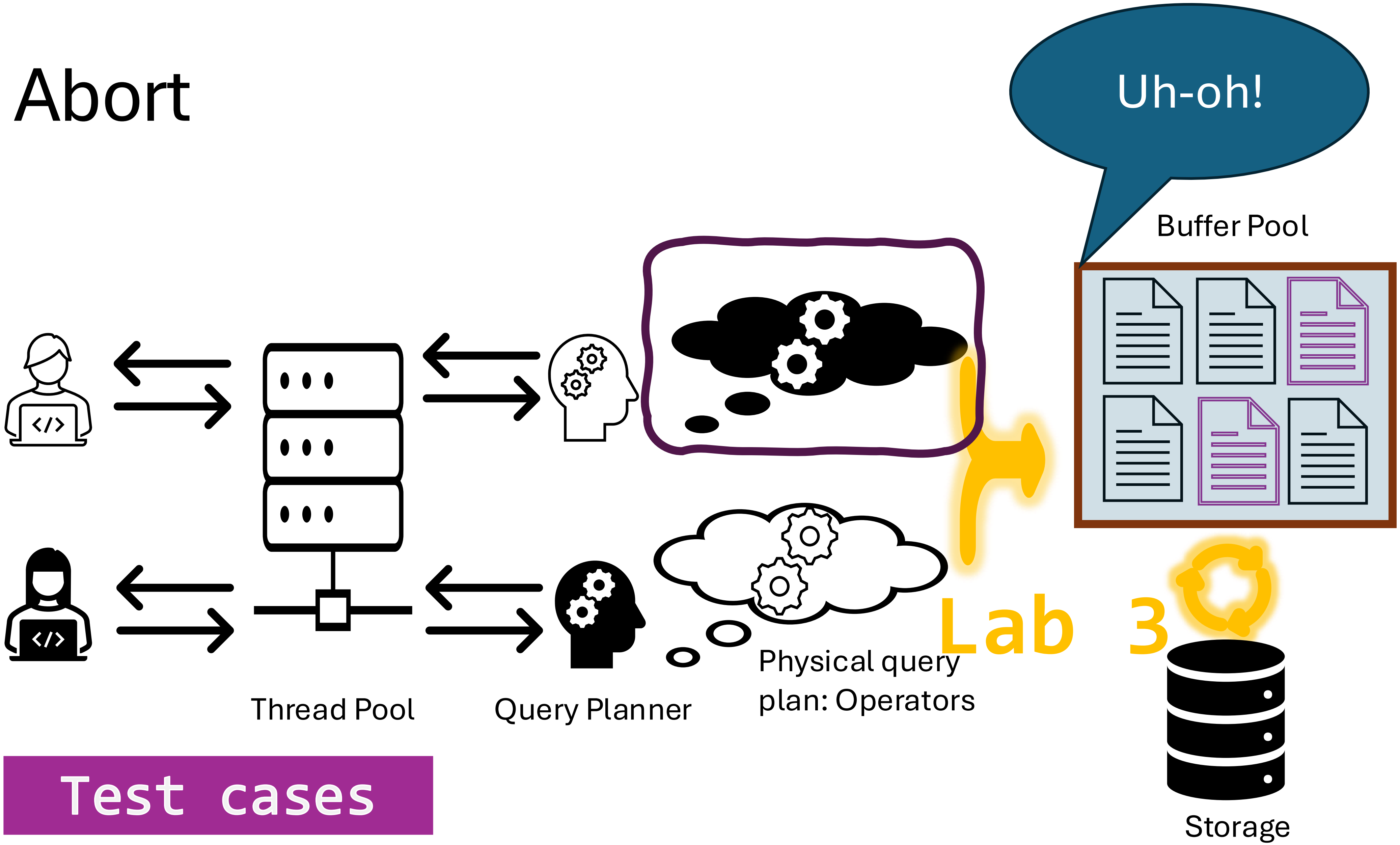
Physical query plan: Operators

Lab 3

Storage

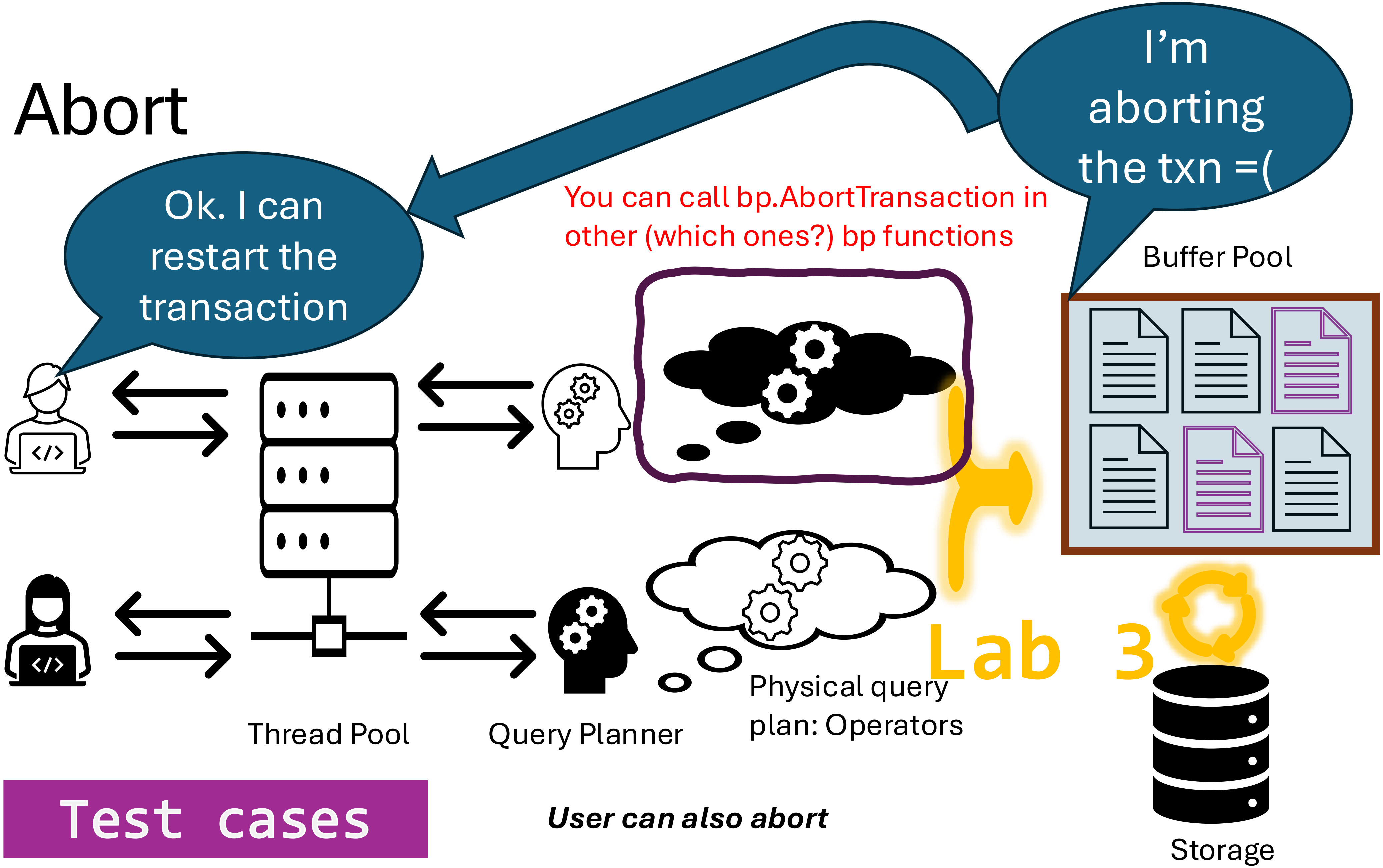
Test cases

Abort



Test cases

Abort



You can call bp.AbortTransaction in other (which ones?) bp functions

Test cases

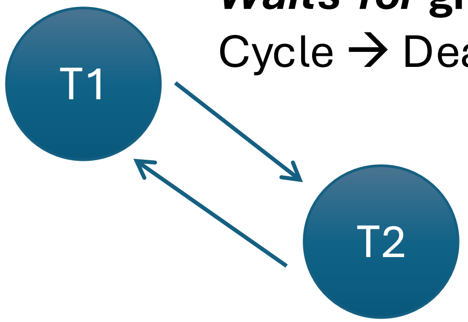
User can also abort

Deadlocks

- Possible for Txn_i to hold a lock Txn_j needs, and vice versa

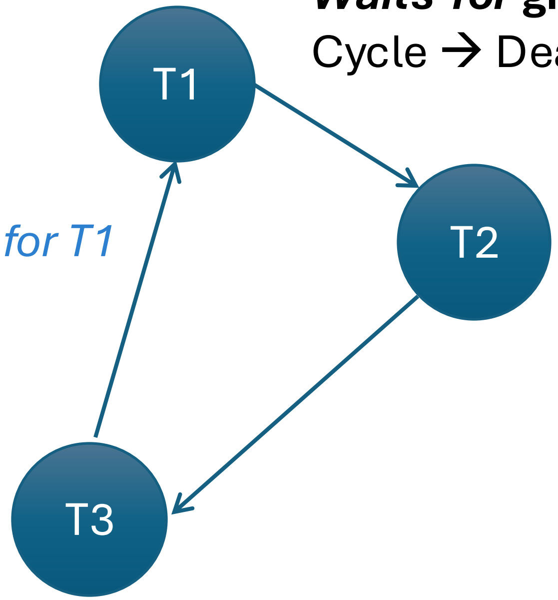
	T1	T2	
	RA		
	WA		
		RB	
		WB	
T1 waits for T2 →	RB		
	WB		
		RA	← *T2 waits for T1*
		WA	

***Waits-for* graph**
Cycle → Deadlock



Complex Deadlocks Are Possible

T1	T2	T3
RA		
WA		
		RC
	RB	
	WB	
		RA *← T3 waits for T1*
		WA
T1 waits for T2 → RB		
WB		
	RC *← T2 waits for T3*	
	WC	



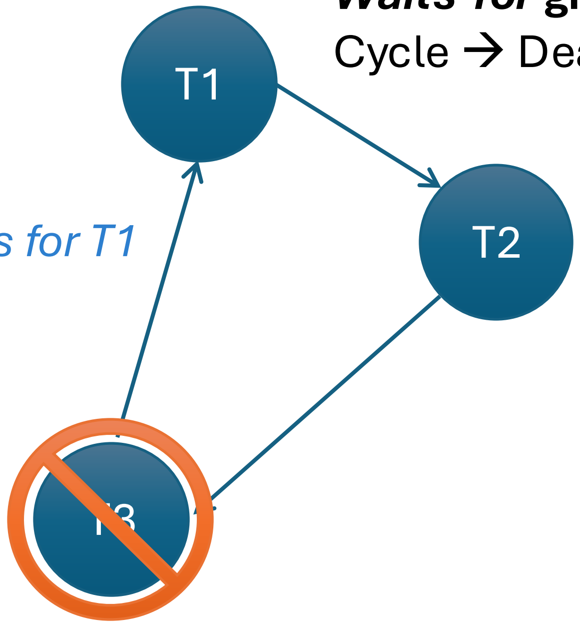
***Waits-for* graph**

Cycle → Deadlock

Resolving Deadlocks

- Solution: abort one of the transactions
  - Recall: users can abort too

T1	T2	T3
RA		
WA		
		RC
	RB	
	WB	
		RA ← *T3 waits for T1*
		WA
T1 waits for T2 → RB		
WB		
	RC ← ~~*T2 waits for T3*~~	
	WC	



Waits-for **graph**
Cycle → Deadlock

- Recall: Strict 2PL avoids cascading aborts (implement this!)

Strict Two-Phase Locking

- Can avoid cascading aborts by holding exclusive locks until end of transaction
- Ensures that transactions never read other transaction’s uncommitted data

Strict Two-Phase Locking Protocol

- Before every read, acquire a shared lock
- Before every write, acquire an exclusive lock (or "upgrade") a shared to an exclusive lock
- ~~Release locks only after last lock has been acquired, and ops on that object are finished~~
- Release *shared* locks only after last lock has been acquired, and ops on that object are finished
- Release *exclusive* locks only after the transaction commits
- Ensures cascadeless-ness

- You need to maintain a waits-for graph on the buffer pool level in lab 3 and detect and resolve deadlocks.
- There are also other implementations to resolve deadlocks.
- In any case, it requires you to abort the transaction from within the system
- **Return a non-nil error to signal to the caller that the transaction was aborted.**

Suggested code structure

- Transactions, on request of a buffer pool operation (i.e. calling a buffer pool API), look at buffer pool level data structure and decide whether it can proceed (acquiring the necessary page locks under strict 2L).
- If it cannot, try again later.
- If it can, proceed, make the corresponding changes in the buffer pool level data structures
- If step 1 detects a deadlock, resolve that.

All these steps need to be protected by necessary mutexes to avoid race conditions.

Buffer Pool Policy

- If we don’t write back dirty pages, they must be held in memory for the duration of the txn (We already told you to do this in Lab 1)

- A DB that writes back dirty pages is said to STEAL
- STEAL requires UNDO to remove uncommitted txns in event of crash
- For Lab 3, implement NO STEAL.
  - We assume that the system does not crash and skip recovery for the lab.
  - How do you deal with aborted transactions?

Some Committed Changes Not Written Back

- If we wrote back all pages at commit time, it would be slow!
  - Many random writes at commit time
- A DB that doesn’t force all writes at commit is NO FORCE
- This is complicated and requires implementing checkpointing and recovery. (You can do this for lab 4)
- NO FORCE requires REDO to install logged writes to DB in event of crash
- For lab 3, implement FORCE: dirty pages are written (flushed) back at commit time!

STEAL/NO FORCE ←→ UNDO/REDO

- If we STEAL pages, we will need to UNDO
- If we don't FORCE pages, we will need to REDO

	FORCE	**NO FORCE**
STEAL	UNDO	UNDO & REDO
NO STEAL	? UNDO	REDO

In GoDB, we do FORCE / NO STEAL, and assume DB won't crash between FORCE and COMMIT

- If we FORCE pages, we will need to be able to UNDO if we crash between the FORCE and the COMMIT
- For aborted transactions, also UNDO. ***How do you undo?***

Debugging tips

- Eliminate race conditions first
- Think about and enforce invariants. Modifications to buffer pool level data structures also need to be atomic!
  - For example, when a transaction commits, right before you release the mutexes and return from the function, waits-for graph should NOT have anything about this transaction. Think about similar properties that other data structures should respect at this exact time point.
  - Use more assertions, less print statements (unless this stretch of code is protected by a mutex and is a point of serialization): stdio is not atomic. stdio operations are also slow which can help ”serialize” your code and hide bugs.
- You should not need to spawn new threads (goroutines). Each transaction is on one thread.
- Start Early - Debugging concurrency can be hard!!!