6.033 Lecture 17: Isolation 4/9/2014

Sam Madden

Recap : Log Based Recovery

- Key idea: keep a log of actions, then use log to recover state of system

On disk data structures

Cell	Value
A	100
B	50

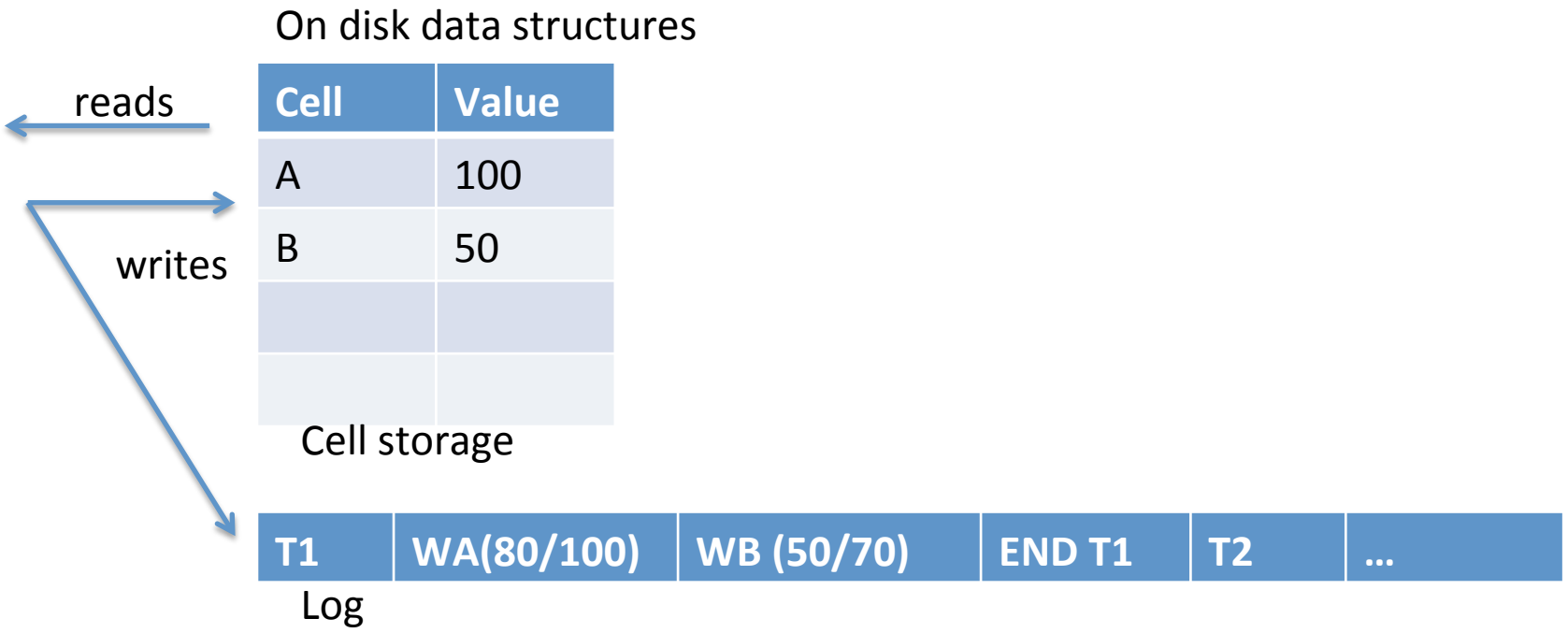
Cell storage

T1	WA(80/100)	WB (50/70)	END T1	T2	…

Log (Before/After) values

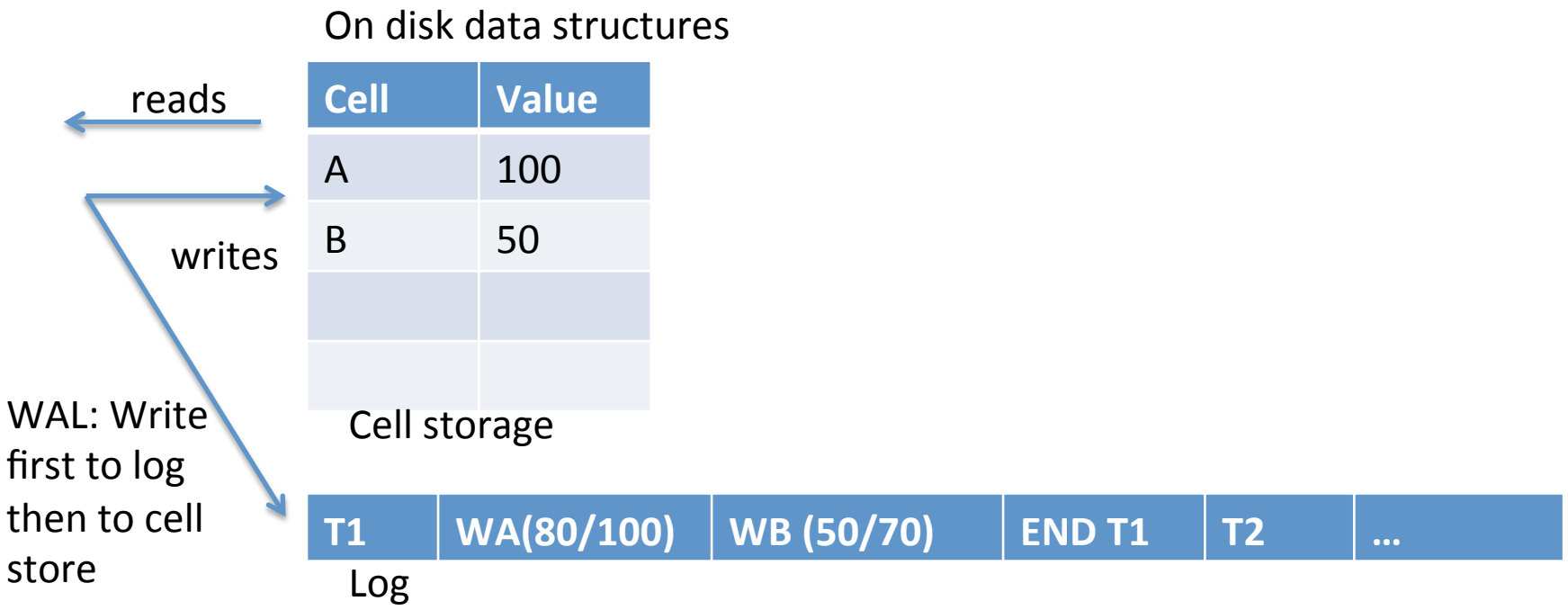
Recap : Log Based Recovery

- Key idea: keep a log of actions, then use log to recover state of system



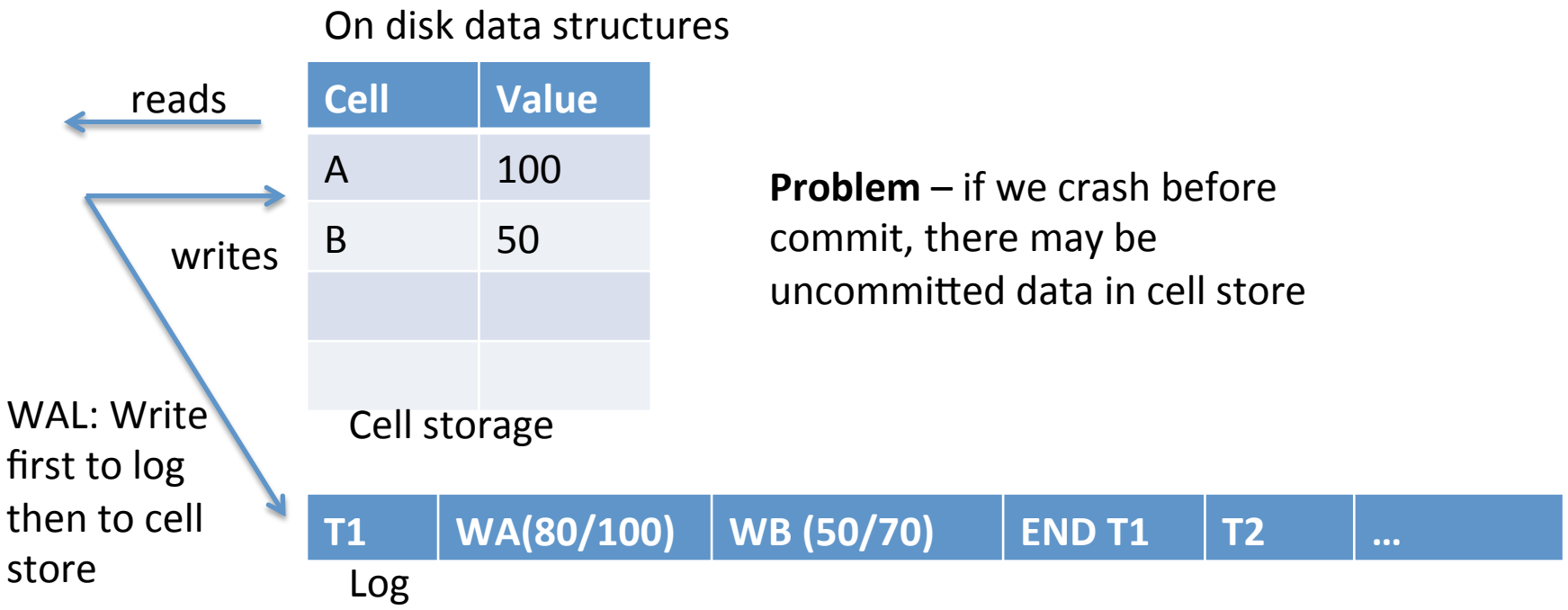
Recap : Log Based Recovery

- Key idea: keep a log of actions, then use log to recover state of system



Recap : Log Based Recovery

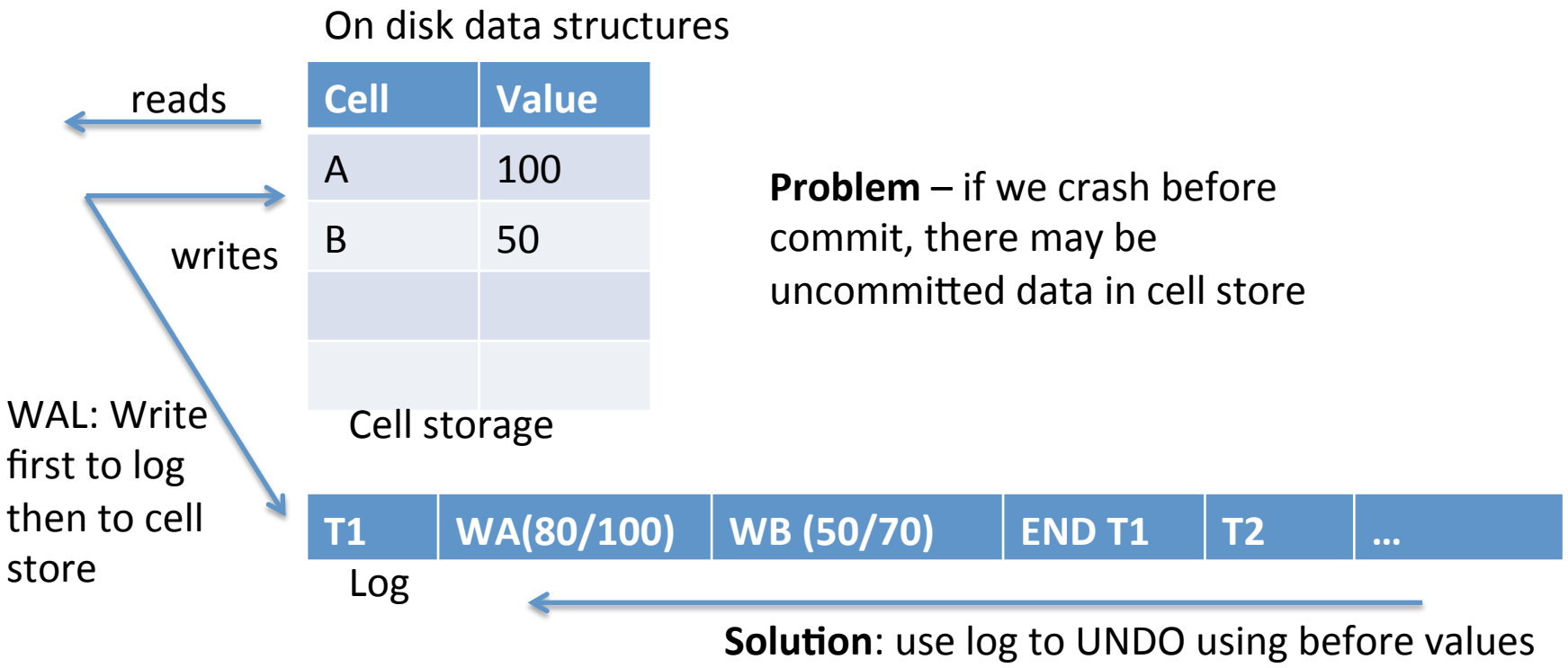
- Key idea: keep a log of actions, then use log to recover state of system



Problem – if we crash before commit, there may be uncommitted data in cell store

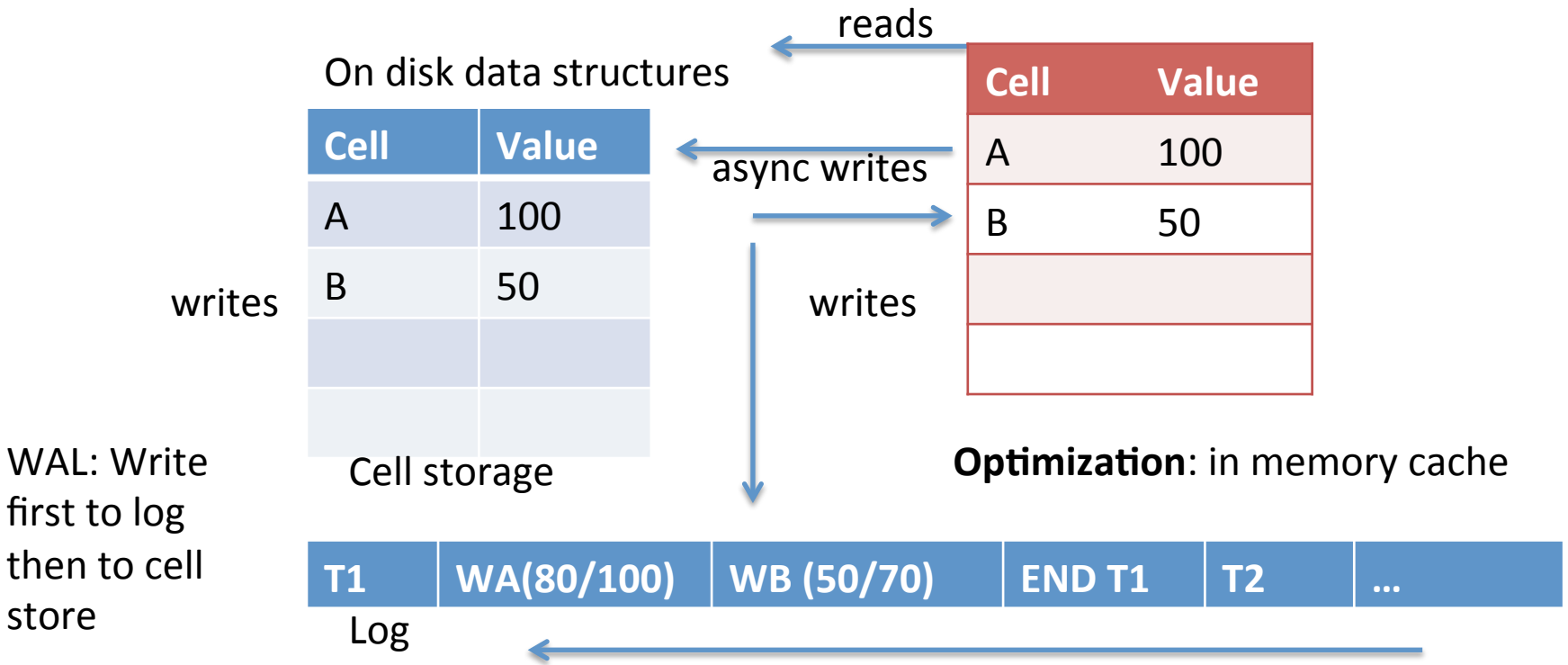
Recap : Log Based Recovery

- Key idea: keep a log of actions, then use log to recover state of system



Recap : Log Based Recovery

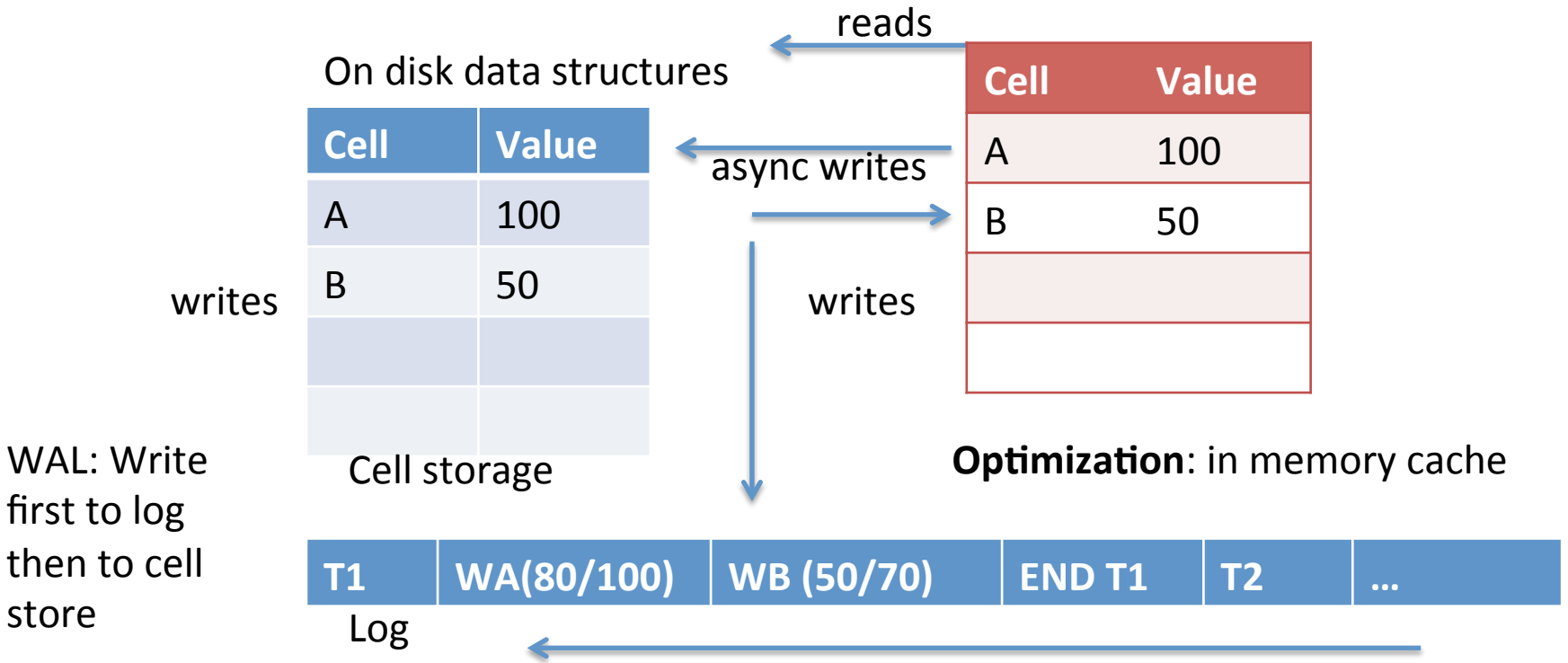
- Key idea: keep a log of actions, then use log to recover state of system



Recap : Log Based Recovery

- Key idea: keep a log of actions, then use log to recover state of system

Problem – crash, some writes from committed transactions may not have been written to disk



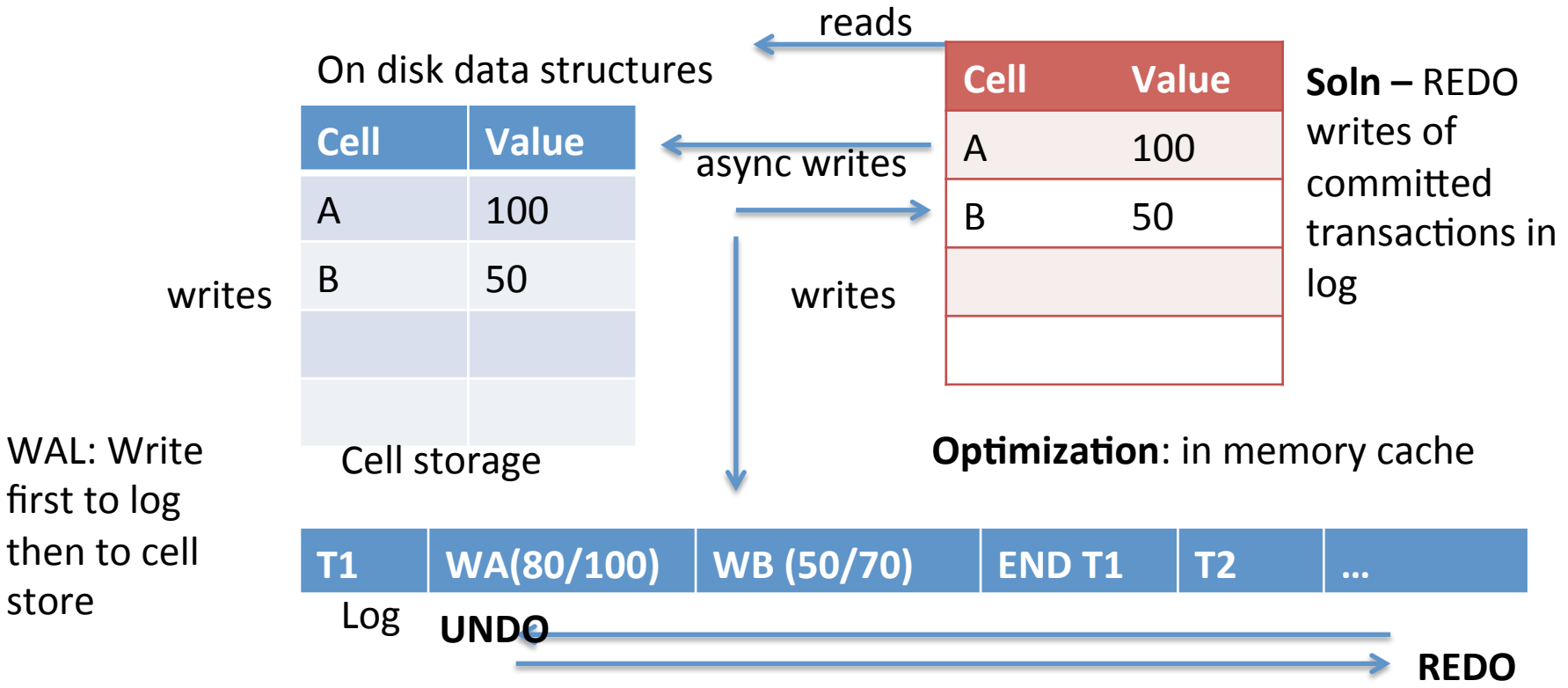
WAL: Write first to log then to cell store

Optimization: in memory cache

Recap : Log Based Recovery

- Key idea: keep a log of actions, then use log to recover state of system

Problem – crash, some writes from committed transactions may not have been written to disk



Recap: Checkpoints

- Problem: log may be very large
- When can we truncate?

- Simple solution:
  - Wait for outstanding transactions to complete
  - Don't start new transactions until
    - Flush of in memory cell cache is complete
    - Log is truncated

Concurrent Actions

```
xfer(a, b, amt):
    begin
    a = a – amt
    b = b + amt
    commit

interest(rate):
    begin
    for each account x:
        x = x * (1+rate)
    commit
```

Conflict Serializability

Given two transactions T1 & T2.

For a read of object o in T1, *conflicts* = {writes of o in T2}

For a write of object o in T1, *conflicts* = {reads or writes of o in T2}

For two transactions T1 & T2, a schedule is **serial equivalent** if:

- Every conflicting read or write in T1 is ordered before the operation it conflicts with in T2,

OR

- Every conflicting read or write in T1 is ordered *after* the operation it conflicts with in T2

Testing for Serializability

```
xfer:                    int:
1 RA [100] (before 6)
                           5 RA [100]
 2 WA [90]  (after 5)
                           6 WA [110]
                           7 RB [50]
                           8 WB [60]
3 RB [60]
4 WB [66]
```

Locking Protocol

```
Read(T, var):
    if var.lock not held by T:
        acquire(T, var.lock)
    return var.value

Write(T, var, newval)
    if var.lock not held by T:
        acquire(T, var.lock)
    var.val = newval //write log record
```

Locking Protocol w/ Release

```
Read(T, var):
    if var.lock not held by T:
        acquire(T, var.lock)
    return var.value

Write(T, var, newval)
    if var.lock not held by T:
        acquire(T, var.lock)
    var.val = newval //write log record

Commit(T):
    write commit record for T
    release all locks for T
```

Locking w/ Reader-Writer Locks

```
Read(T, var):
    if var.lock not held by T:
        acquire_reader(T, var.lock)
            # block if any writers
    return var.value

Write(T, var, newval):
    if var.lock not held as writer by T:
        acquire_writer(T, var.lock)
            # block if any readers or writers
    var.value = newval //and write log record
```

Read committed

Table of doctors w/ names and whether on call

T1	T2
begin	
	begin
	select count(*) from doctors where oncall=true
update doctors set oncall=true where name = 'bob’	
commit	
	select count(*) from doctors where oncall=true

- W/ serializable, T1 will wait for T2
- W/ read committed, T2 will release read lock after select, which will allow T1 to run; T2 will see T1’s update (but do we care)?