CSC7072: Databases, fall 2015

Dr. Kim Bauters

transactions

Transactions

definition

what is a transaction?

it is a *unit of execution* that *accesses and/or modifies* data

for example, transferring money from A to B is a transaction:

1. read(A)
2. A = A – 50
3. write(A)
4. read(B)
5. B = B + 50
6. write(B)

requires all 6 steps!

two issues to deal with

failures, *e.g.* hardware crash

concurrency, *e.g.* other transfer from B

Transactions

essential properties

what is ACID?

a transaction should satisfy the ACID properties

- atomicity
- consistency
- isolation
- durability

atomicity

either *all* changes are correctly reflected in the DB, *or none* are

1. read(A)
2. A = A – 50
3. write(A)
4. read(B)
5. B = B + 50
6. write(B)

if only steps 1–3 are executed, then money would be lost!

Transactions

essential properties

what is ACID?

consistency

consistency constraints (*e.g.* allowable values, FK consistency, …) must be guaranteed *before*, and *after* a transaction executes.

1. read(A)
2. A = A – 50
3. write(A)
4. read(B)
5. B = B + 50
6. write(B)

example: before and after execution of the transaction we have that the sum of A and B is unchanged i.e. no money disappears

during a transaction the database may be **temporarily** inconsistent!

Transactions

essential properties

what is ACID?

isolation

each transaction is unaware and unaffected by any other transactions that is executed *concurrently*

1. read(A)
2. A = A – 50
3. write(A)

1. print(A+B)

hint: DB is inconsistent at this stage

4. read(B)
5. B = B + 50
6. write(B)

yet big speed gains can be achieved through (careful!) concurrent execution

Transactions

essential properties

what is ACID?

durability

once a transaction has been committed, the changes must *persist* even if the system crashes afterwards

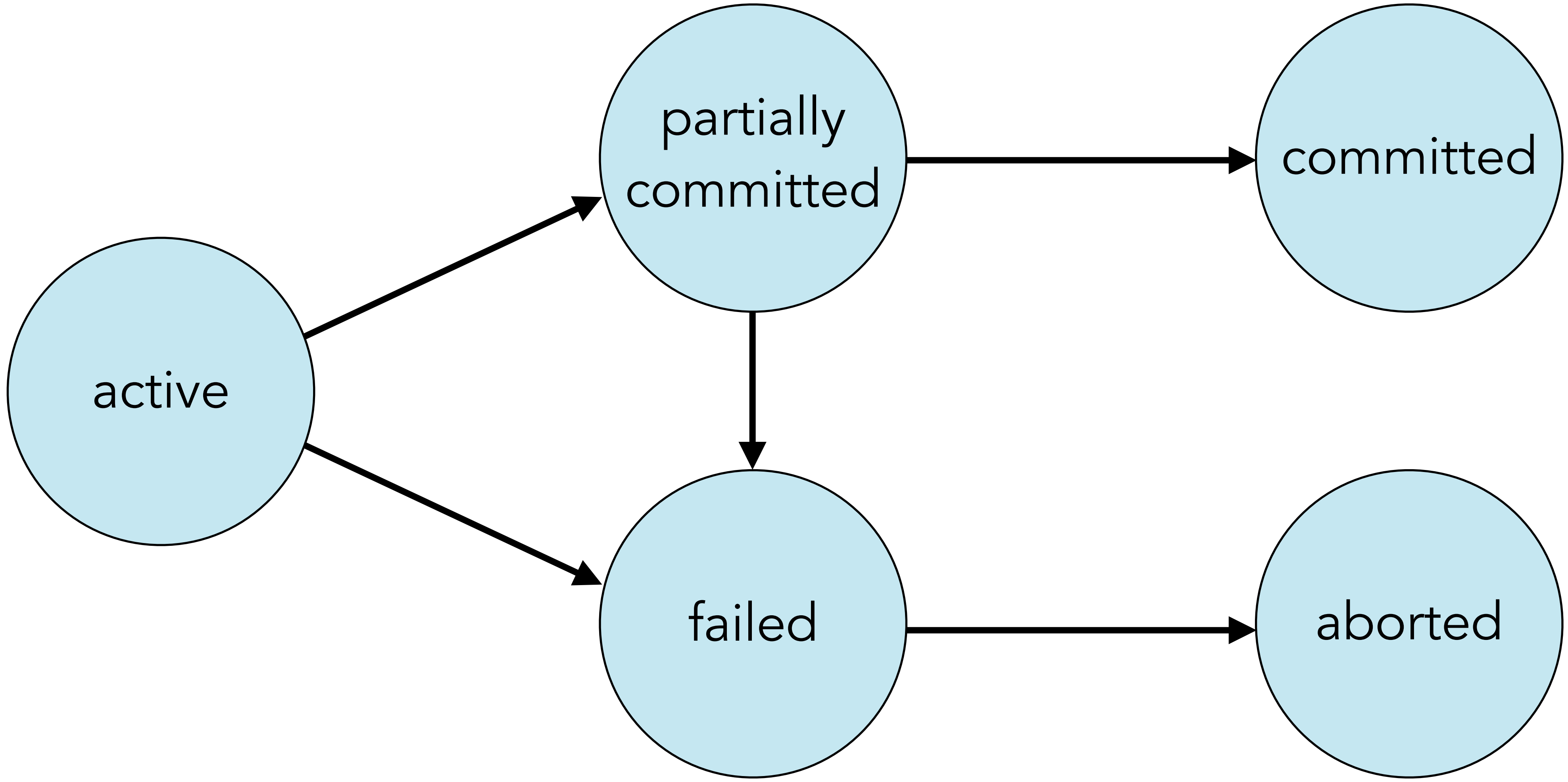
when transactions satisfy the ACID properties, then we have a guarantee that database transaction are processed reliably

these properties are widespread, and found in various places
for example, your file system (NTFS, EXT#, HFS+) behaves as a DB!

Transactions

states of a transaction

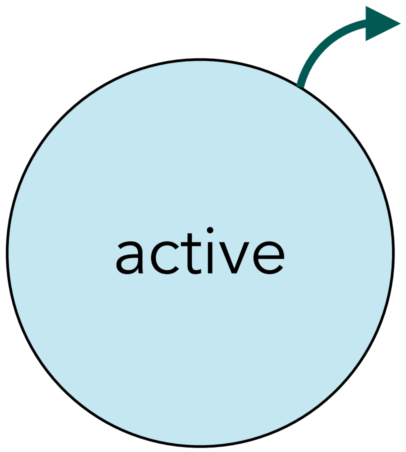
stages of a transaction



Transactions

states of a transaction

stages of a transaction

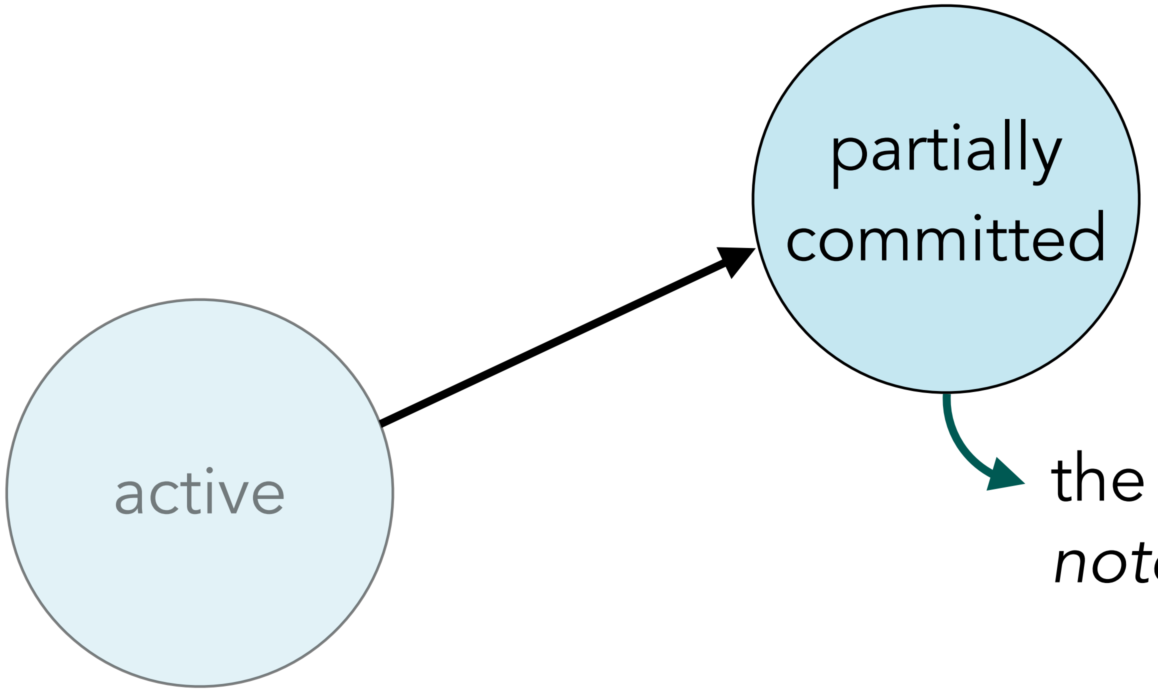


this is the initial state
a transaction stays in this state while executing

Transactions

states of a transaction

stages of a transaction

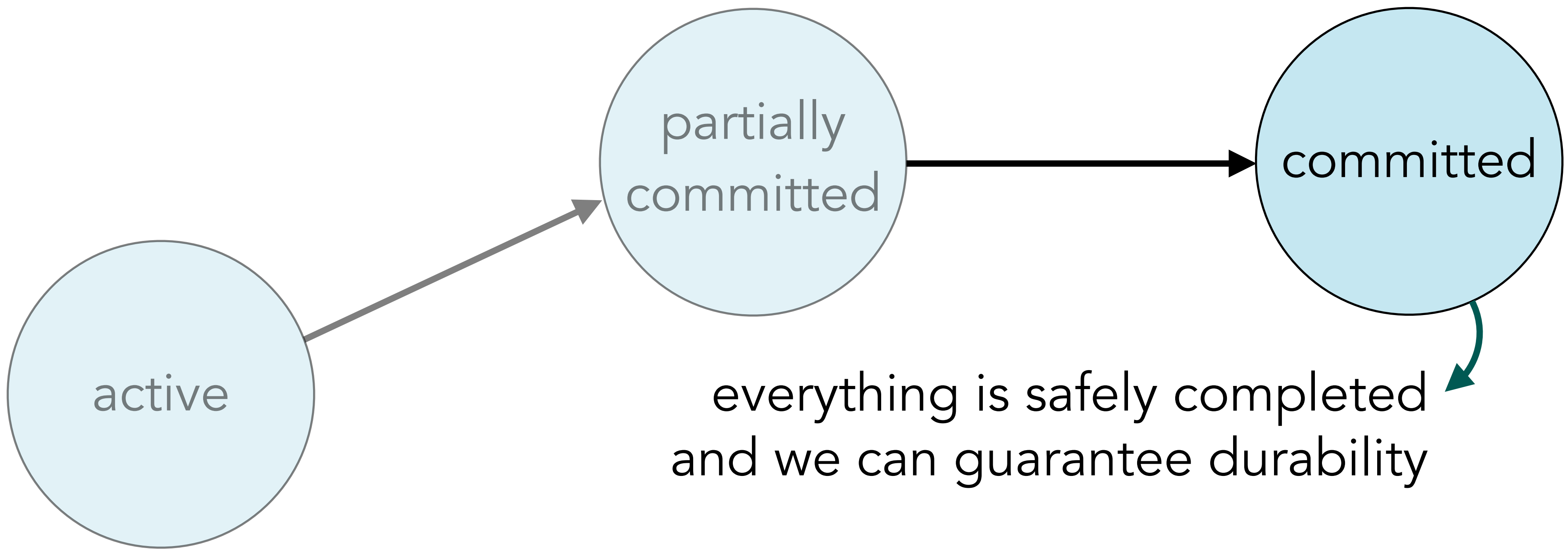


the last statement has been executed
note: changes might not be saved yet!

Transactions

states of a transaction

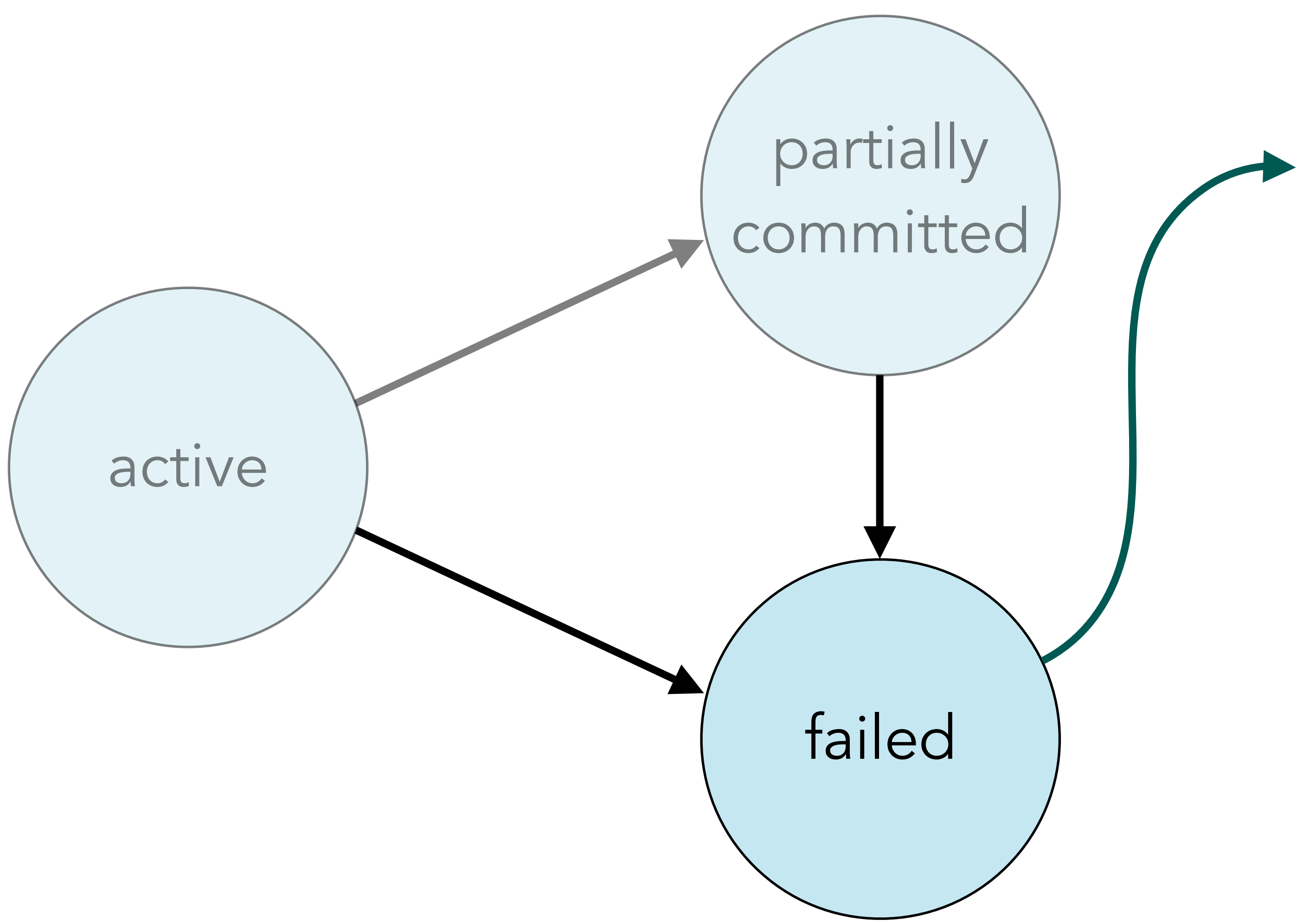
stages of a transaction



Transactions

states of a transaction

stages of a transaction

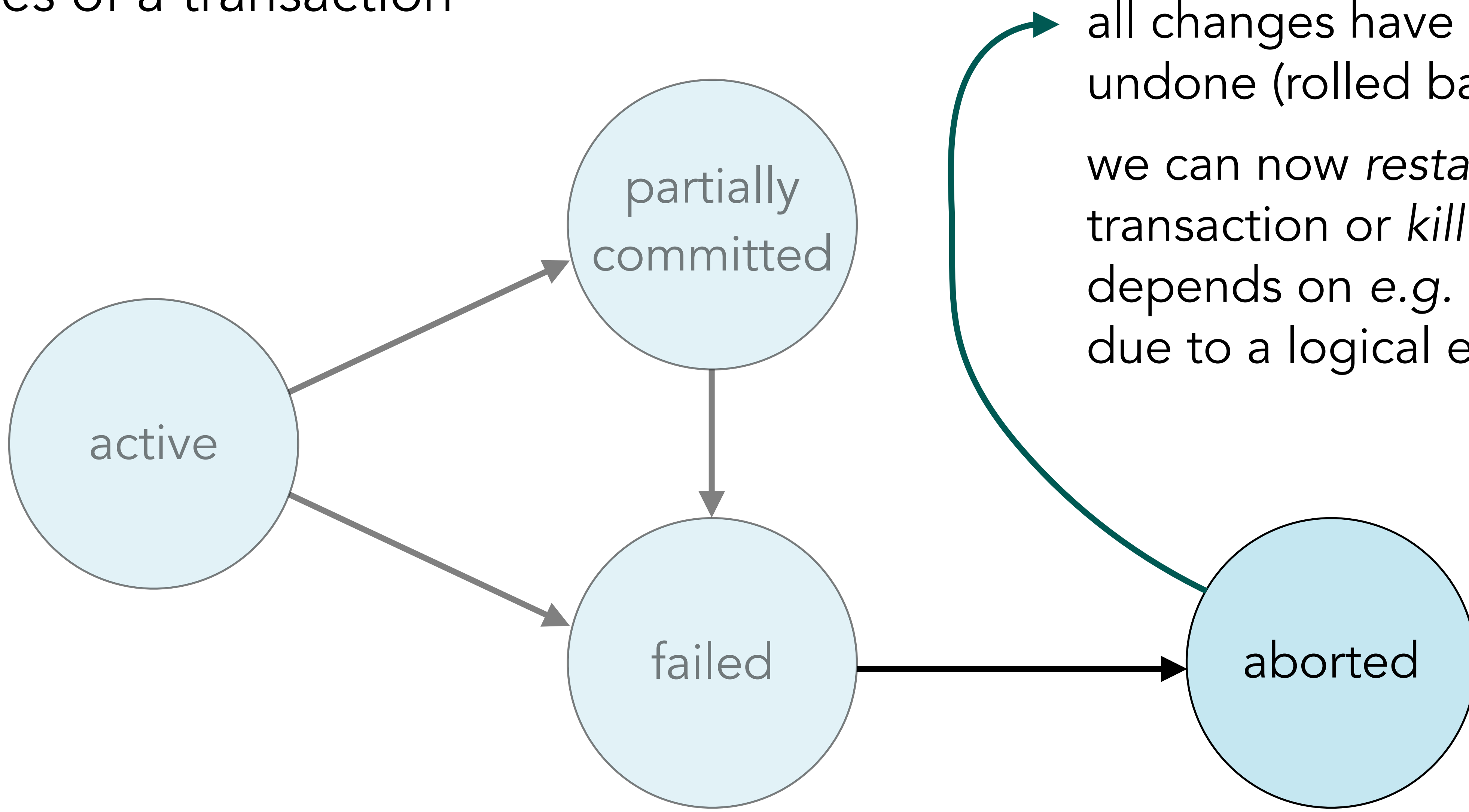


something caused the transaction to fail, *e.g.* a hardware error, a logical fault …

Transactions

states of a transaction

stages of a transaction



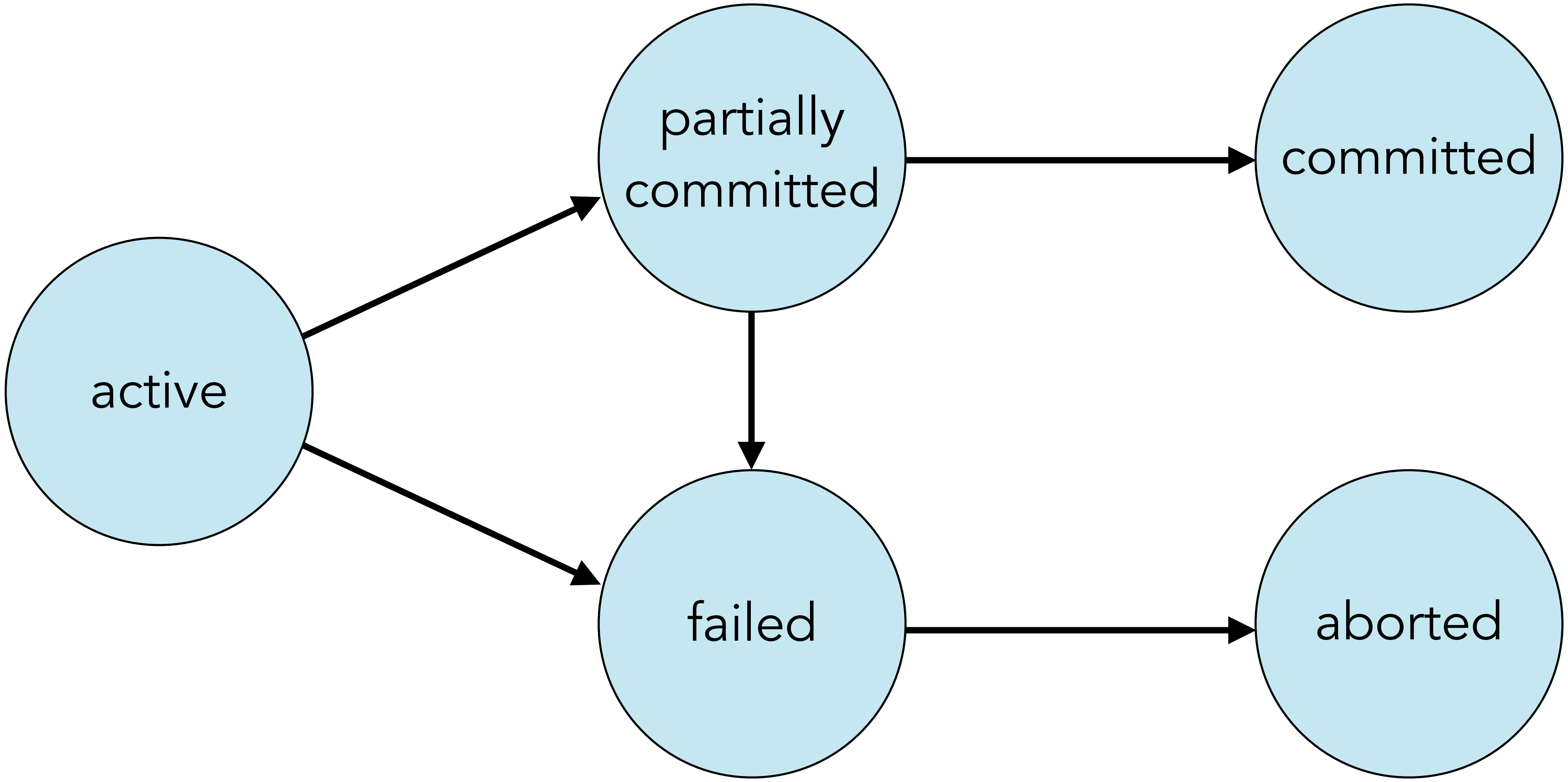
all changes have been undone (rolled back)

we can now *restart* the transaction or *kill* it (choice depends on *e.g.* if it was due to a logical error)

Transactions

states of a transaction

stages of a transaction



Transactions

concurrency: the need for speed

isolation: why even allow concurrency?

- better CPU/memory/storage utilisation
  e.g. if one transaction is CPU intensive, another is memory intensive, then running them both maximally uses capabilities
- reduced average response time
  i.e. short transactions need not wait behing long ones

concurrency control schemes are used to guarantee isolation

achieved by controlling the interaction between transactions in order to prevent them from destroying the consistency

Transactions

concurrency: schedules

key notion is the idea of a *schedule*

represents the chronological order in which instructions are handled

- clearly, it must consist of *all* the instructions for all transactions
- must preserve order of instructions as found in a transaction

hence, a schedule is only interweaves transactions!

for clarity, we henceforth explicitly write commit to move to committed state

Transactions

schedules: example

simple example: A transfers £50 to B (T_1); A transfers £20 to B (T_2)

T_1	T_2
read(A)	
A:= A – 50	
write(A)	
read(B)	
B:= B + 50	
write(B)	
commit	
	read(A)
	A:= A – 20
	write(A)
	read(B)
	B:= B + 20
	write(B)
	commit

a *serial schedule* where T_1 is followed by T_2

Transactions

schedules: example

simple example: A transfers £50 to B (T_1); A transfers £20 to B (T_2)

	T_1	T_2
£100	read(A)	
(£50)	A:= A – 50	
£50	write(A)	
	read(B)	
	B:= B + 50	
	write(B)	
	commit	
£50		read(A)
(£30)		A:= A – 20
£30		write(A)
		read(B)
		B:= B + 20
		write(B)
		commit

a *serial schedule* where T_1 is followed by T_2

Transactions

schedules: alternative example

alternative serial schedule:

T_1	T_2
	read(A)
	A:= A – 20
	write(A)
	read(B)
	B:= B + 20
	write(B)
	commit
read(A)	
A:= A – 50	
write(A)	
read(B)	
B:= B + 50	
write(B)	
commit	

a *serial schedule* where T_2 is followed by T_1

Transactions

schedules: concurrent example

a working concurrent scheme:

T_1	T_2
read(A)	
A:= A – 50	
write(A)	
	read(A)
	A:= A – 20
	write(A)
read(B)	
B:= B + 50	
write(B)	
commit	
	read(B)
	B:= B + 20
	write(B)
	commit

a *serial schedule* where T_1 is followed by T_2

Transactions

schedules: problematic example

a problematic concurrent scheme:

T_1	T_2
read(A)	
A:= A – 50	
	read(A)
	A:= A – 20
	write(A)
write(A)	
read(B)	
B:= B + 50	
write(B)	
commit	
	read(B)
	B:= B + 20
	write(B)
	commit

a *serial schedule* where T_1 is followed by T_2

Transactions

schedules: problematic example

a problematic concurrent scheme:

	T_1	T_2
£100	read(A)	
(£50)	A:= A – 50	
£100		read(A)
(£80)		A:= A – 20
£80		write(A)
£50	write(A)	
	read(B)	
	B:= B + 50	
	write(B)	
	commit	
		read(B)
		B:= B + 20
		write(B)
		commit

a *serial schedule* where T_1 is followed by T_2

Transactions

safe concurrency

what is correct concurrent execution?

core idea: any correct schedule has the same effect as a schedule without concurrent execution, *i.e.* a serial schedule

main question: when is a schedule *serialisable*?

or: which schedules can be turned into equivalent serial schedules?

some simplifying assumptions:

- we only consider operations *open* and *write*
- transactions can perform arbitrary computations in between

Transactions

safe concurrency: conflict

the problem with *conflict*

assume we have value A, and two transactions, T_1 and T_2 where T_1 has the instruction IN_1, and T_2 has the instruction IN_2

✓ IN_1 = read(A) IN_2 = read(A)
the order does not matter; IN_1 or IN_2 can both be first

Transactions

safe concurrency: conflict

the problem with *conflict*

assume we have value A, and two transactions, T_1 and T_2 where T_1 has the instruction IN_1, and T_2 has the instruction IN_2

✓ IN_1 = read(A) IN_2 = read(A)
the order does not matter; IN_1 or IN_2 can both be first

✗ IN_1 = read(A) IN_2 = write(A)
the order **does** matter; if IN_1 goes first it does not read the result of IN_2
if IN_2 goes first, then IN_1 it reads result of IN_2

✗ IN_1 = write(A) IN_2 = read(A)
the order **does** matter; similar as above

Transactions

safe concurrency: conflict

the problem with *conflict*

assume we have value A, and two transactions, T_1 and T_2 where T_1 has the instruction IN_1, and T_2 has the instruction IN_2

✓ IN_1 = read(A) IN_2 = read(A)
the order does not matter; IN_1 or IN_2 can both be first

✗ IN_1 = read(A) IN_2 = write(A)
the order **does** matter; if IN_1 goes first it does not read the result of IN_2
if IN_2 goes first, then IN_1 it reads result of IN_2

✗ IN_1 = write(A) IN_2 = read(A)
the order **does** matter; similar as above

✗ IN_1 = write(A) IN_2 = write(A)
the order **does** matter; only last value of A actually stored in DB!

Transactions

safe concurrency: conflict serialisability

the problem with *conflict*, cont.

when working on the *same data*, ❶
and at least one of the instructions is *write(...)*, ❷
then we say that the instructions are in *conflict*

leads us to *conflict serialisability*:

if we can turn a schedule into a serial one,
only by swapping instructions that *do not* have any conflict,
then the original schedule is said to be *conflict serialisable*

Transactions

your turn

is the following schedule of T1 and T2 conflict-serialisable?

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Transactions

your turn

is the following schedule of T1 and T2 conflict-serialisable?

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
	read(B)
write(B)	
	write(B)

Transactions

your turn

is the following schedule of T1 and T2 conflict-serialisable?

T_1	T_2
read(A)	
	read(A)
read(A)	

Transactions

your turn

is the following schedule of T1 and T2 conflict-serialisable?

T_1	T_2
write(A)	
	write(A)
write(A)	

Transactions

when things go wrong

if things go wrong: the *failed* state

remember atomicity:

either *all* changes are correctly reflected in the DB, *or none* are

what happens if a transaction fails midway?

1. read(A)
2. A = A – 50
3. write(A)
4. read(B)
5. B = B + 50
6. write(B)

a *rollback* is initiated to undo all changes so far

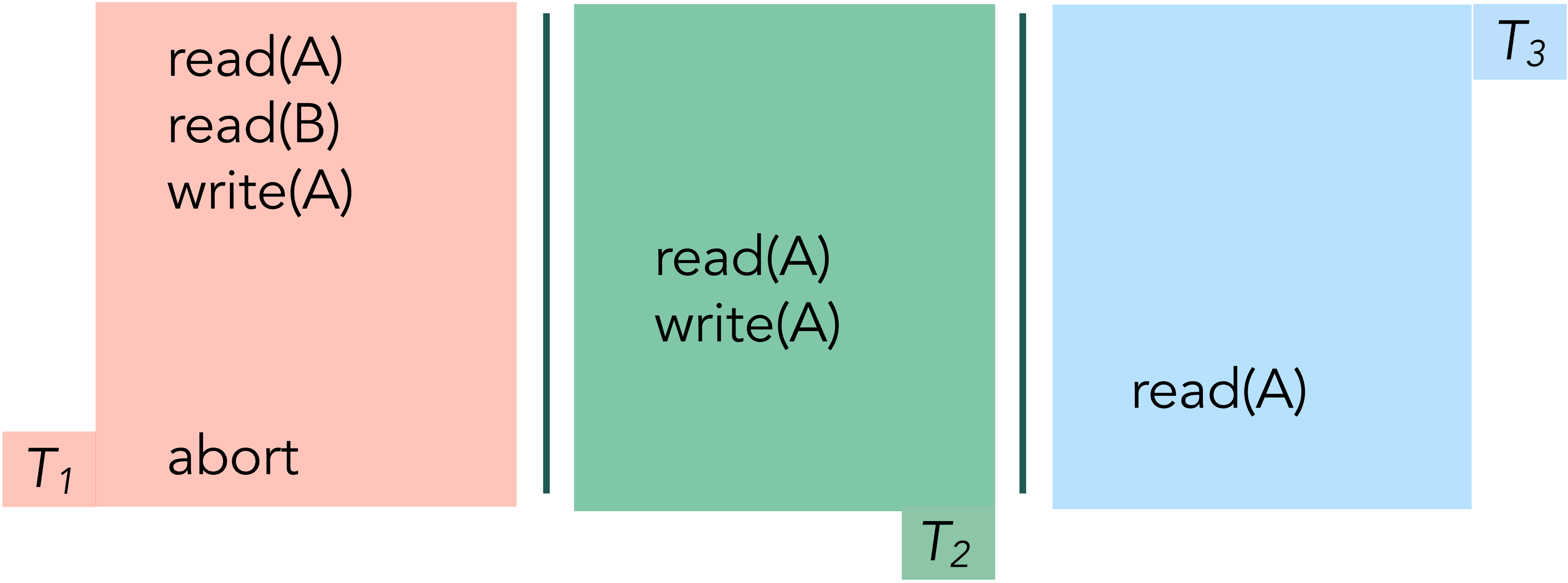
care must be taken to avoid steps that cannot be turned back!
e.g. sending out emails, displaying partial results on screen …

Transactions

when things go wrong … and cause a torrent

when things get worse: nested *failed* states

sometimes, a *cascading rollback* is triggered:



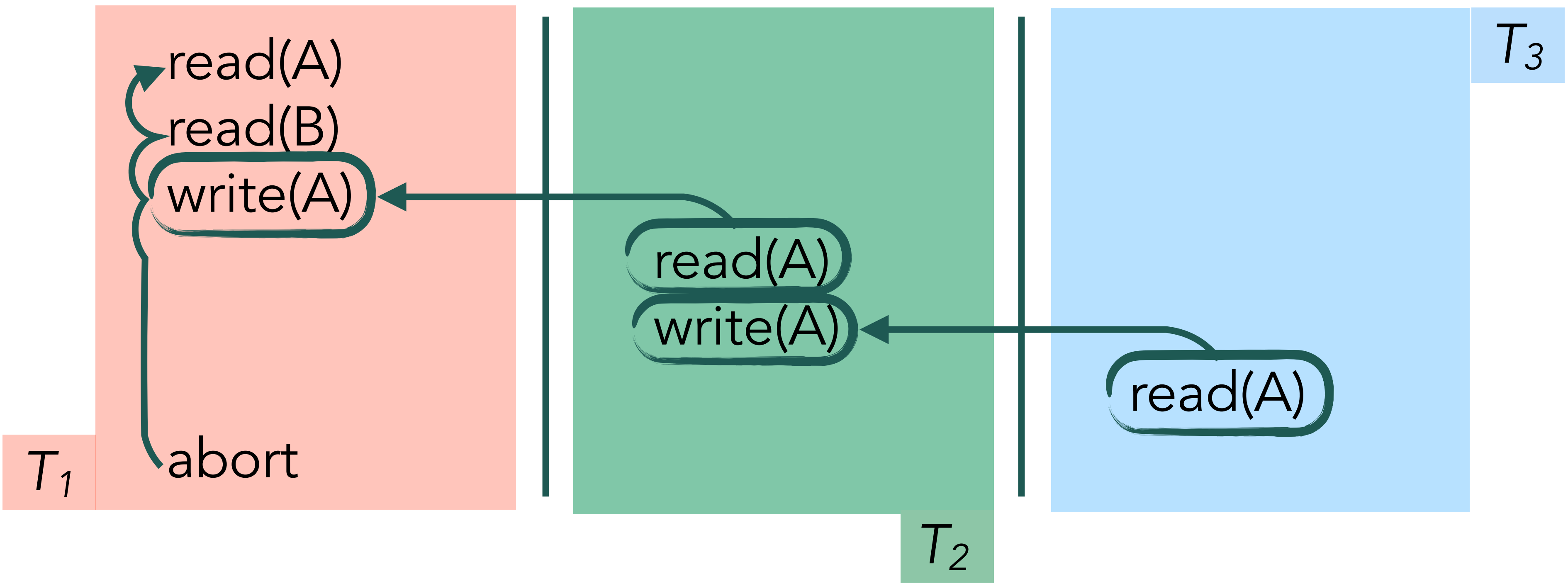
can lead to the undoing of significant amounts of work

Transactions

when things go wrong … and cause a torrent

when things get worse: nested *failed* states

sometimes, a *cascading rollback* is triggered:



can lead to the undoing of significant amounts of work

Transactions

mid summary

before we continue: transactions concepts

- transactions need to satisfy the *ACID* properties
- *isolation* plays a role when we allow *concurrency*
  we want concurrency as it can add a lot of performance
- *conflict-serialisability* is one approach to guarantee isolation
- transactions can be in one of 5 *states*
- failed transactions require a rollback, due to *atomicity*
  some things cannot be undone; we try to do those after a commit
- *cascading rollbacks* are costly, and can undo a lot of work
  often we restrict ourselves to cascadeless schedules

Transactions

why do we need recovery?

why again did we need transactions/recovery?

a lot can go wrong:

- transaction error

  something went wrong with the transaction;this can be e.g. an explicit abort, or a system error such as a deadlock (A waits for B, B waits for A)

- system crash

  power failure or software failure causes the system to go down

- disk failure

  a disk head crashes, memory becomes corrupt …

idea: make a copy?

Transactions

log-based recovery

log-based recovery: overview

every DBMS has support for transactions,
SQL assumes COMMITS implicitly after each statement

log-based recovery is complete to allow successful recovery
but also light enough to not have too much overhead

log-based recovery is very common in DBMS (and file systems, …)

idea: keep track in log of all changes (not a full copy of original)

Transactions

log-based recovery

log-based recovery: overview

a *log* consists of *log records*; maintains record of updates to DB
log record created when:

- a transaction T_k starts, to register it:
  <T_k start>
- **before** T_k executes write(X), to track the update:
  <T_k, X, V_{before}, V_{after}>
  where V_{before} and V_{after} are *resp.* the value of X as it is now in the DB (*i.e.* before), and the value to change X into
- when transaction T_k finishes: ← transaction is committed here
  <T_k commit>

Transactions

log-based recovery: example

example: transfer (T_1) and withdrawal transaction (T_2)

log records	**DB writes**	**storage writes**
<T_1 start>		
<T_1, A, 1000, 950>		
	A = 950	
<T_2 start>		
<T_1, B, 775, 825>		
	B = 825	
<T_1 commit>		BL_B
<T_2, C, 700, 600>		
	C = 600	BL_A, BL_C
<T_2 commit>		

Transactions

storage in a computer

storage writes? a primer on storage

a computer has different types of storage:

- volatile storage
  storage that disappears as the result of system crashes
  e.g. main memory, cache memory (think of copy-paste)

- non-volatile storage
  storage that survives system crashes, *e.g.* hard disk, SSD
  but it can still fail in other ways, such as a disk crash

- permanent storage
  mythical form of storage that survives all crashes
  mimicked in clever ways using various backups

Transactions

storage in a computer: where DB sits

storage writes? a primer on storage

a computer has different types of storage:

- volatile storage
  storage that disappears as the result of system crashes
  e.g. main memory, cache memory (think of copy-paste)
- non-volatile storage
  storage that survives system crashes, *e.g.* hard disk, SSD
  but it can still fail in other ways, such as a disk crash
- permanent storage
  mythical form of storage that survives all crashes
  mimicked in clever ways using various (networked) backups

DB

log

slow

Transactions

storage in a computer: where DB sits

storage writes? a primer on storage

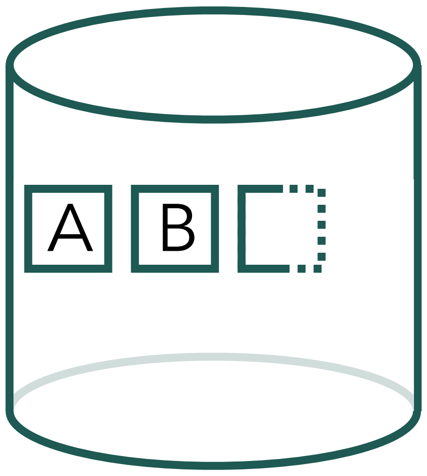
too volatile
(remember durability?)

work area T_k

A = A – 50
write(A)
read(B)

volatile storage

too slow
(remember efficiency?)



non-volatile storage

Transactions

storage in a computer: where DB sits

storage writes? a primer on storage

too volatile
(remember durability?)

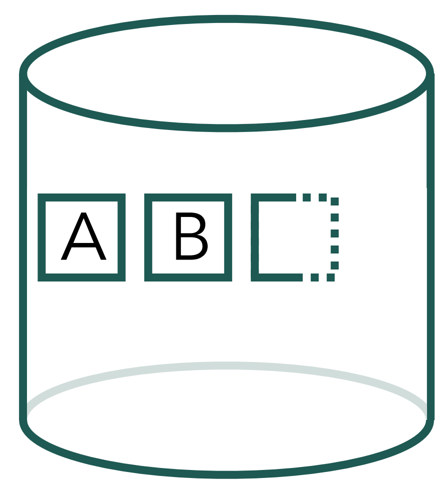
buffer

work area T_k

A = A – 50
write(A)
read(B)

volatile storage

too slow
(remember efficiency?)



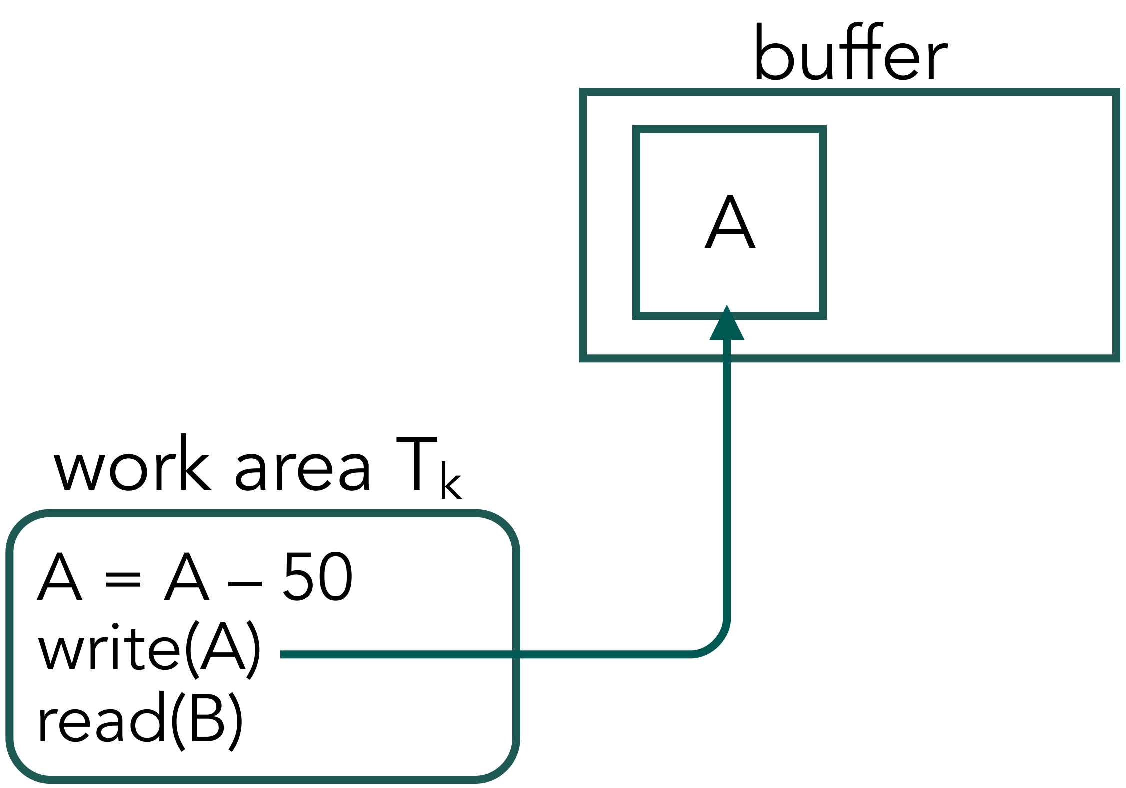
non-volatile storage

Transactions

storage in a computer: where DB sits

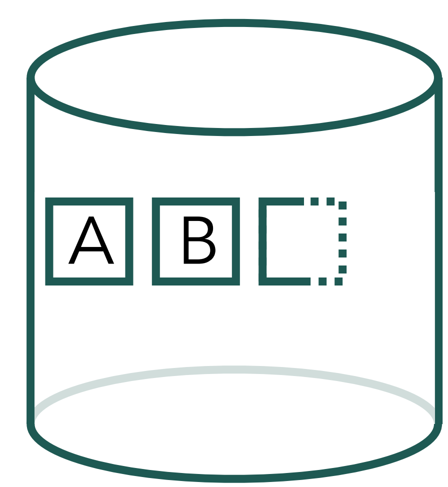
storage writes? a primer on storage

too volatile
(remember durability?)



volatile storage

too slow
(remember efficiency?)

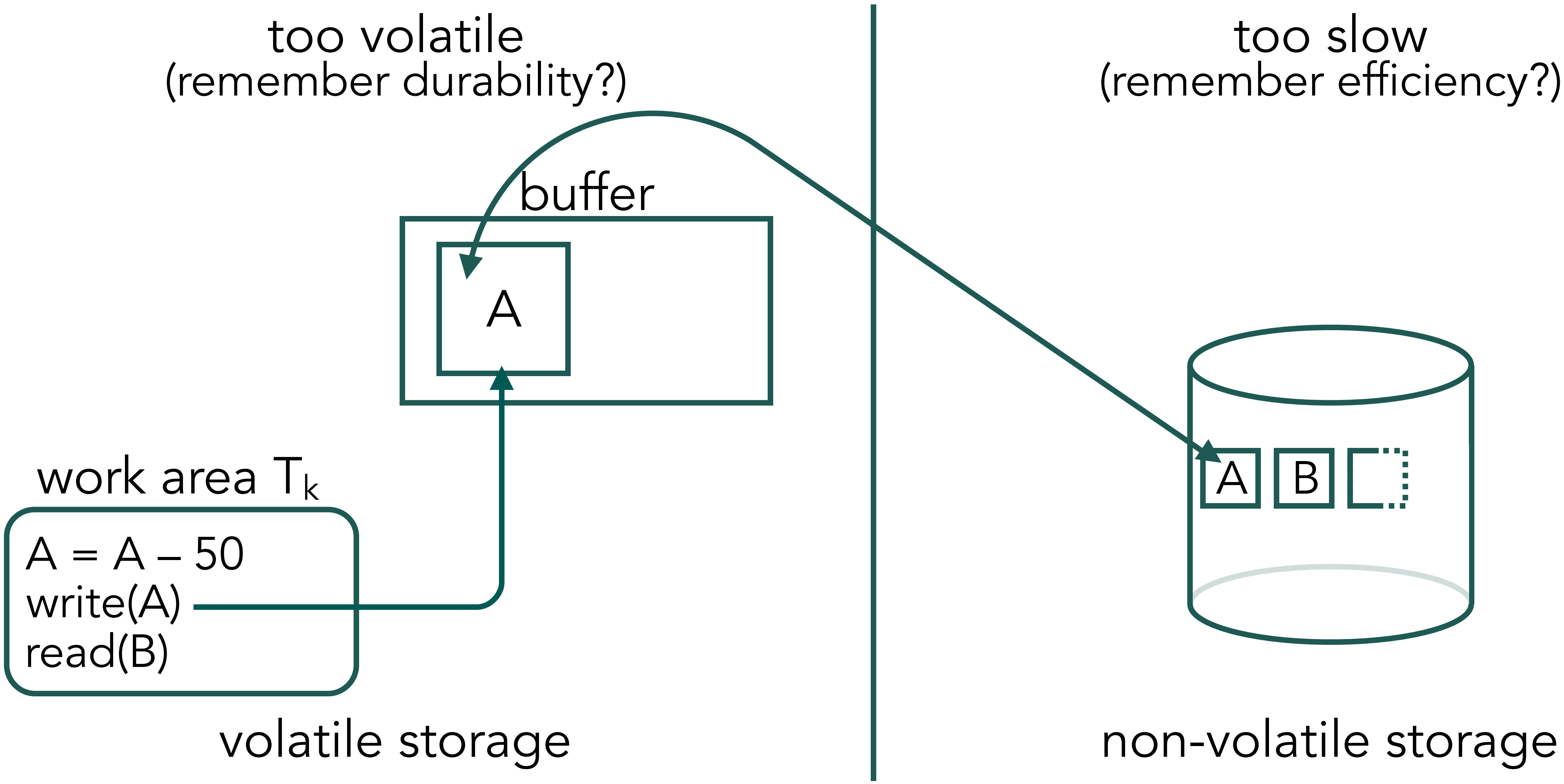


non-volatile storage

Transactions

storage in a computer: where DB sits

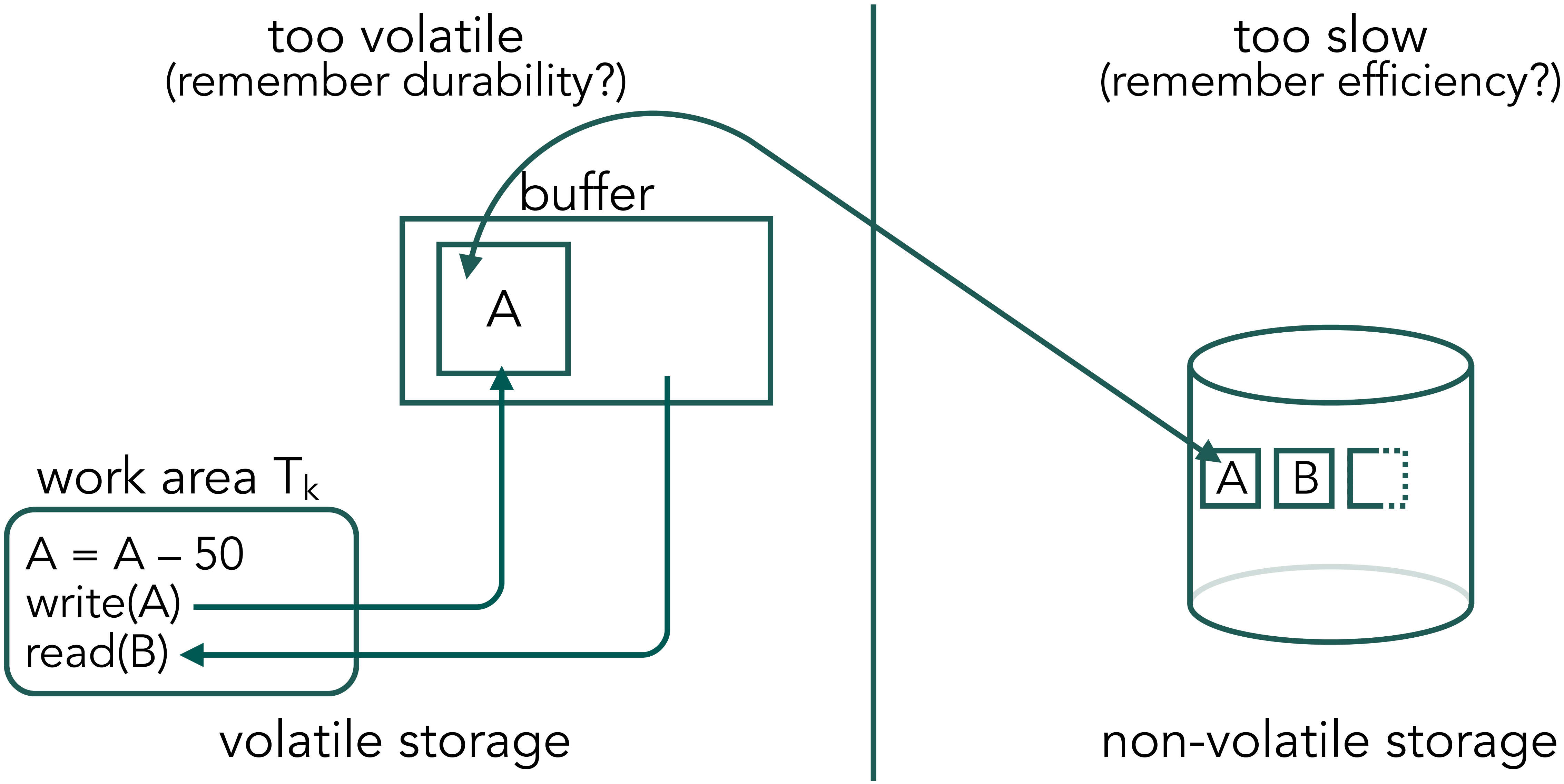
storage writes? a primer on storage



Transactions

storage in a computer: where DB sits

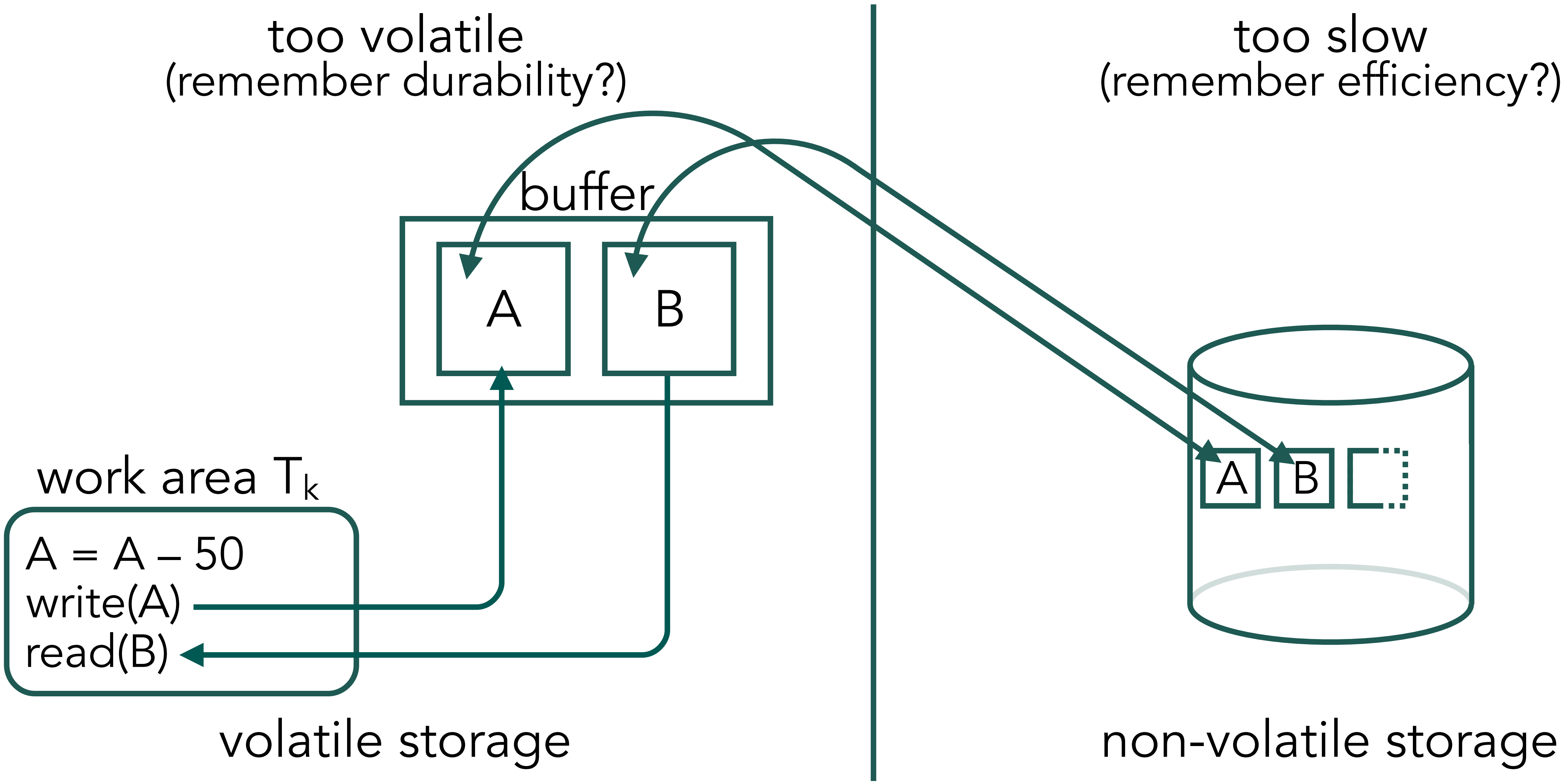
storage writes? a primer on storage



Transactions

storage in a computer: where DB sits

storage writes? a primer on storage



Transactions

log-based recovery: recovery strategy

recovering from a fault in log-based recovery

we have two options to choose from:

1. undo changes: write old value of <T_k, X, V_{before}, V_{after}>
2. redo changes: write new value of <T_k, X, V_{before}, V_{after}>

idea: undo all uncommited, then redo all committed transactions

how does *undo* work?

move **backward** over the log file and change to V_{before}

each time, add a log record <T_k, X, V_{before}> to record the change

when T_k is fully undone (we reach <T_k start>), write <T_k abort>

Transactions

log-based recovery: recovery strategy

recovering from a fault in log-based recovery

we have two options to choose from:

1. undo changes: write old value of $<T_k, X, V_{before}, V_{after}>$
2. redo changes: write new value of $<T_k, X, V_{before}, V_{after}>$

idea: undo all uncommited, then redo all committed transactions

how does *redo* work?

- move **forward** over the log file and change to V_{after}
- no log record is created (it's all there already)

Transactions

log-based recovery: recovery strategy

example: log based recovery

log records	**DB writes**	**storage writes**
<T_1 start>		
<T_1, A, 1000, 950>		
	A = 950	
<T_2 start>		
<T_1, B, 775, 825>		
	B = 825	
<T_1 commit>		BL_B
<T_2, C, 700, 600>		

BL_B = 825 BL_A = 1000 BL_C = 700 *inconsistent/lost update!*

Transactions

log-based recovery: recovery strategy

example: log based recovery

log records	**recovery**
<T_1 start>	
<T_1, A, 1000, 950>	
<T_2 start>	
<T_1, B, 775, 825>	
<T_1 commit>	
<T_2, C, 700, 600>	

$BL_B = 825$ $BL_A = 1000$ $BL_C = 700$

Transactions

log-based recovery: recovery strategy

example: log based recovery

log records	**recovery**
<T_1 start>	
<T_1, A, 1000, 950>	
<T_2 start>	
<T_1, B, 775, 825>	
<T_1 commit>	
→ <T_2, C, 700, 600>	

$BL_B = 825$ $BL_A = 1000$ $BL_C = 700$

Transactions

log-based recovery: recovery strategy

example: log based recovery

log records	**recovery**		
<T_1 start>			
<T_1, A, 1000, 950>			
<T_2 start>			
<T_1, B, 775, 825>			
<T_1 commit>			
→ <T_2, C, 700, 600>			
<T_2, C, 700>			$BL_C = 700$
	$BL_B = 825$	$BL_A = 1000$	$BL_C = 700$

Transactions

log-based recovery: recovery strategy

example: log based recovery

log records	**recovery**
<T_1 start>	
<T_1, A, 1000, 950>	
<T_2 start>	
<T_1, B, 775, 825>	
→ <T_1 commit>	
<T_2, C, 700, 600>	
<T_2, C, 700>	$BL_C = 700$

$BL_B = 825$ $BL_A = 1000$ $BL_C = 700$

Transactions

log-based recovery: recovery strategy

example: log based recovery

log records	**recovery**
<T_1 start>	
<T_1, A, 1000, 950>	
<T_2 start>	
→ <T_1, B, 775, 825>	
<T_1 commit>	
<T_2, C, 700, 600>	
<T_2, C, 700>	BL_C = 700

BL_B = 825 BL_A = 1000 BL_C = 700

Transactions

log-based recovery: recovery strategy

example: log based recovery

log records	**recovery**
<T_1 start>	
<T_1, A, 1000, 950>	
→ <T_2 start>	
<T_1, B, 775, 825>	
<T_1 commit>	
<T_2, C, 700, 600>	
<T_2, C, 700>	$BL_C = 700$

$BL_B = 825$ $BL_A = 1000$ $BL_C = 700$

Transactions

log-based recovery: recovery strategy

example: log based recovery

log records	**recovery**
<T_1 start>	
<T_1, A, 1000, 950>	
<T_2 start>	
<T_1, B, 775, 825>	
<T_1 commit>	
<T_2, C, 700, 600>	
<T_2, C, 700>	$BL_C = 700$
<T_2 abort>	

$BL_B = 825$ $BL_A = 1000$ $BL_C = 700$

Transactions

log-based recovery: recovery strategy

example: log based recovery

log records	**recovery**
<T_1 start>	
<T_1, A, 1000, 950>	
<T_2 start>	
<T_1, B, 775, 825>	
<T_1 commit>	
<T_2, C, 700, 600>	
<T_2, C, 700>	$BL_C = 700$
<T_2 abort>	

$BL_B = 825$ $BL_A = 1000$ $BL_C = 700$

Transactions

log-based recovery: recovery strategy

example: log based recovery

log records	**recovery**
<T_1 start>	
<T_1, A, 1000, 950>	
<T_2 start>	
<T_1, B, 775, 825>	
<T_1 commit>	
<T_2, C, 700, 600>	
<T_2, C, 700>	$BL_C = 700$
<T_2 abort>	

$BL_B = 825$ $BL_A = 1000$ $BL_C = 700$

Transactions

log-based recovery: recovery strategy

example: log based recovery

log records	**recovery**		
<T_1 start>			
→ <T_1, A, 1000, 950>		BL_A = 950	
<T_2 start>			
<T_1, B, 775, 825>			
<T_1 commit>			
<T_2, C, 700, 600>			
<T_2, C, 700>			BL_C = 700
<T_2 abort>			
	BL_B = 825	BL_A = 950	BL_C = 700

Transactions

log-based recovery: recovery strategy

example: log based recovery

log records	**recovery**
<T_1 start>	
<T_1, A, 1000, 950>	$BL_A = 950$
<T_2 start>	
<T_1, B, 775, 825>	
<T_1 commit>	
<T_2, C, 700, 600>	
<T_2, C, 700>	$BL_C = 700$
<T_2 abort>	
	$BL_B = 825$ $BL_A = 950$ $BL_C = 700$

Transactions

log-based recovery: recovery strategy

example: log based recovery

log records	**recovery**		
<T_1 start>			
<T_1, A, 1000, 950>		BL_A = 950	
<T_2 start>			
→ <T_1, B, 775, 825>			
<T_1 commit>			
<T_2, C, 700, 600>			
<T_2, C, 700>			BL_C = 700
<T_2 abort>			
	BL_B = 825	BL_A = 950	BL_C = 700

Transactions

log-based recovery: recovery strategy

example: log based recovery

log records	**recovery**		
<T_1 start>			
<T_1, A, 1000, 950>		$BL_A = 950$	
<T_2 start>			
<T_1, B, 775, 825>	$BL_B = 825$		
<T_1 commit>			
<T_2, C, 700, 600>			
<T_2, C, 700>			$BL_C = 700$
<T_2 abort>			
	$BL_B = 825$	$BL_A = 950$	$BL_C = 700$

Transactions

log-based recovery: recovery strategy

example: log based recovery

log records	**recovery**		
<T_1 start>			
<T_1, A, 1000, 950>		BL_A = 950	
<T_2 start>			
<T_1, B, 775, 825>	BL_B = 825		
<T_1 commit>			
<T_2, C, 700, 600>			
<T_2, C, 700>			BL_C = 700
<T_2 abort>			
	BL_B = 825	BL_A = 950	BL_C = 700

Transactions

log-based recovery: recovery strategy

example: log based recovery

log records	**recovery**
<T_1 start>	
<T_1, A, 1000, 950>	$BL_A = 950$
<T_2 start>	
<T_1, B, 775, 825>	$BL_B = 825$
<T_1 commit>	✓
<T_2, C, 700, 600>	
<T_2, C, 700>	$BL_C = 700$
<T_2 abort>	
	$BL_B = 825$ $BL_A = 950$ $BL_C = 700$

Transactions

log-based recovery: checkpoints

checkpoints: making our approach more efficient

going over an entire log file can be ***very*** time consuming:

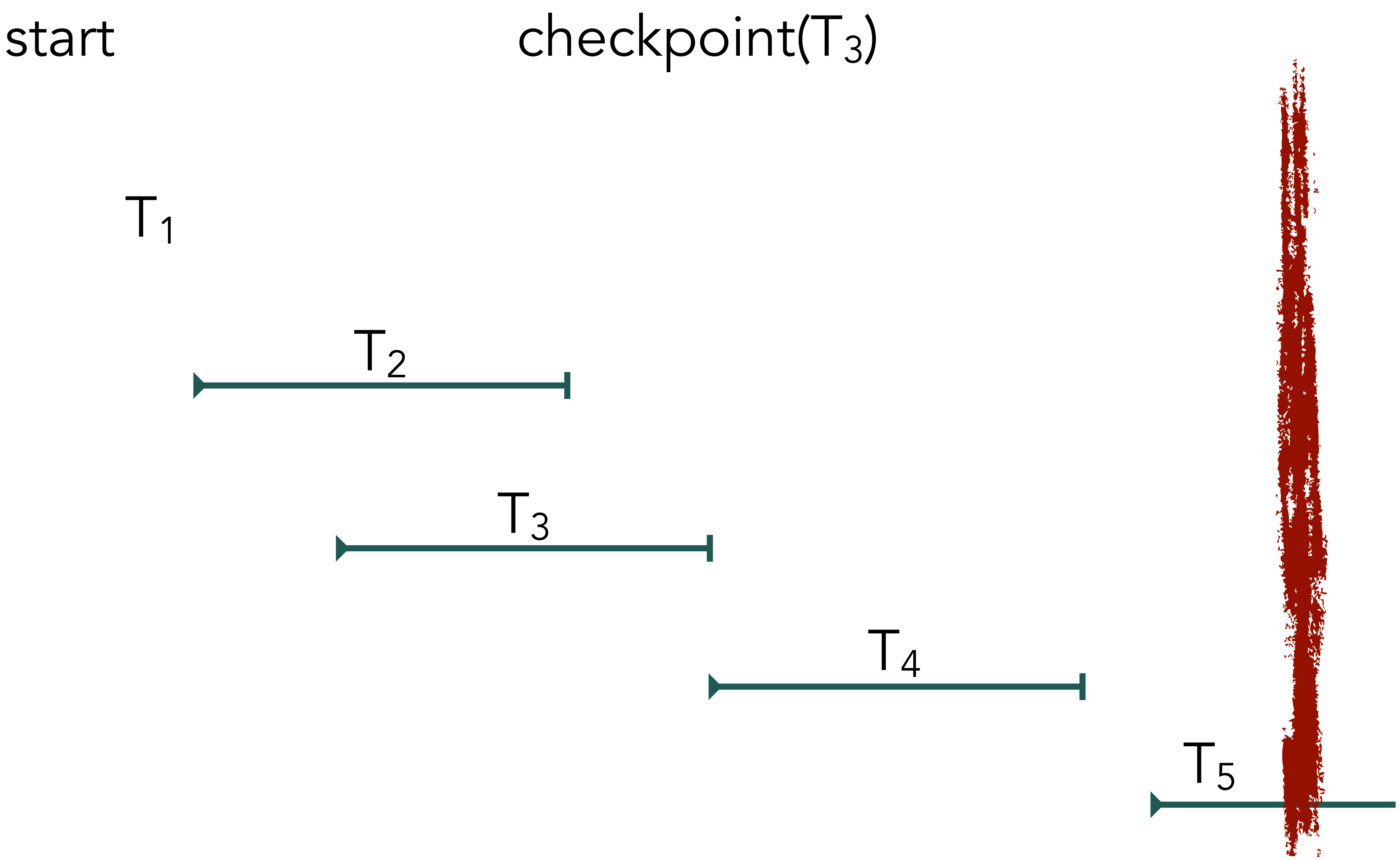
- system might have been running for a long time; and
- we may redo a lot of changes that are already in DB

this can be streamlined by periodic checkpoints

- output all modified buffer blocks to non-volatile blocks
- write a log record <checkpoint L> with L a list of all transactions active at the time of the checkpoint

Transactions

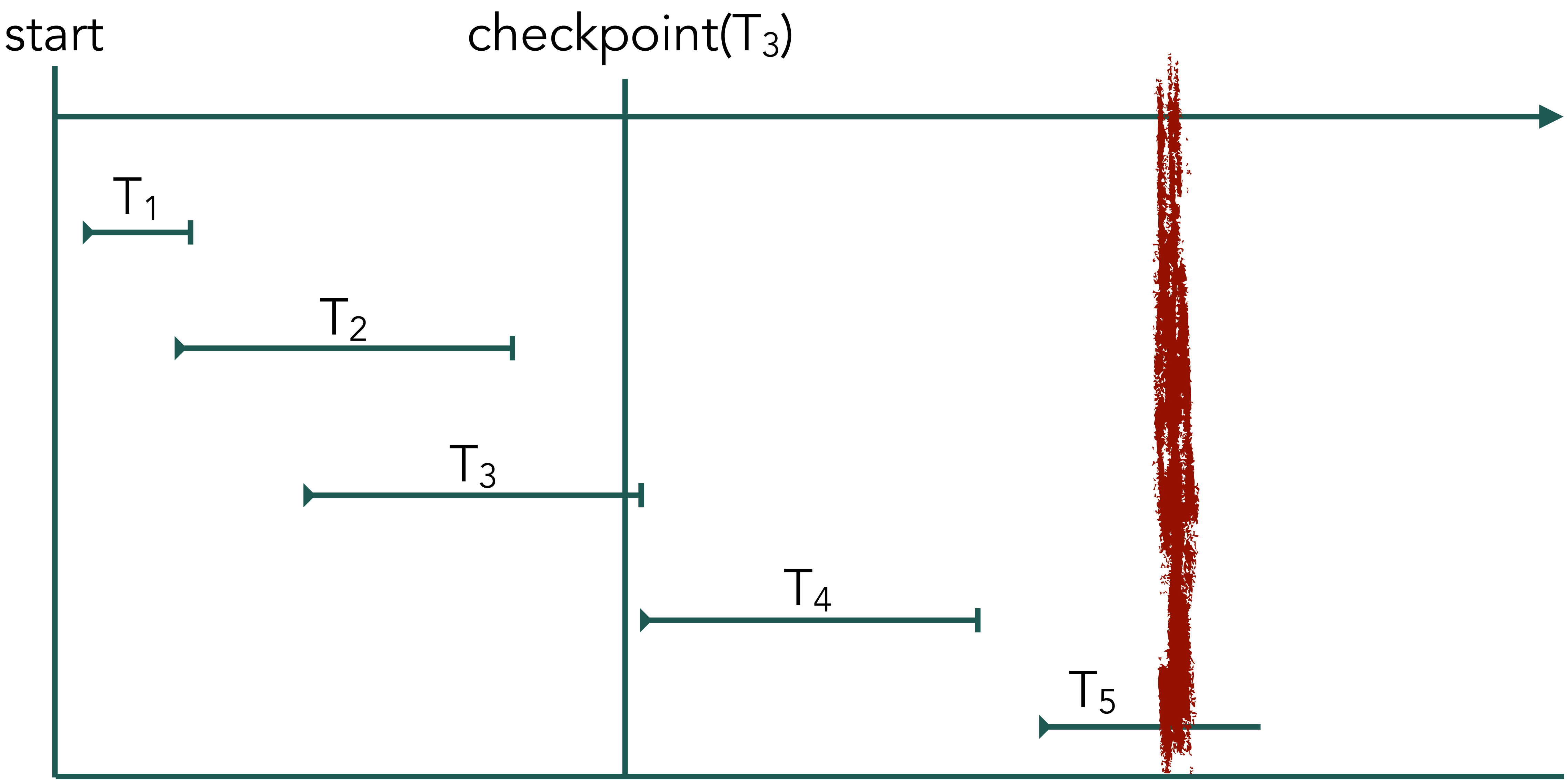
log-based recovery: checkpoints



thanks to checkpoint, we don't need to worry about T_1 and T_2

Transactions

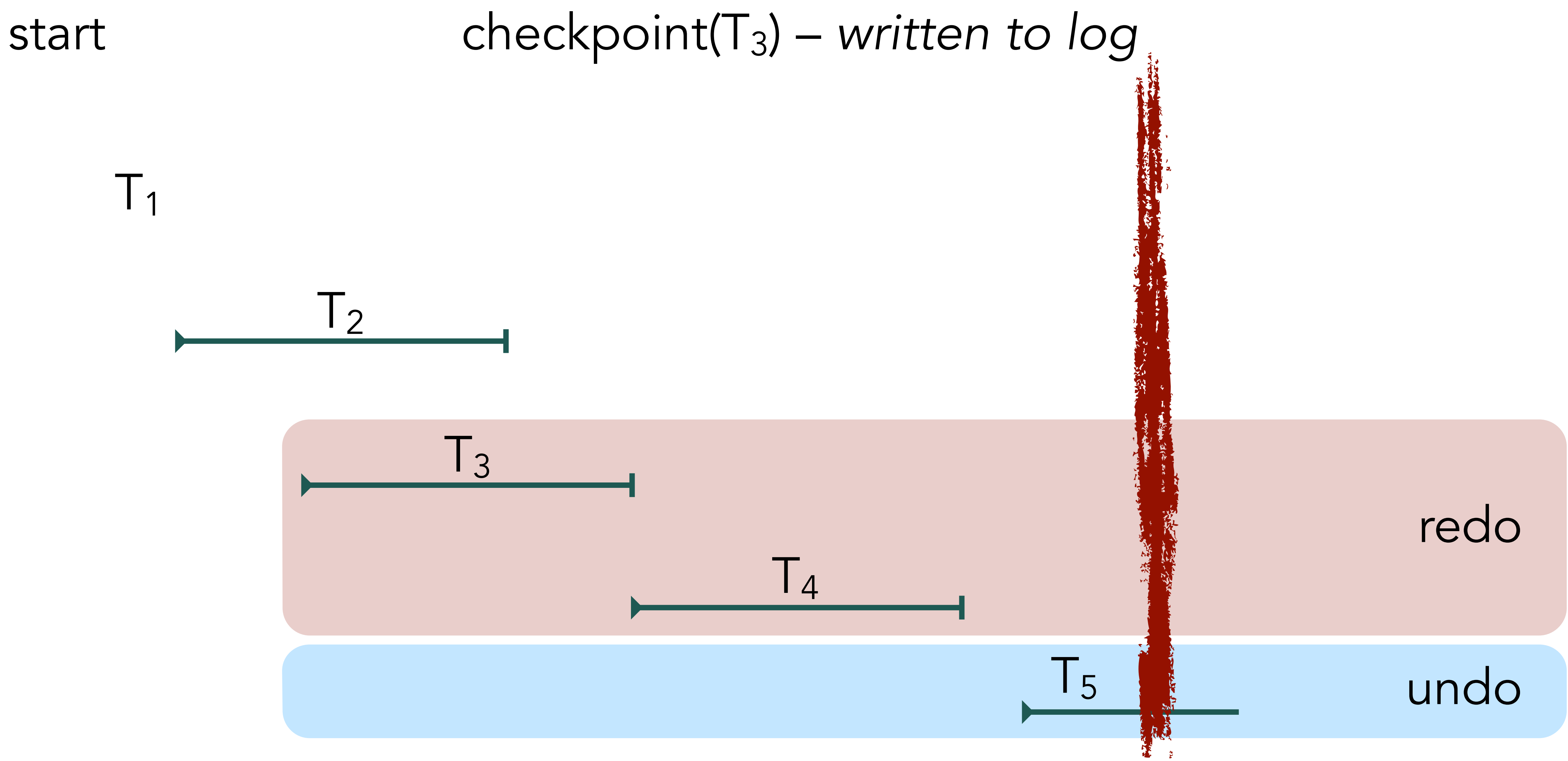
log-based recovery: checkpoints



thanks to checkpoint, we don't need to worry about T_1 and T_2

Transactions

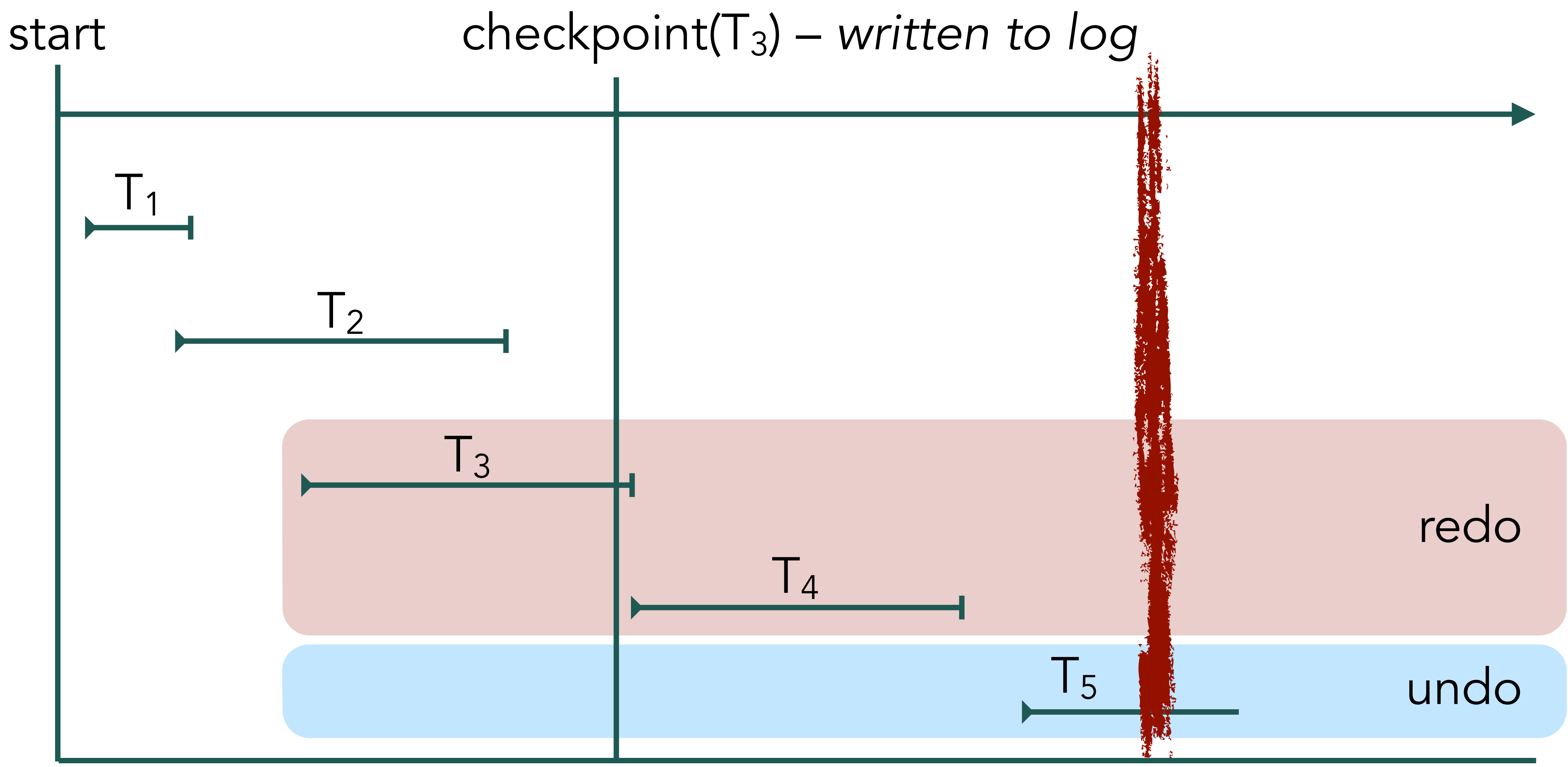
log-based recovery: checkpoints



thanks to checkpoint, we don't need to worry about T_1 and T_2

Transactions

log-based recovery: checkpoints



thanks to checkpoint, we don't need to worry about T_1 and T_2

Transactions

log-based recovery: checkpoints

undo/redo with checkpoints

similar as before, but:

1. undo all until checkpoint is reached
2. then only further undo transactions stated in checkpoint
3. undo until last <T_k start> mentioned in checkpoint is reached
4. from there, redo as normal all T_k in checkpoint and later

Transactions

summary

transactions: summary

- must adhere to ACID properties
- concurrency offers increased performance
- conflict serialisability tells us if schedule is OK
- cascading rollbacks are bad!
- log-based recovery is very common, lightweight
- 1st part is to keep track of changes using log records
- 2nd part is to recover from errors using undo/redo strategy
- checkpoints offer increased performance