● Distributed transaction.

```
T1: Transfer                  T2 : Audit

  x = GET(x)                  x = GET(X)
  y = GET(y)                  y = GET(Y)
if x>10{                      print (x+y)
  PUT (x, x-10)
  PUT(Y, y+10)
}
```

● ACID guarantees:

- Atomicity: All parts of txn execute or none (x's balance decreases, y's balance does not increase)
- Consistency: Preserves invariants. (eg. x's balance >0)
- Durability: Txn's effect are not lost (even if servers restart)

●

- Isolation-

```
T1: Transfer                  T2: Audit
                                                  what if
  x = GET(x)                    x = GET(X)        T1 executes
  y = GET(Y)                                      here?
  PUT(x, x-10)                  y = GET(Y)        Prints 210
  PUT(Y, y+10)                  print (x+y)
              → what if
                T2 executes here?
                Prints 190
```

- Serializability

$\boxed{T_1}\boxed{T_2}$ x: 90 y: 110 P 200

or

$\boxed{T_2}\boxed{T_1}$

Strict serializability

If T_2 started after T_1 committed, then.

$\boxed{T_1}\boxed{T_2}$

- T_2 Stale Read Rx100 Ry100 P200

T_2 Rx100 Ry100 Wx90 Wy110

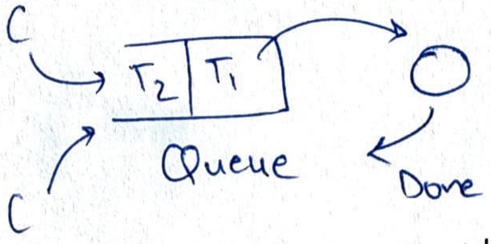
Serializable but not linearizable.

T_2 Rx90 Ry100 P190

T_1 Rx100 Ry100 Wx90 Wy110

Linearizable but not serializable

- Start with single machine



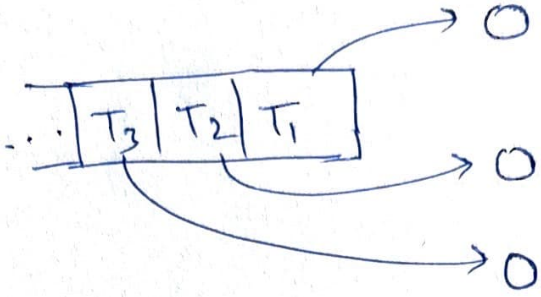
Pull one transaction at a time.

Trivially serializable!

✓ Queue order determines serial order

× Bad perf: uses only one CPU. Could have run independent Txn's in parallel

- Multi Cores



Optimistic
Pessimistic
multi version

Better perf. May not be serializable
(T2 executes in middle of T1)
→ Need concurrency control mechanisms

- Optimistic concurrency control (Redis)

T1:

```
WATCH(X)
WATCH(Y)
MULTI
x = GET(X)
y = GET(Y)
if x>10
PUT(X, x-10)
PUT(Y, y+10)
EXEC
```

ABORT

T2:

```
WATCH(X)
WATCH(Y)
MULTI
x = GET(X)
y = GET(Y)
EXEC
```

or ABORT

- Pessimistic concurrency control

T1

```
rlock(X,Y)
x = GET(X)
y = GET(Y)
if x>10 {
wlock(X,Y)
PUT(X, x+10)
PUT(Y, y-10)
}
release(X,Y)
```

T2

```
rlock(X,Y)
x = GET(X)
y = GET(Y)
release(X,Y)
Print(x+y)
```

- Wound wait deadlock avoidance.

T₁: rlock (X,Y)

wlock (X,Y)

release (X,Y)

T₂: rlock (X,Y)

deadlock!

wlock (X,Y)

release (X,Y)

First to wlock, aborts other transaction.

- optimistic

  - If high conflicts, keep aborting and restarting
  - Need to maintain writes locally that all commit atomically.

Persimistic

  - unnecessary locking if no conflicts
  - Deadlock avoidance is needed

- - Challenge: data does not fit on a single machine (eg. X and Y)

  - sharding

A	B
M-X	Y-L

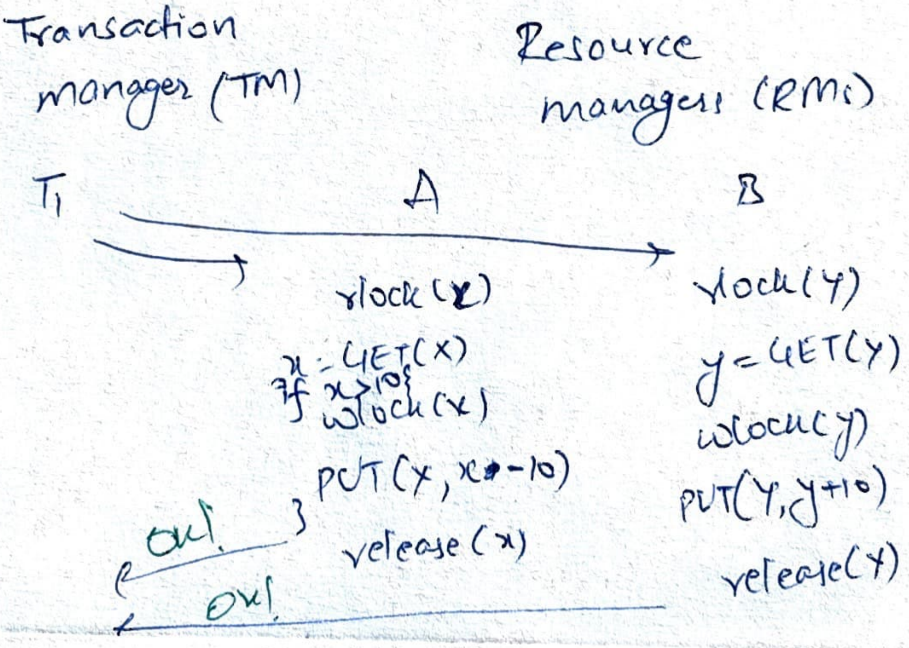
- Might also shard for performance. Transfer from M→N and F→G can happen in parallel on separate machines.

- → Reads, writes, locks need to happen on separate machines

```
A (owns x)                 B (owns y)

rlock (x)                  rlock(Y)
x = GET(x)                 y=GET(Y)
if x>10                    wlock(Y)
    wlock (x)              PUT (Y, y-10)
    PUT (X, x-10)          release (Y)
release (x)
```

- Bad commit protocol



```
A                          B

rlock (x)                  rlock(Y)
x = GET(X)                 y=GET(Y)
if x>10{                   wlock(y)
    wlock(x)               PUT(Y,y+10)
    PUT(X, x-10)           release(Y)
}
release (x)
```

- what can go wrong?
  - Not enough money in X
  - ~~B~~ Y account does not exist
  - A or B crashes before receiving msgs
  - Network fails
  - TM crashes after sending txn to A but before " " B

- Safety?

Atomic commits: Everyone commits

or everyone aborts.

Keep aborting forever?

- Liveness

- If no failures, A, B can commit, then commit
- If failures, reach a conclusion ASAP

R/W transactions can be thought of as 2 phases

```
T1: rlock(X,Y)

    x = get(X)
    assert(x>10)
    y = GET(Y)
    wlock(X,Y)
```

Prepare phase: Read all values. Take all locks. No writes!

```
    PUT(X, x-10)
    PUT(Y, y+10)
    release(X,Y)
```

Commit phase, Write and release locks.

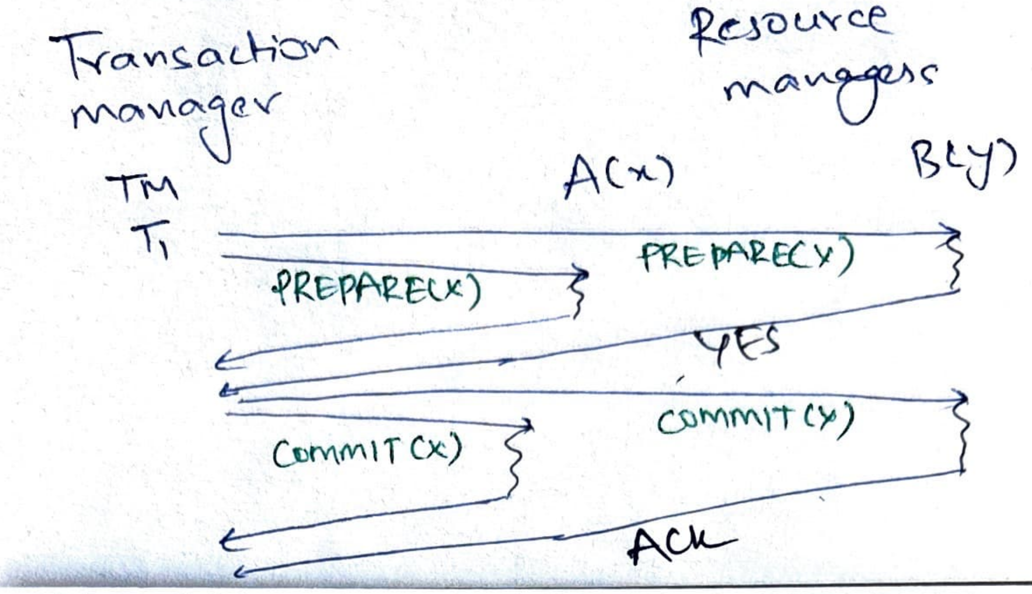
Write both x,y or none i.e must be atomic.

```
PREPARE(X):              COMMIT(X)
    rlock(X)                 PUT(X, x+10)
    x = GET(X)
    assert(x>10)             release(X)
    wlock(X)
    return x.            ABORT(X)

                             release(X)
```

Atomicity

- Two phase Commits



- why does it give atomic commits?

  - TM can send commit only if it has heard Yes from all RMs.

    ↗ All or nothing ↙

  - Ex: If B cannot wlock(y), it replies NO

    ⇒ TM abort transaction.

- B crashes before sending YES to TM.

  - TM timeouts and unilaterally aborts.

  - or n/w lost YES message

  -

- B crashes after sending YES to TM
  - TM sends commit to A.
  - B restarts
    - must remember it was in middle of Txn. $\text{PREPARE}_{T_1}(Y)$ i.e, wlock (y)
    - TM keeps retrying commit (T_1)
    - Is B guaranteed to get wlock (y)?

  [WAL $\text{Prepare}_{T_1}(Y)$ → Yes]

- What if TM restarts before sending prepare to B?
  - Send prepare again.
  - B prepares
  - A should remember it was already prepared and reply YES

- What if TM restarts after prepares?
  - If participant had replied Yes, it is blocked. waiting for commit/abort
  - After restart, TM must commit / abort all pending transactions

TM log

- <Txn ID> <details>
- <commit> <Txn ID>

  or

  <abort> <Txn ID>

why is it ok to not log -

- Sent prepare to A?
  - can just resend prepares
- received yes from A?
  - can unilaterally abort
- Sent commit to A?
  - can just resend commit.

RM/
Participant log

- <Txn ID> <details> <prepared>
- " <committed>

  or <aborted>

Safety -

- No commit unless everyone says Yes
- RM Cannot back out after saying Yes
  across restarts

Liveness

- Not live if TM crashes forever after prepare (or becomes unreachable)