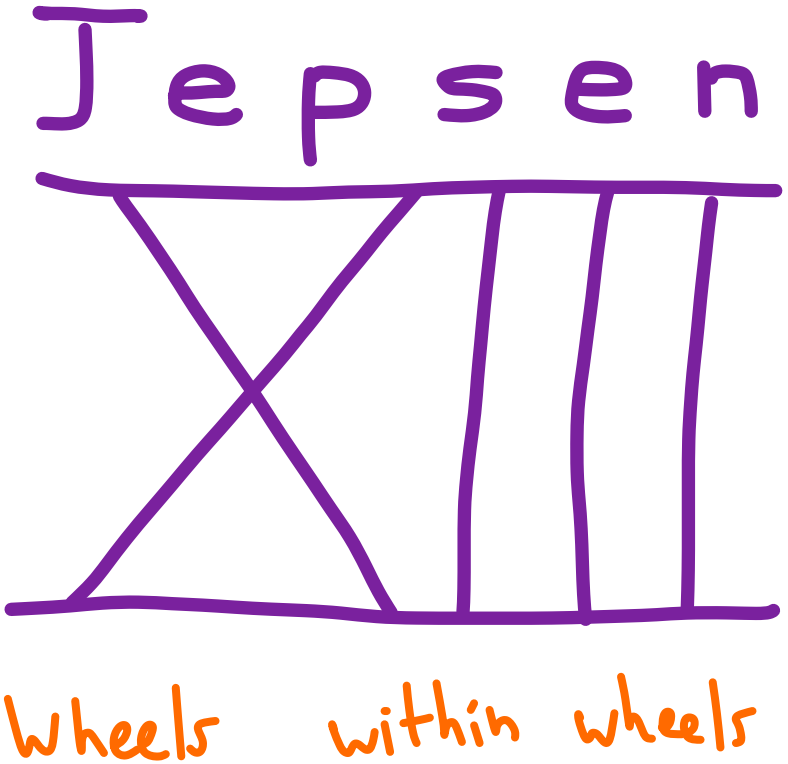

Jepsen
XIII
Wheels within wheels

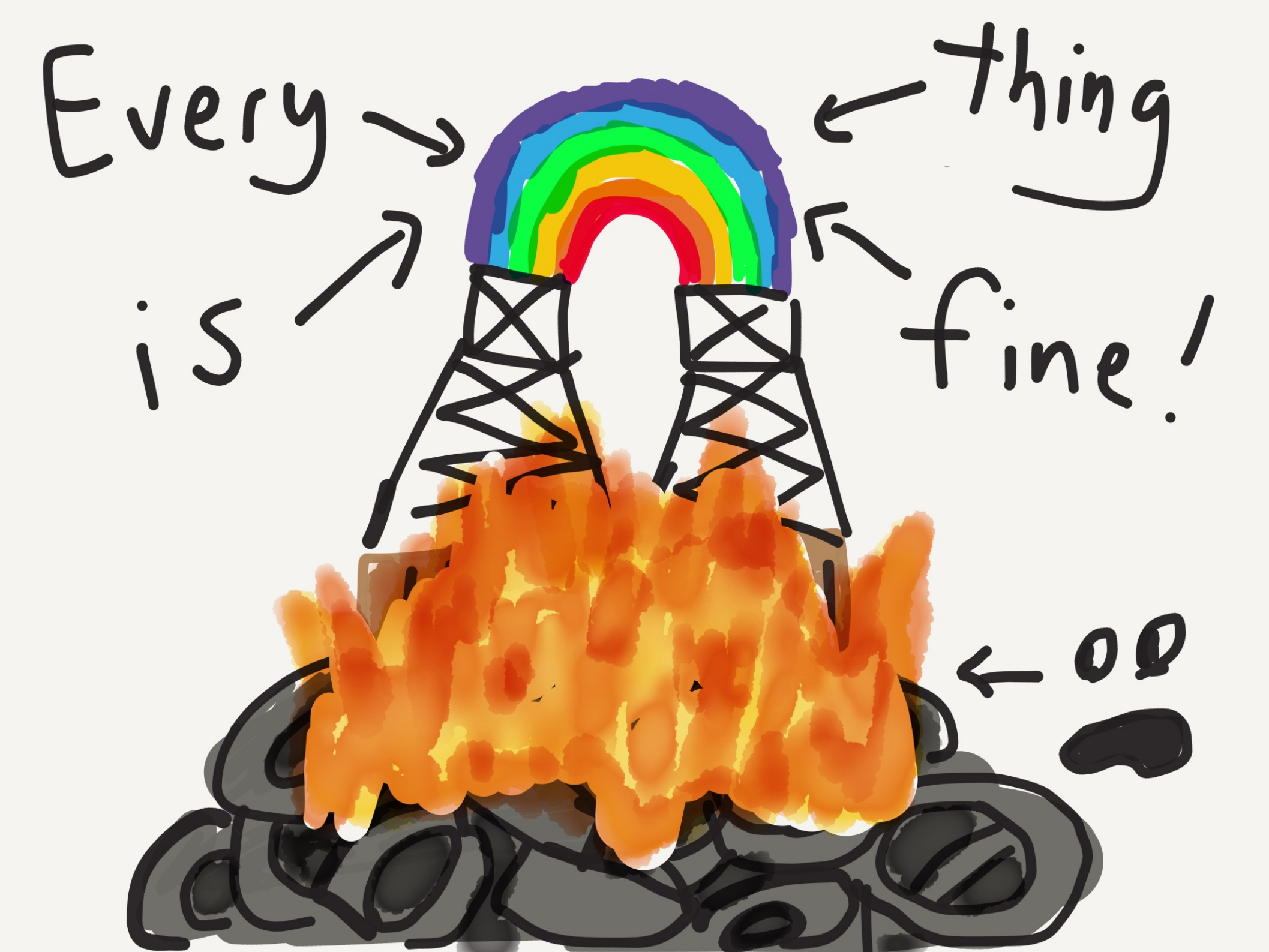
Kyle Kingsbury

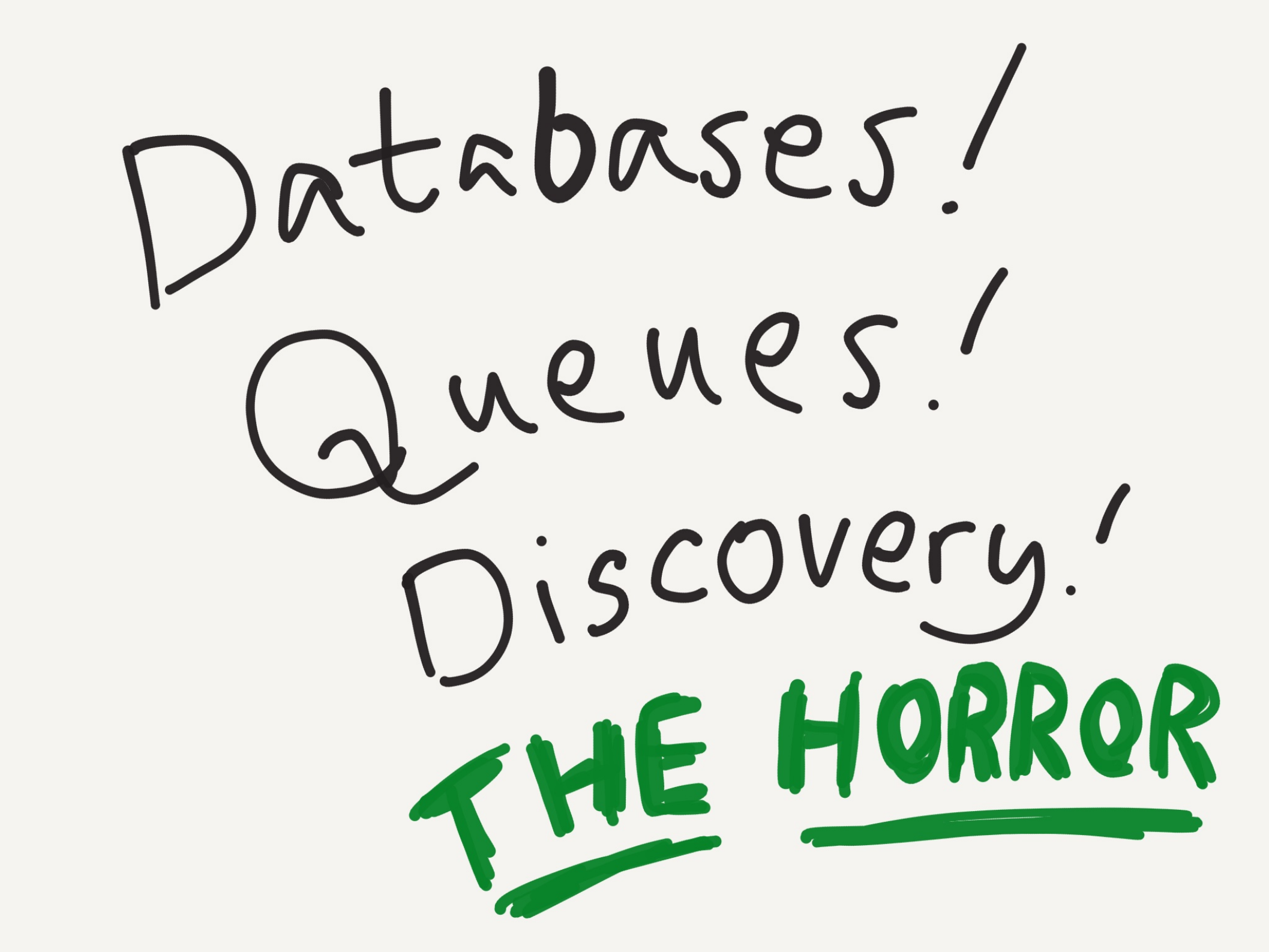
@aphyr

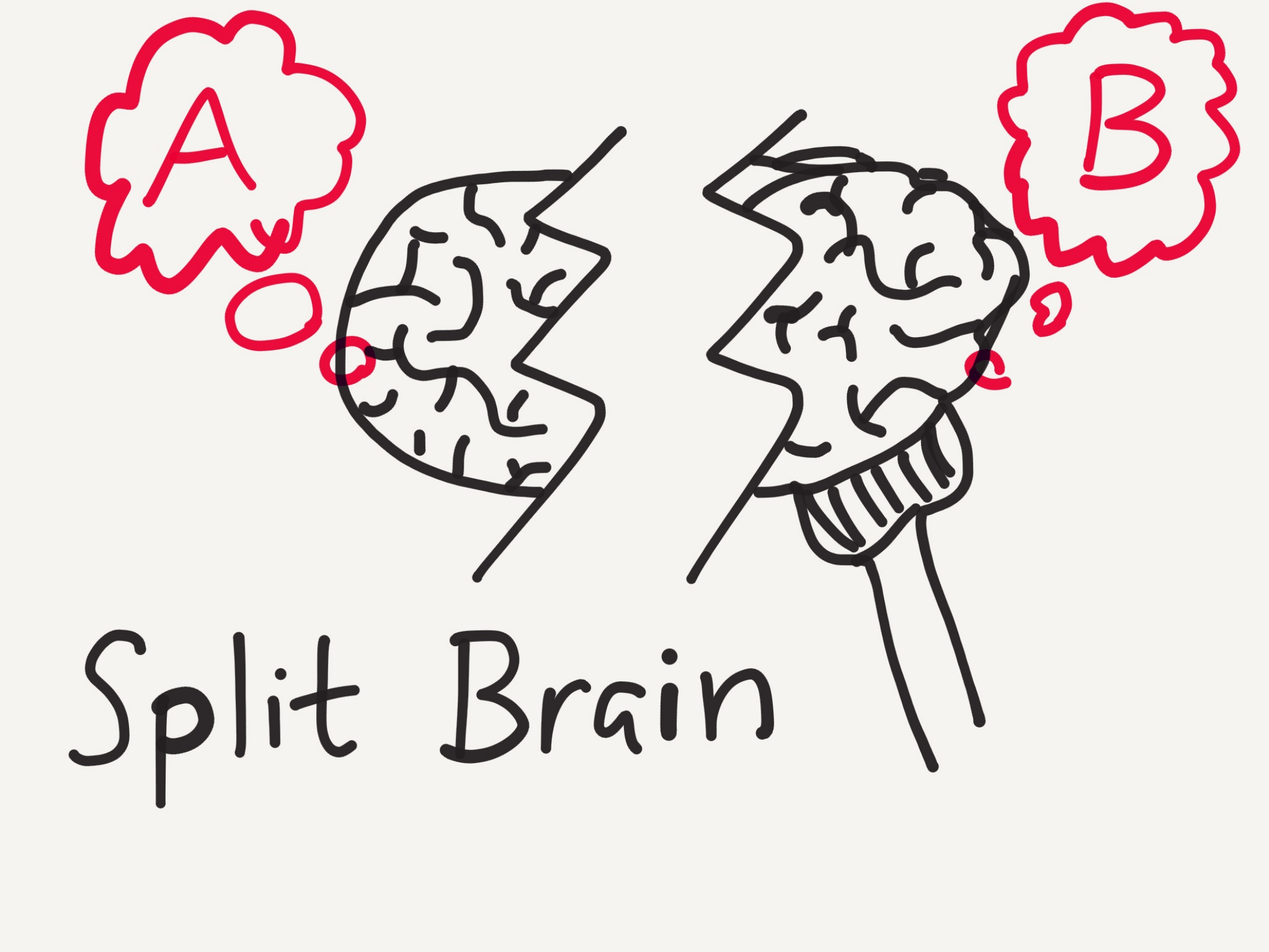
He/him

I Break Databases!

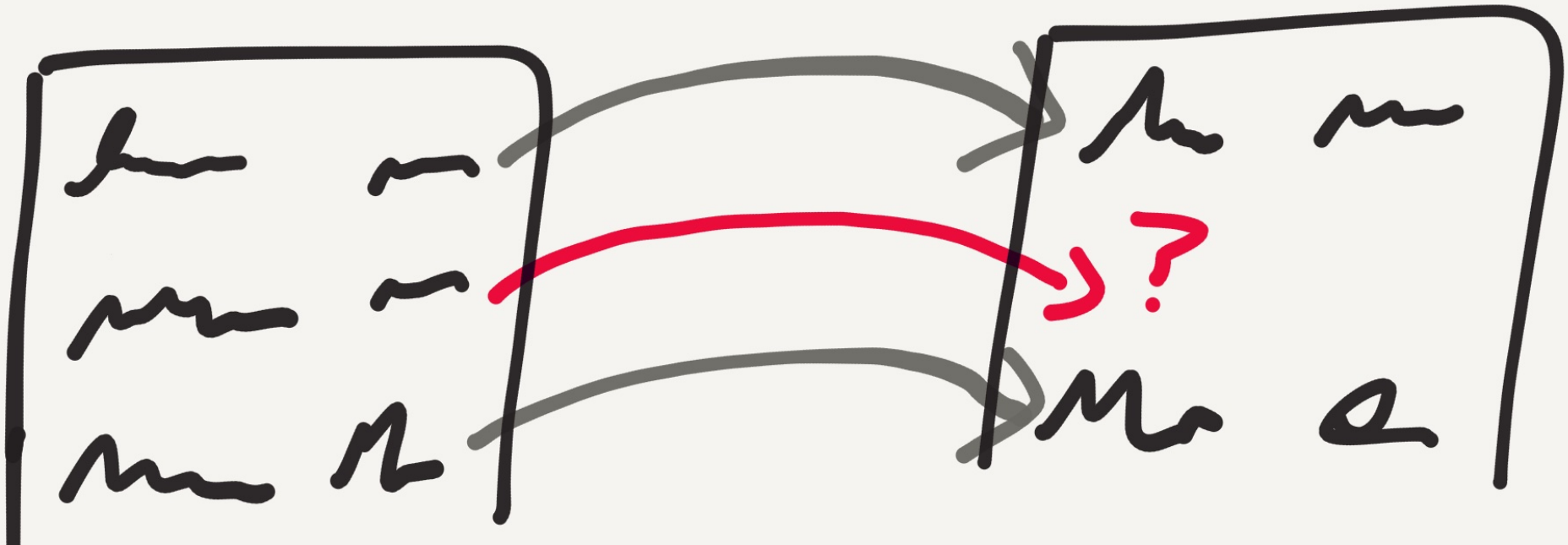

Public API
API code
Ruby
DBs

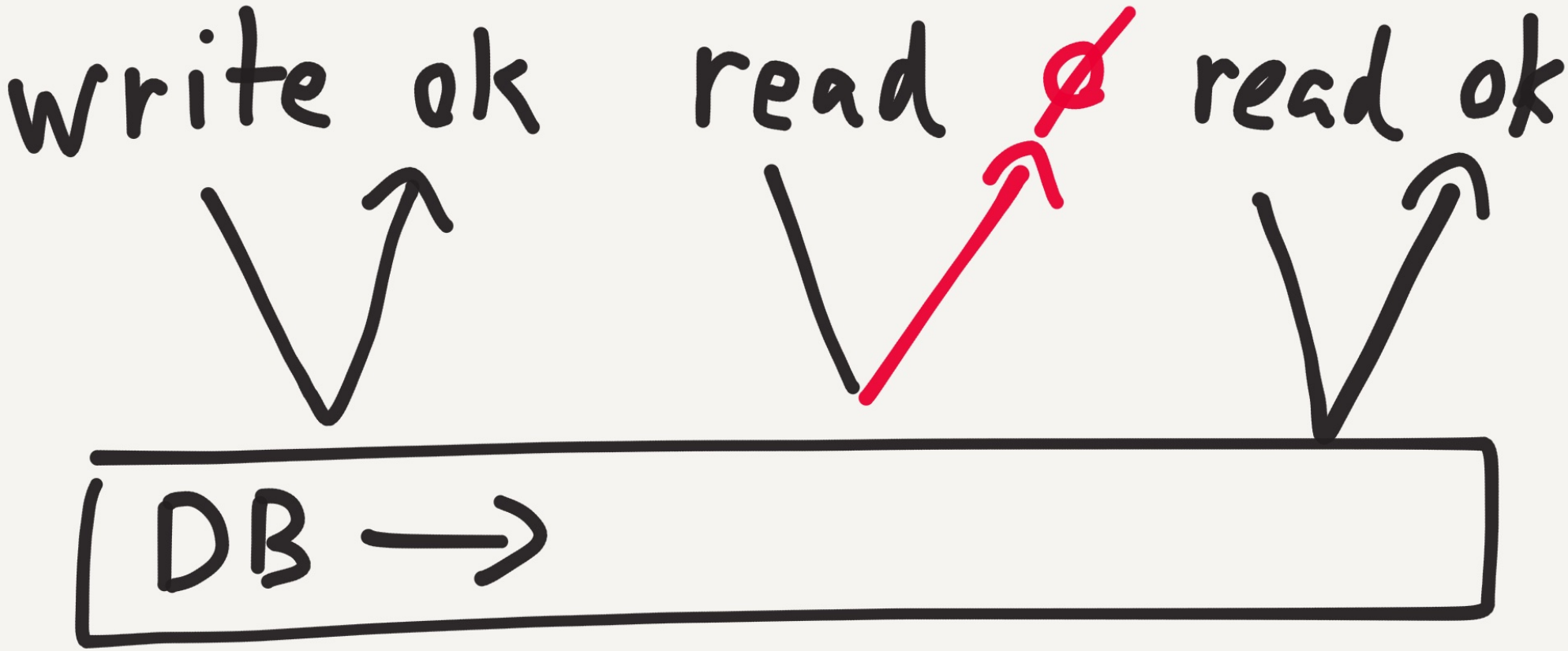

Every thing is fine!
oo


Databases!
Queues!
Discovery!
THE HORROR


A
B
Split Brain

Broken Foreign Keys

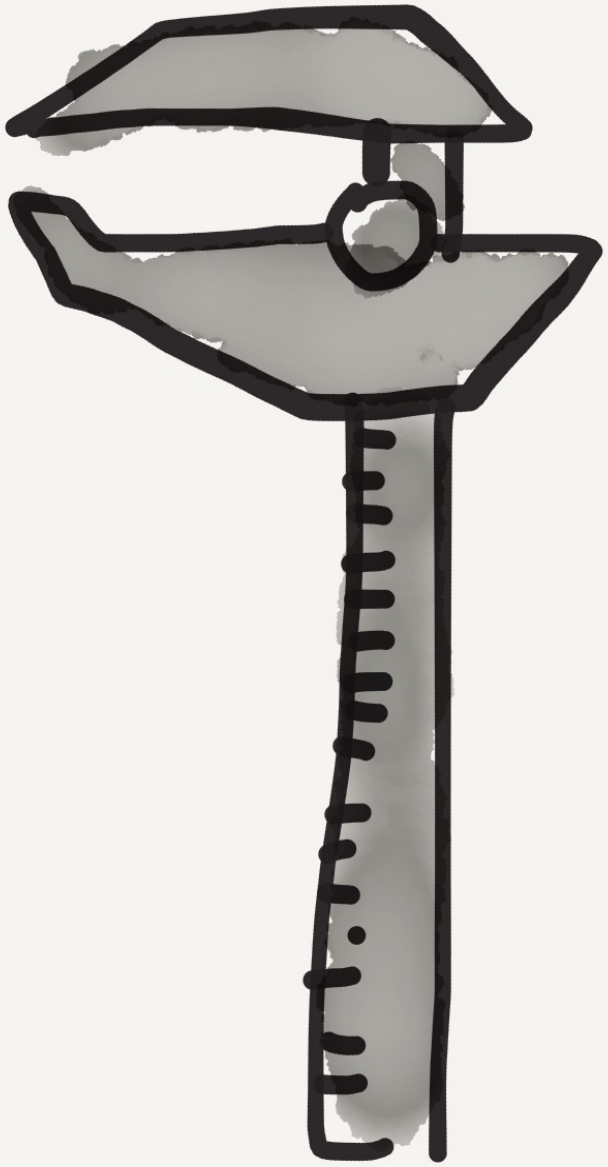




Anomalies

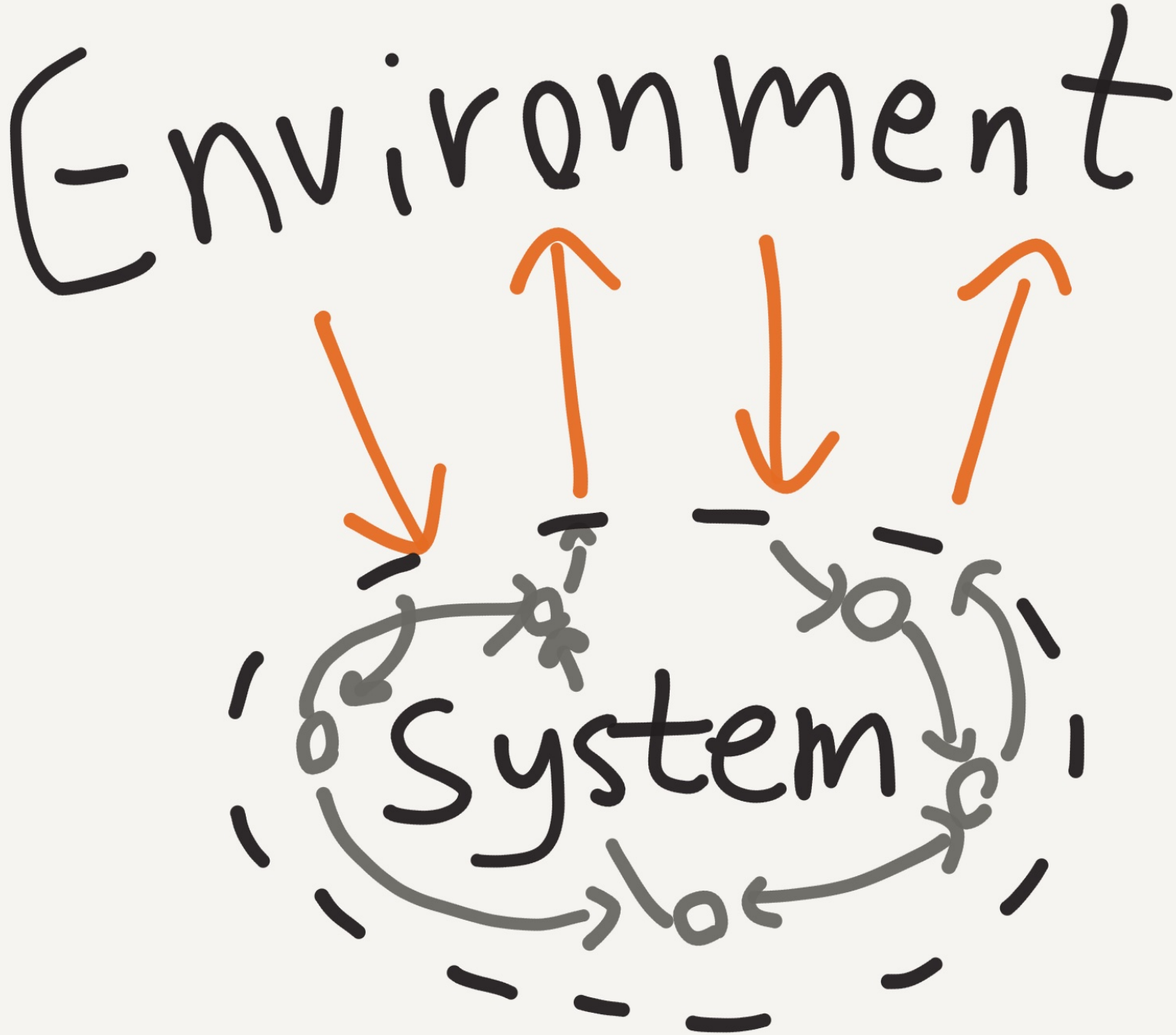
How do you know if a system is safe?

Measure your Systems

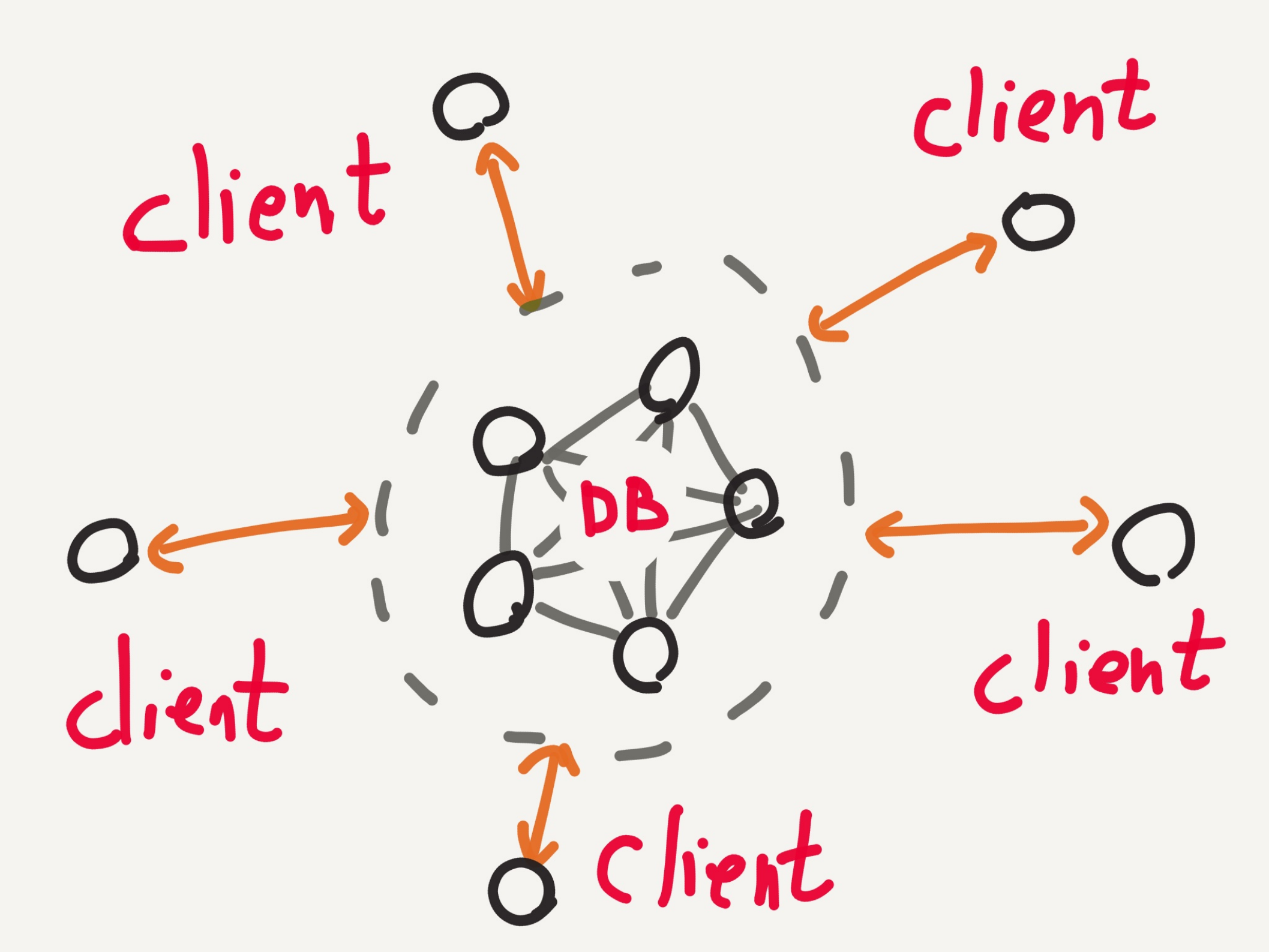


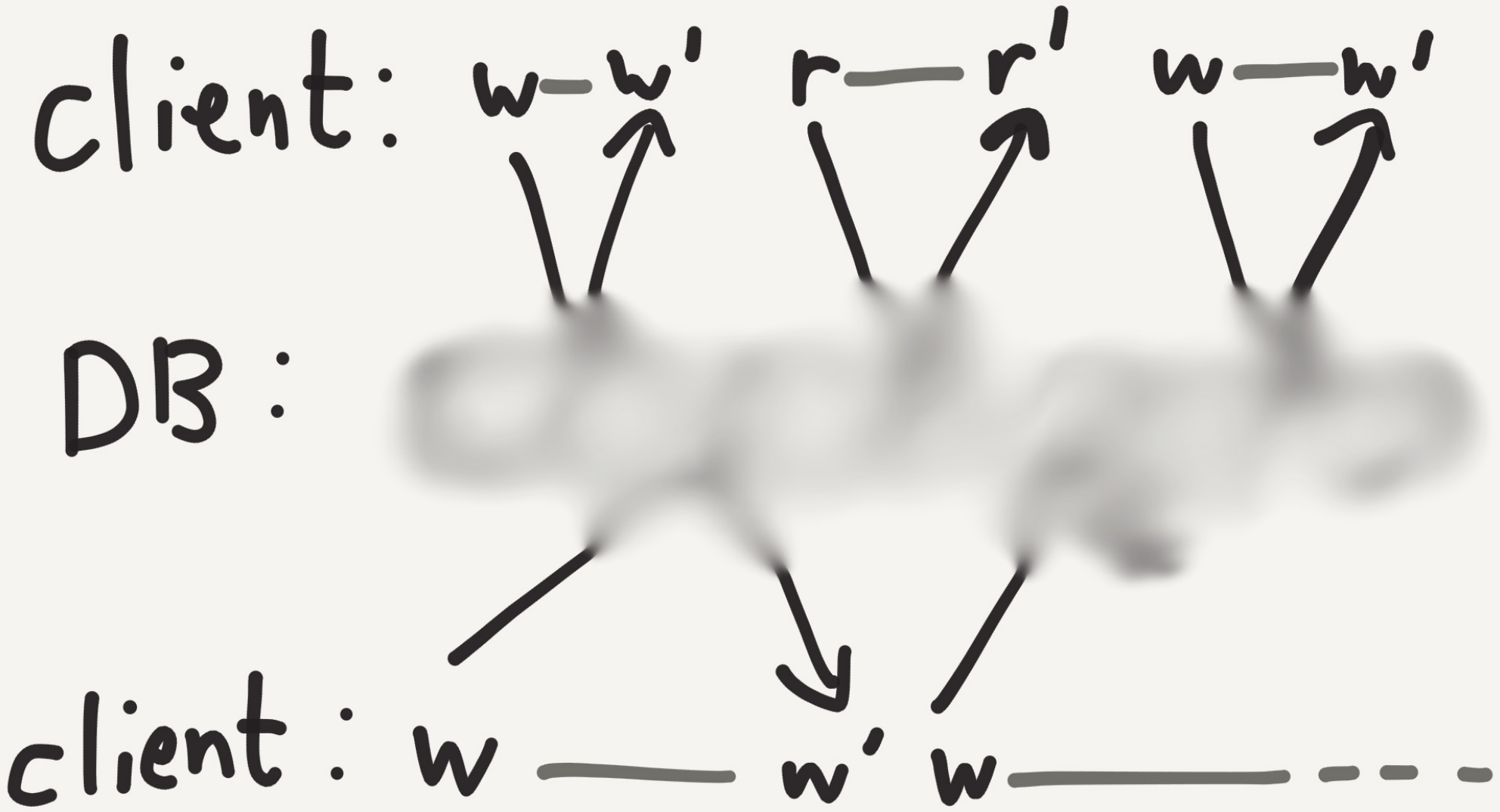
Jepsen

https://jepsen.io


Environment
System


-INVARIANTS-

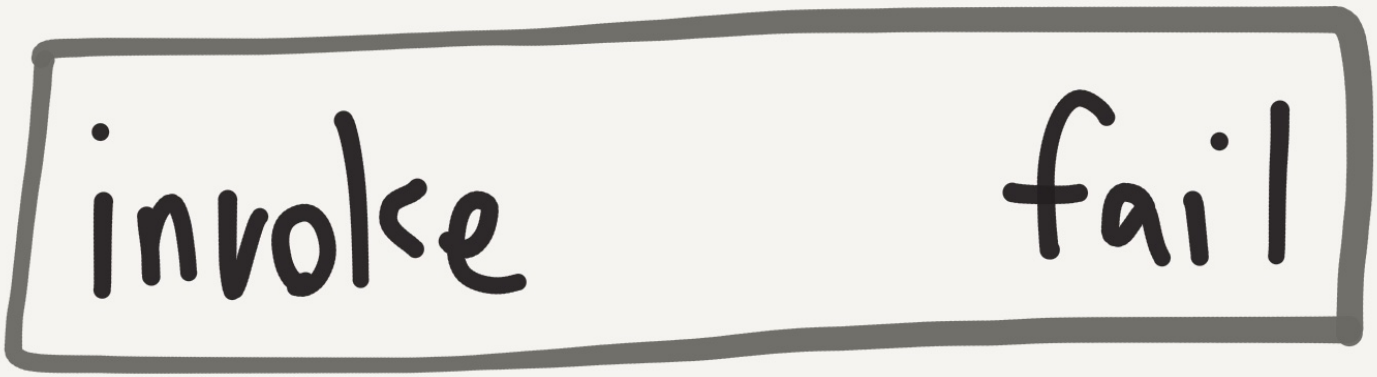

client
client
client
client
client
client
DB


client: w—w' r—r' w—w'
DB:
client: w —— w' w ——

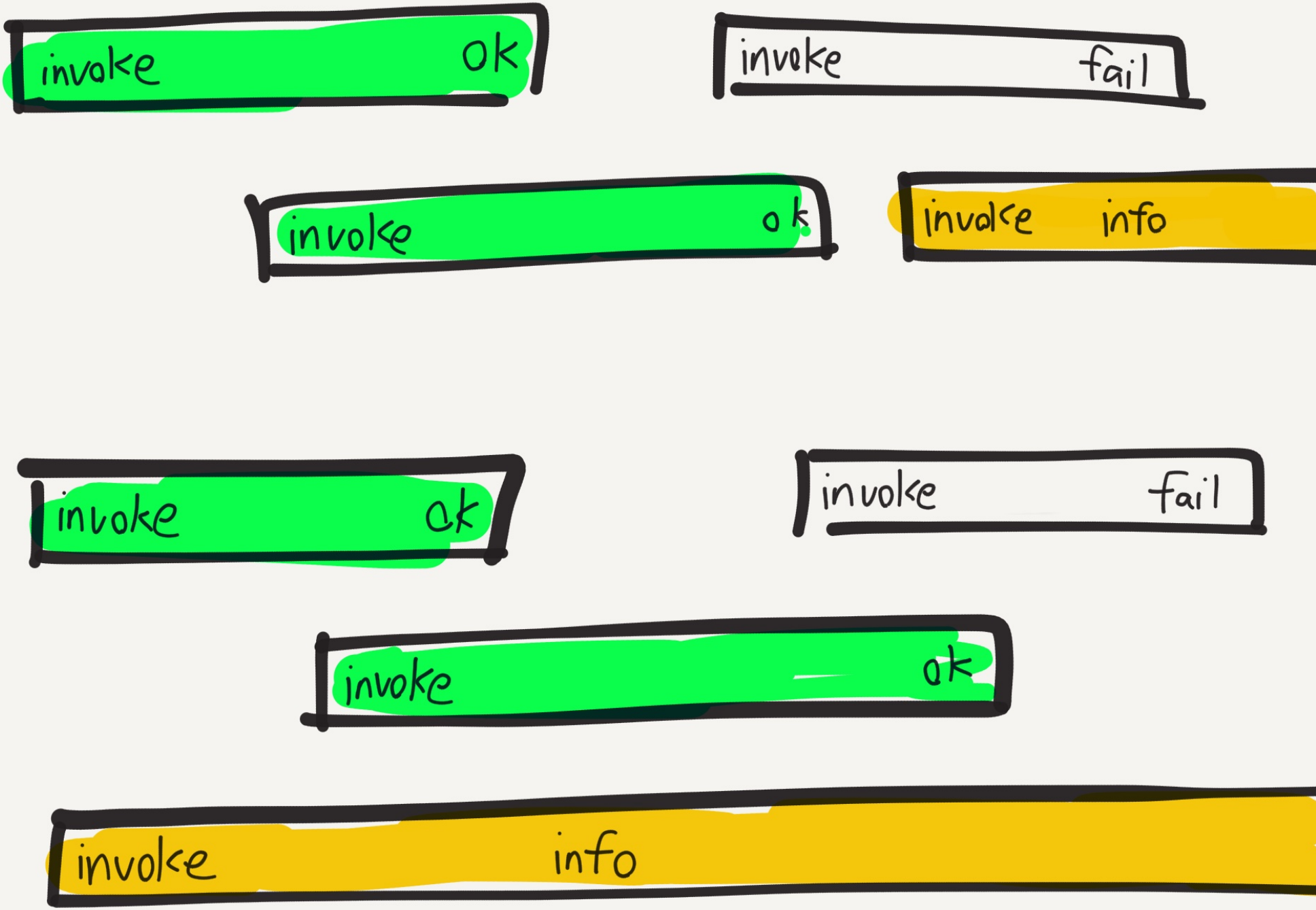
Clients Generate random operations (w) and apply them to the system

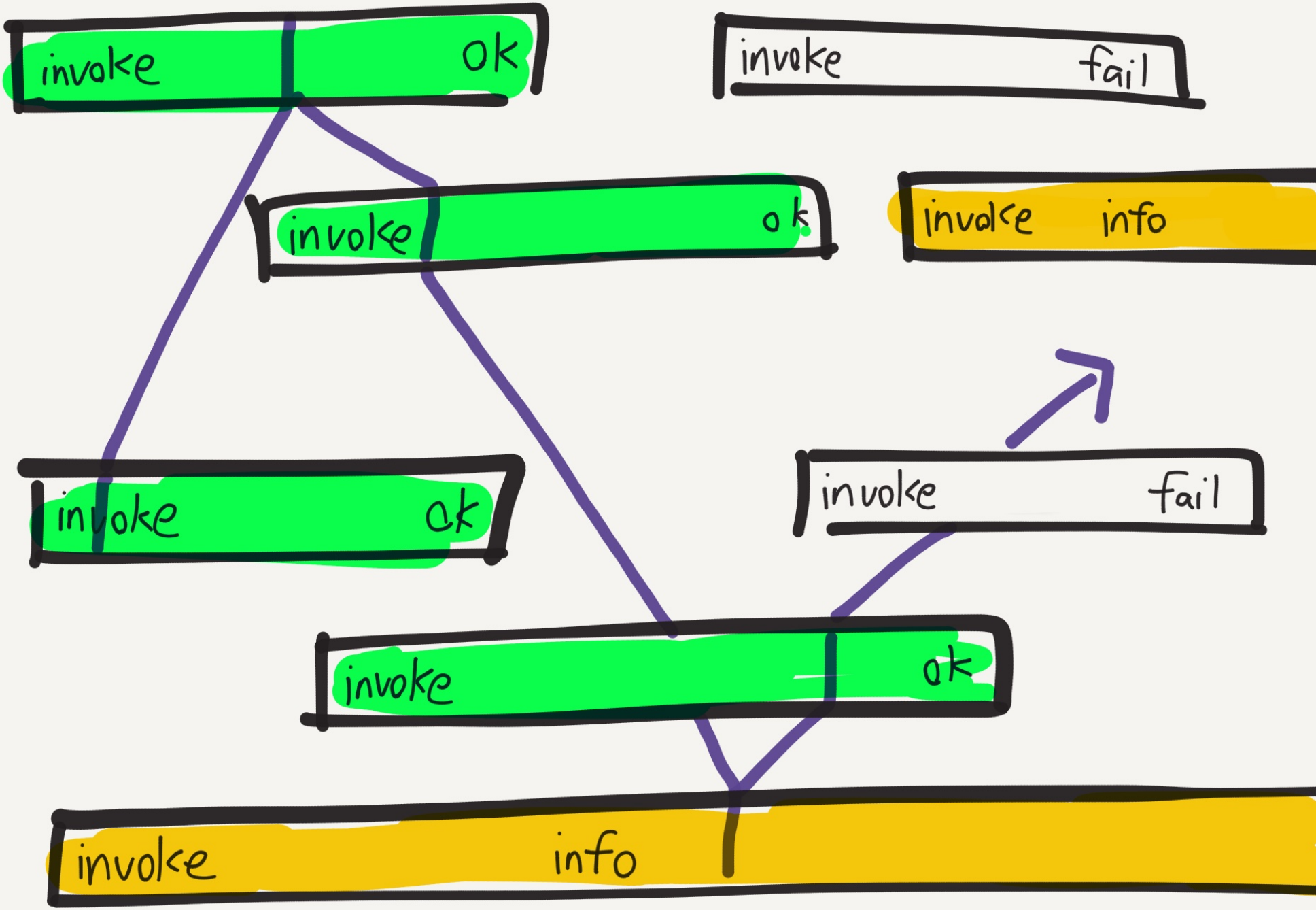
(w')


invoke
ok

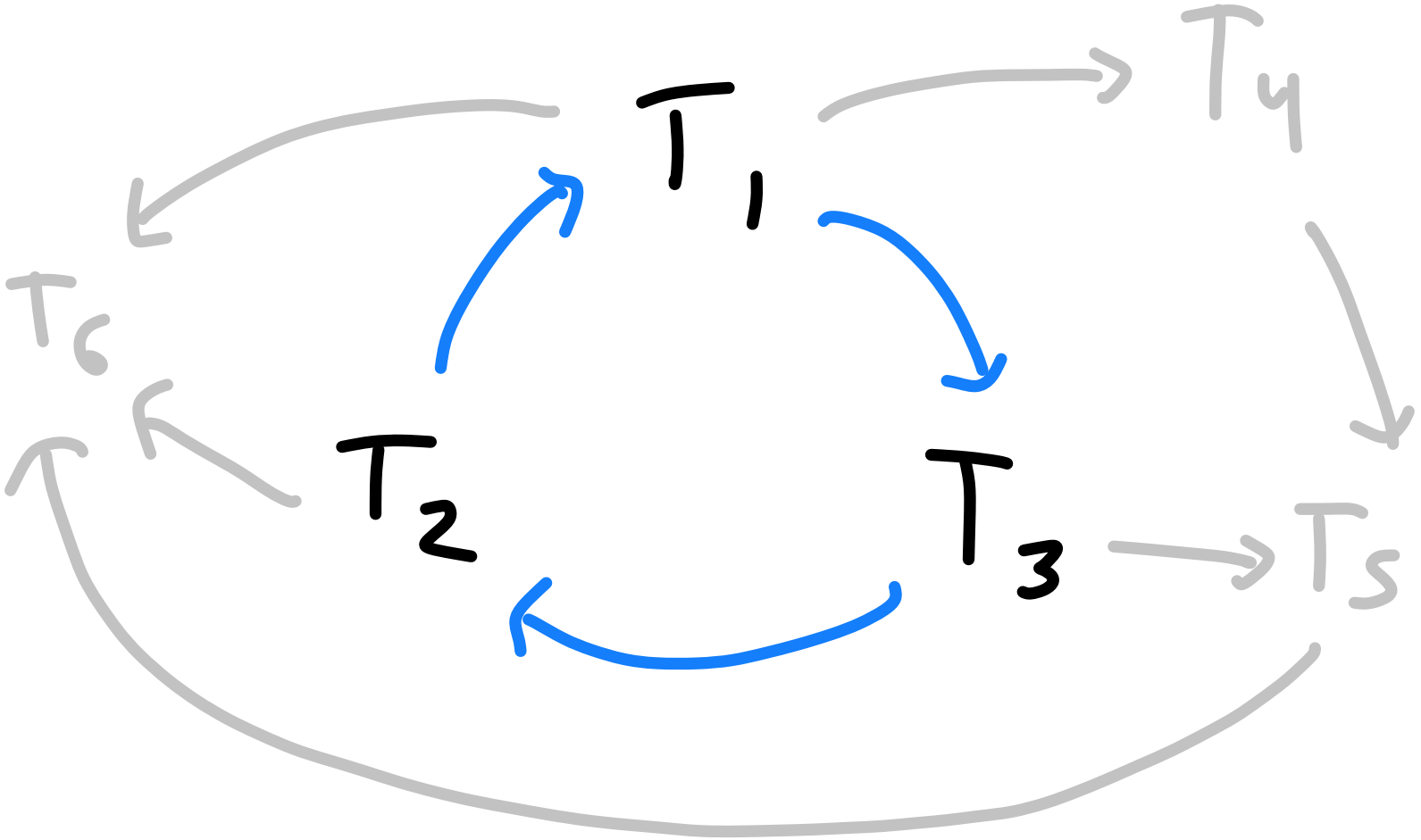

invoke
fail


invoke
info


invoke ok
invoke fail
invoke ok
invoke info
invoke ok
invoke fail
invoke ok
invoke info


invoke
ok
invoke
fail
invoke
ok
invoke
info
invoke
ok
invoke
fail
invoke
ok
invoke
info

Elle: Find Cycles



$T_1 \rightarrow T_2$

ww: $w(v_1) \rightarrow w(v_2)$

wr: $w(v_1) \rightarrow r(v_1)$

rw: $r(v_1) \rightarrow w(v_2)$

$v_1 \ll v_2$

read (x) ⇒ 5

read (x) ⇒ 3

$3 \ll 5 \mid 5 \ll 3$

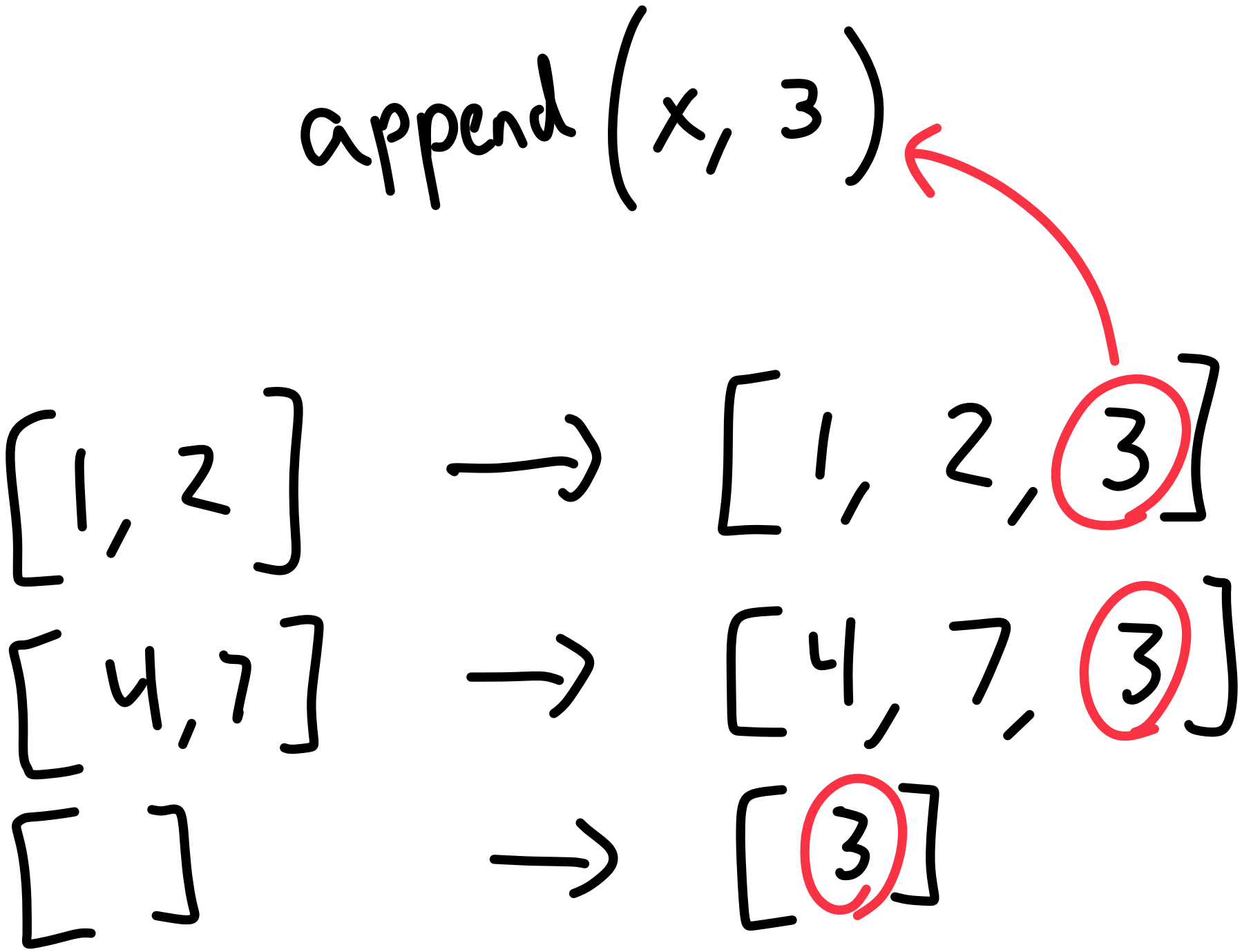
Either could be true!

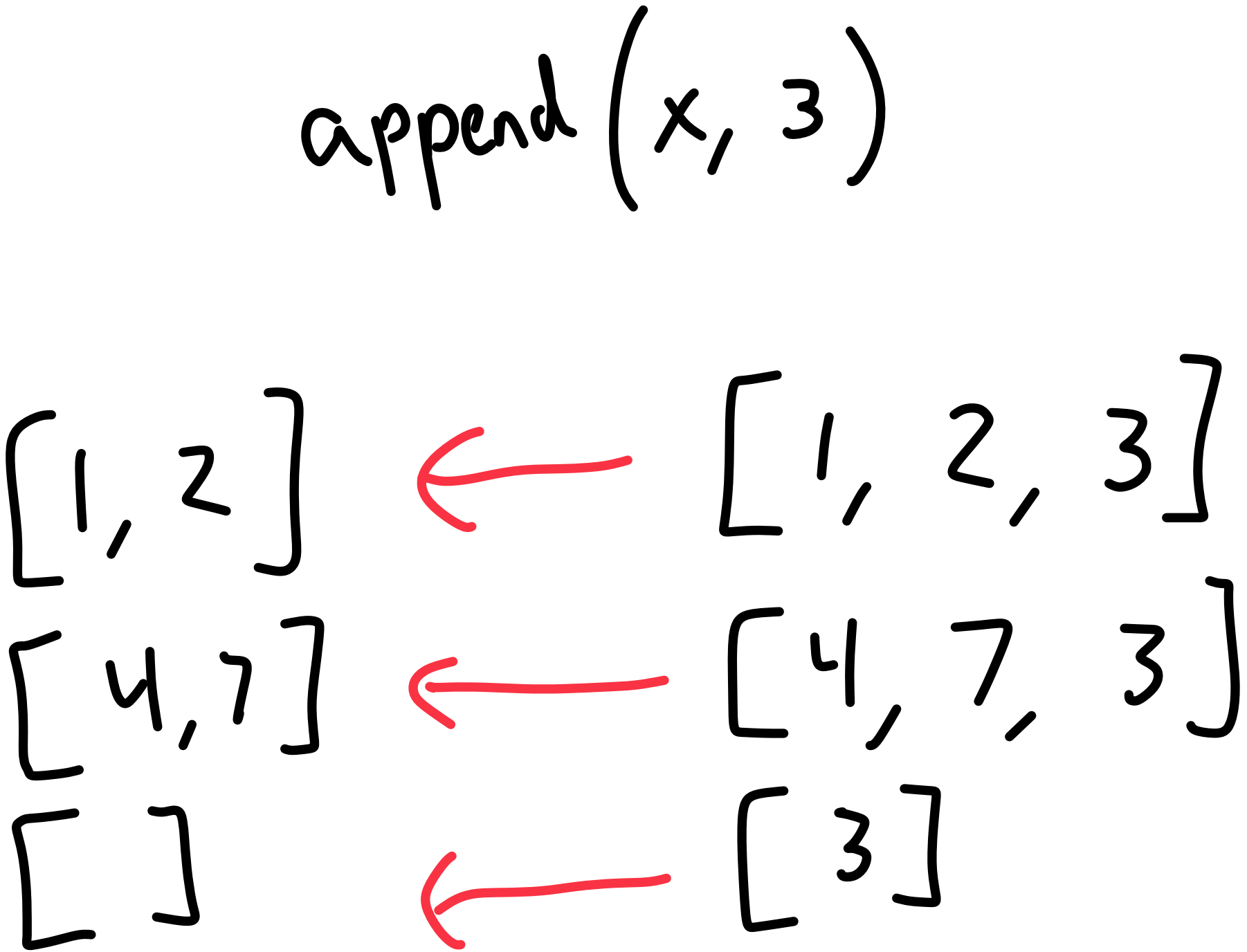
append(x, 3)

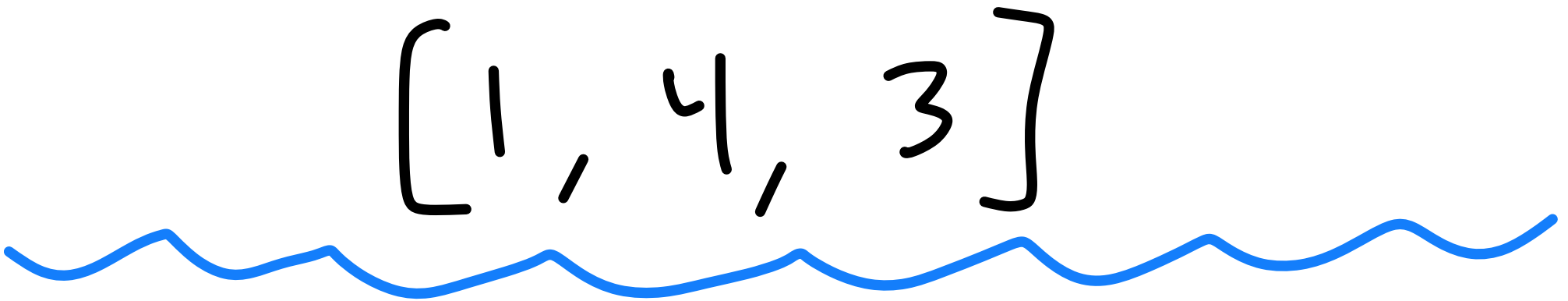
[1, 2] → [1, 2, 3]

[4, 7] → [4, 7, 3]

[ ] → [3]


append(x, 3)
[1, 2] → [1, 2, 3]
[4, 7] → [4, 7, 3]
[ ] → [3]


append(x, 3)
[1, 2]
[1, 2, 3]
[4, 7]
[4, 7, 3]
[ ]
[3]

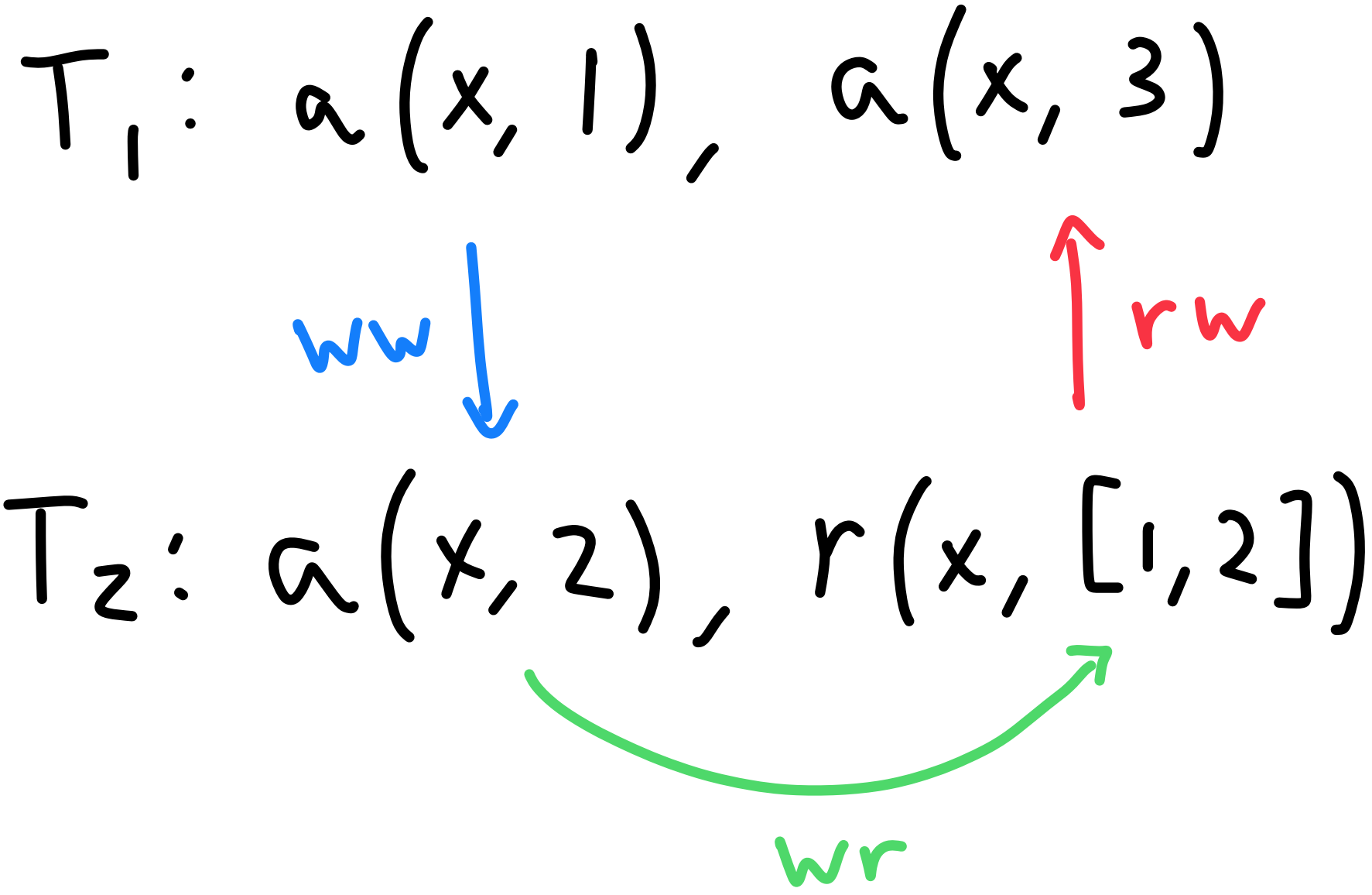

[1, 4, 3]

$v_3 = [1, 4, 3]$

$v_2 = [1, 4]$

$v_1 = [1]$

$v_0 = []$


T1: a(x, 1), a(x, 3)
ww
rw
T2: a(x, 2), r(x, [1,2])
wr

- Write Cycle
- Aborted Read
- Intermediate Read
- Cyclic Information Flow
- Read skew
- Anti-dependency Cycle

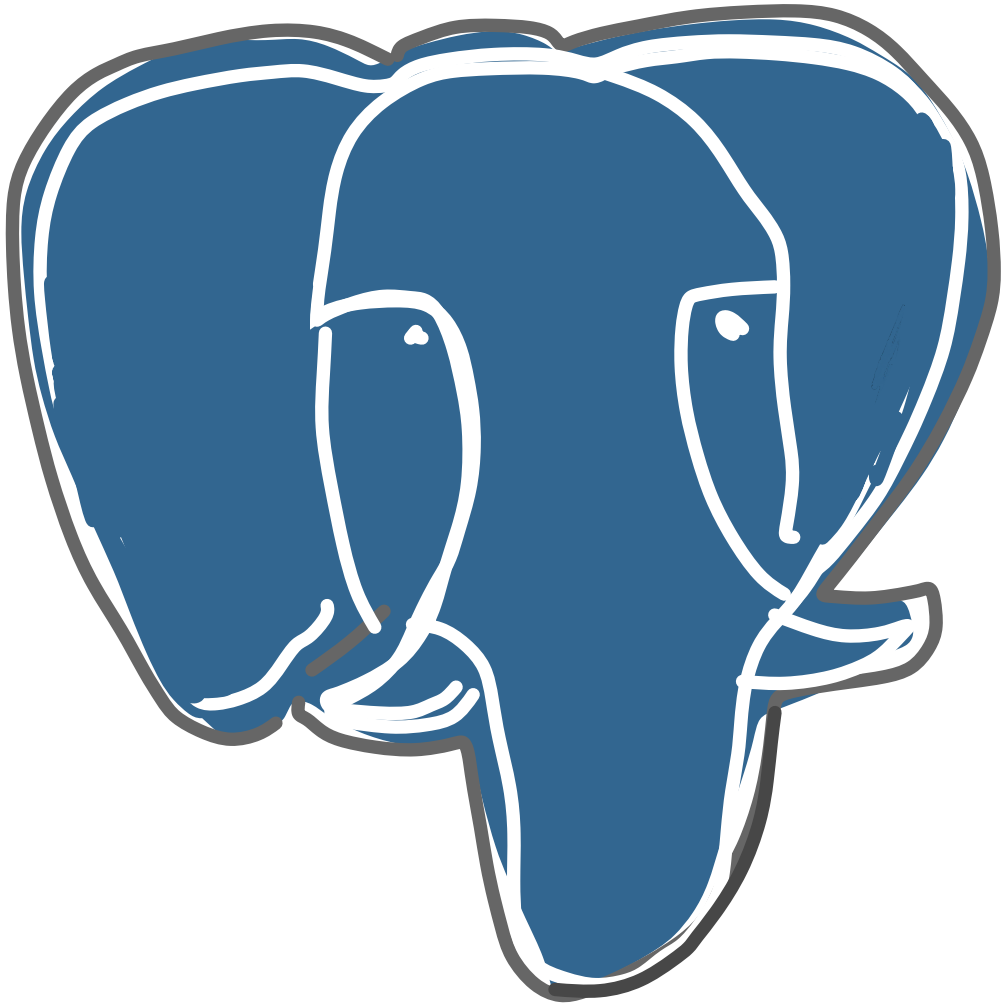
- Garbage Reads
- Incompatible Versions
- Internal Consistency Violations
- Duplicate Writes

$O(n!)$

$O(n)$

1. Generate ops
2. Record history
3. Verify history is consistent w/ model

Network Partitions

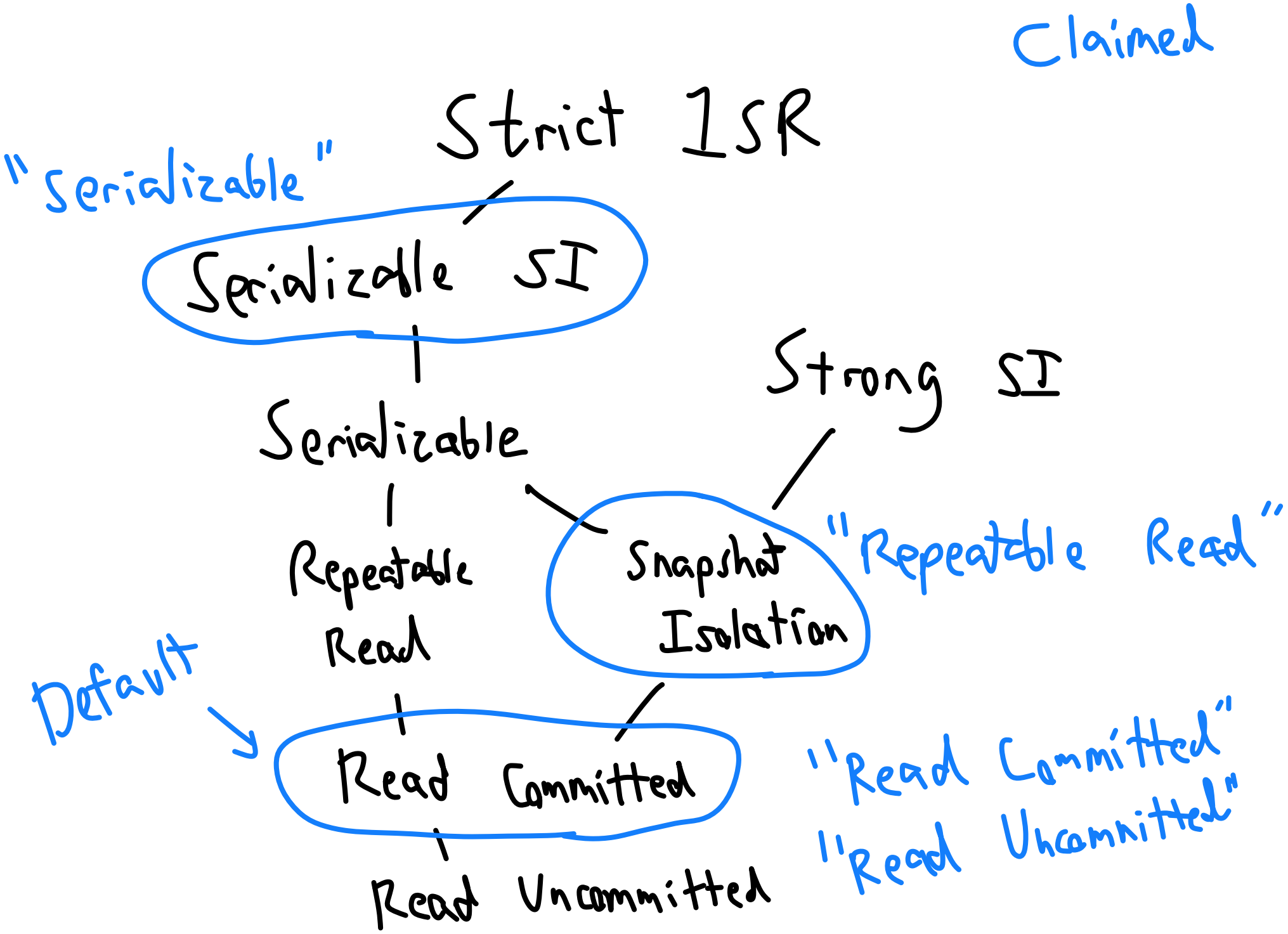


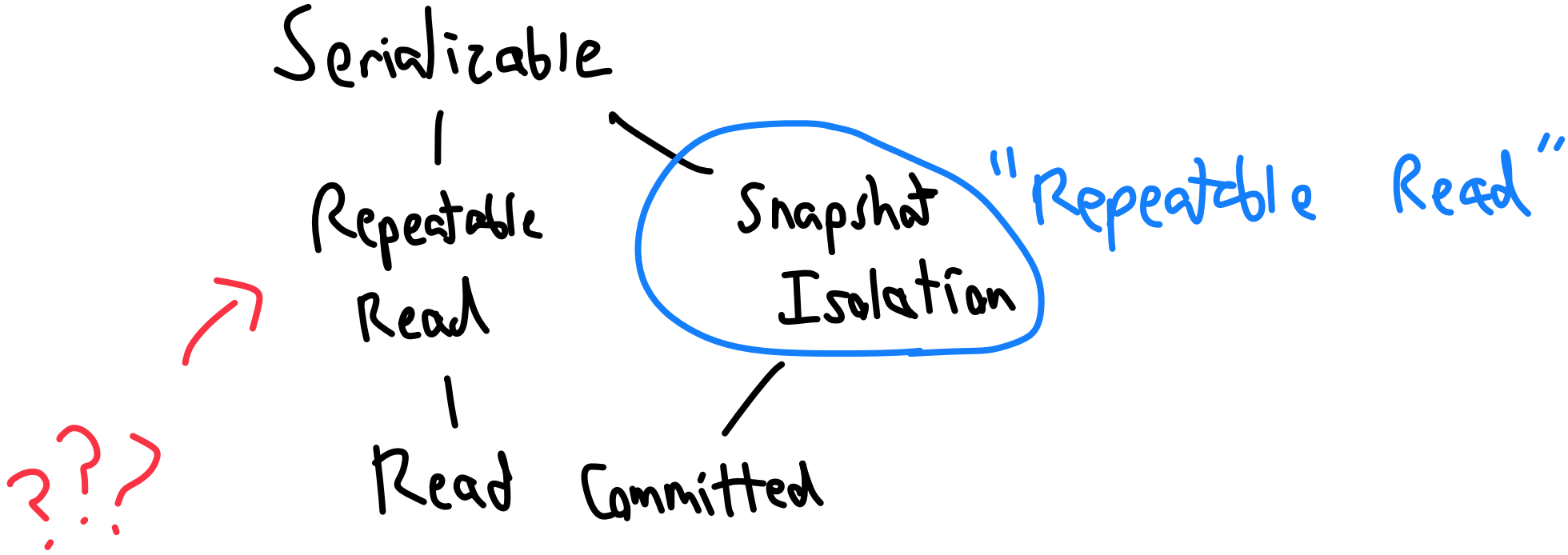
POSTGRESQL

12.3

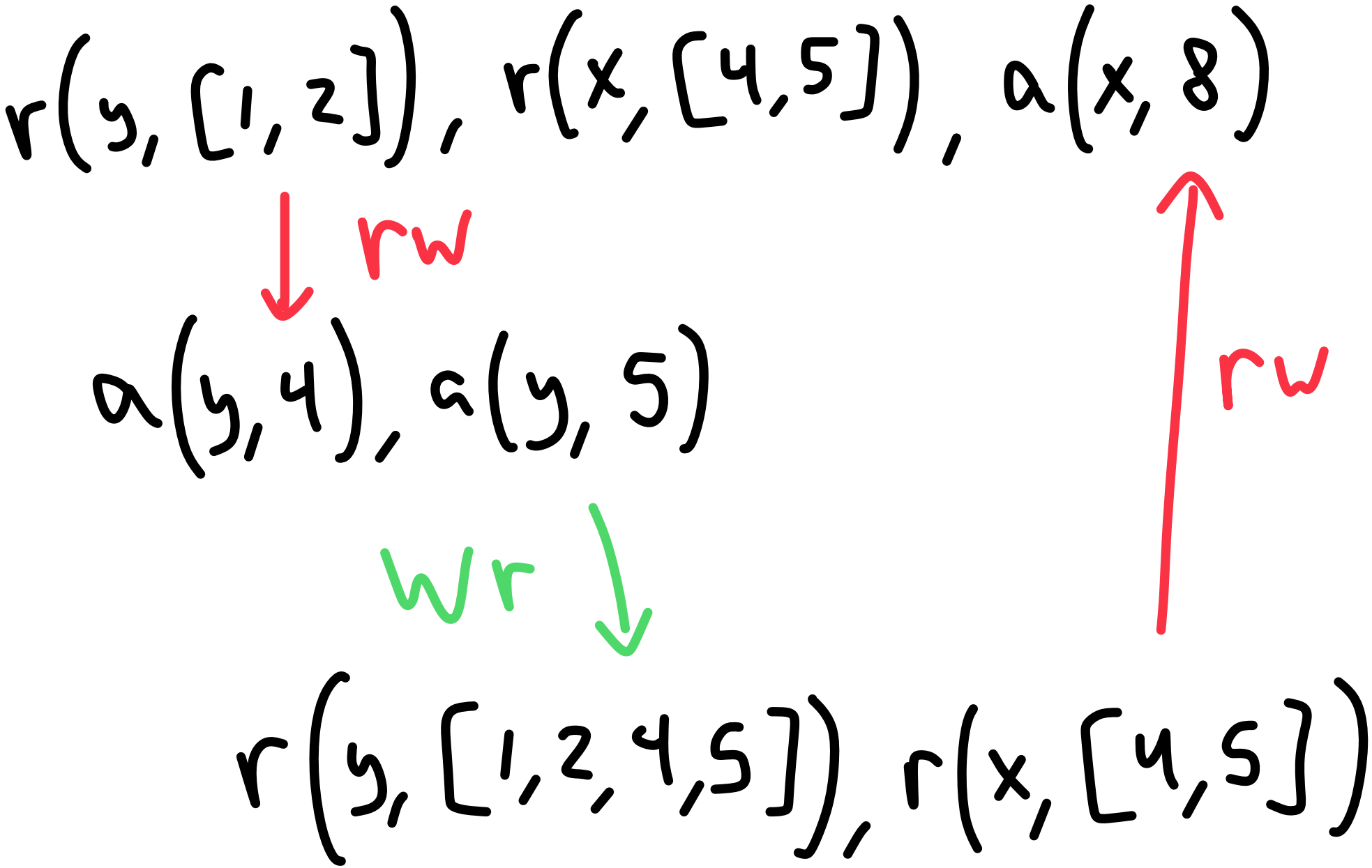
Single-node

SQL database


Claimed
Strict 1SR
"Serializable"
Serializable SI
Strong SI
Serializable
Repeatable Read
Snapshot Isolation
"Repeatable Read"
Default
Read Committed
"Read Committed"
"Read Uncommitted"
Read Uncommitted


Serializable
Repeatable Read
Read
Committed
Snapshot Isolation
"Repeatable Read"
???

Okay, let's try it!


r(y, [1, 2]), r(x, [4, 5]), a(x, 8)
rw
a(y, 4), a(y, 5)
wr
rw
r(y, [1, 2, 4, 5]), r(x, [4, 5])

G2-Item

Illegal under RR

Legal under SI

But PG claims "RR"

is stronger

than SQL

standard RR!

SQL standard is

aMbIguous

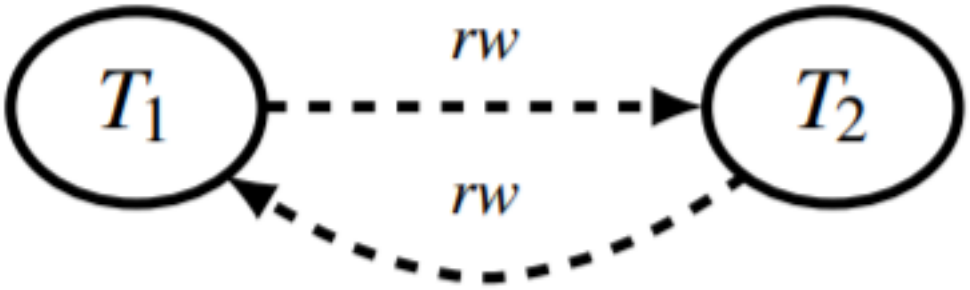
(Berenson & Adya argue the spec is bad)

Ok haw bout

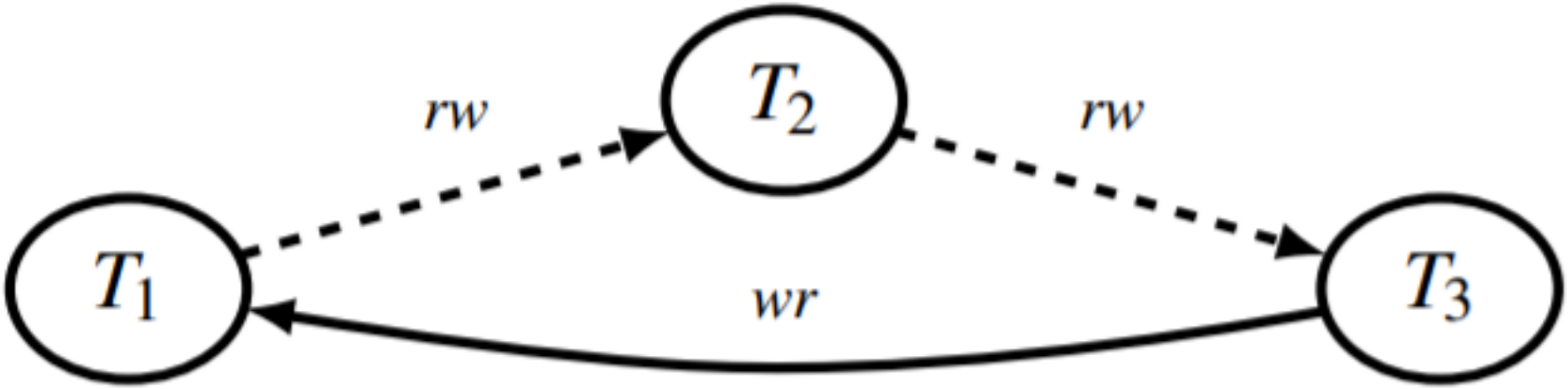
SERIALIZABLE?

SI, but also prevent

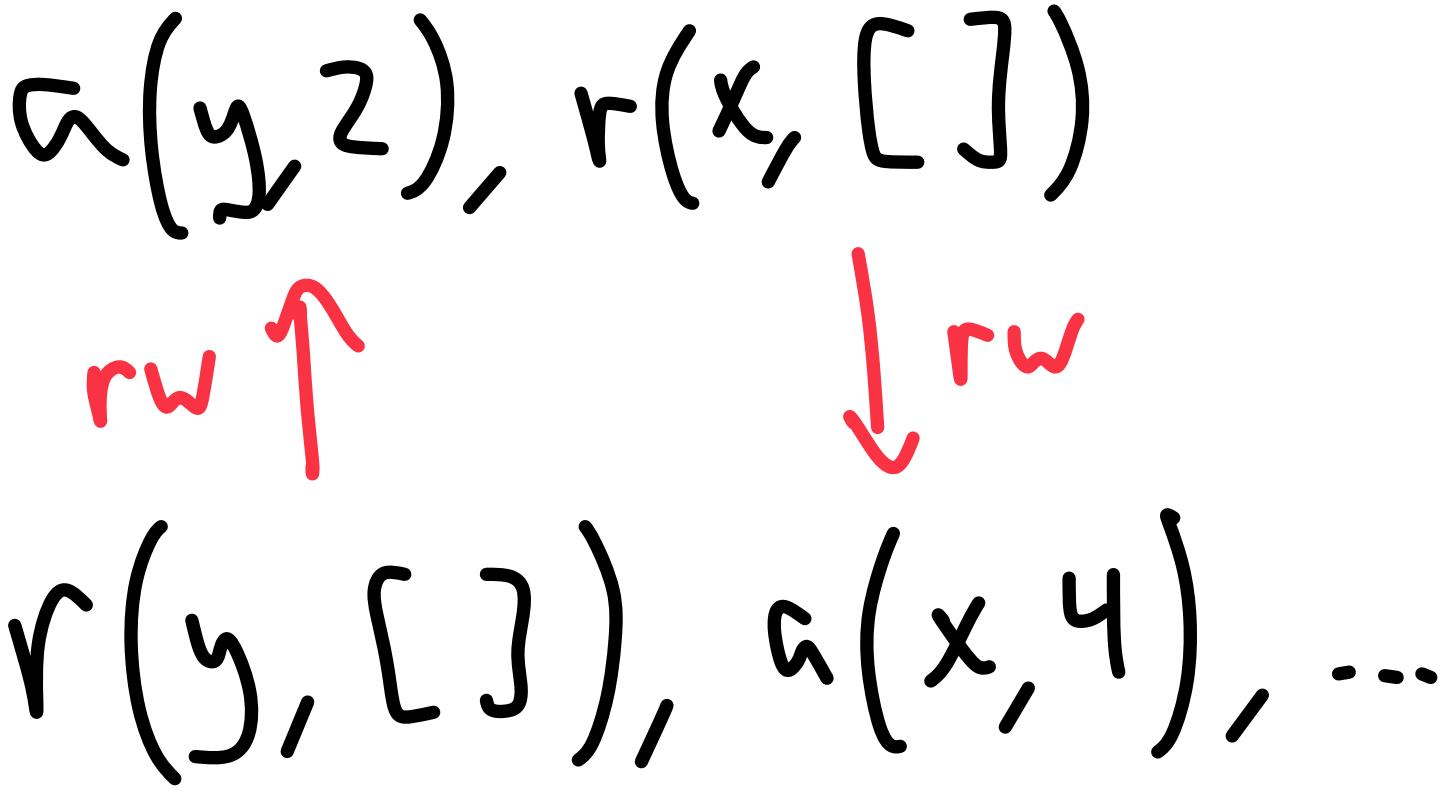
"Dangerous Structures"



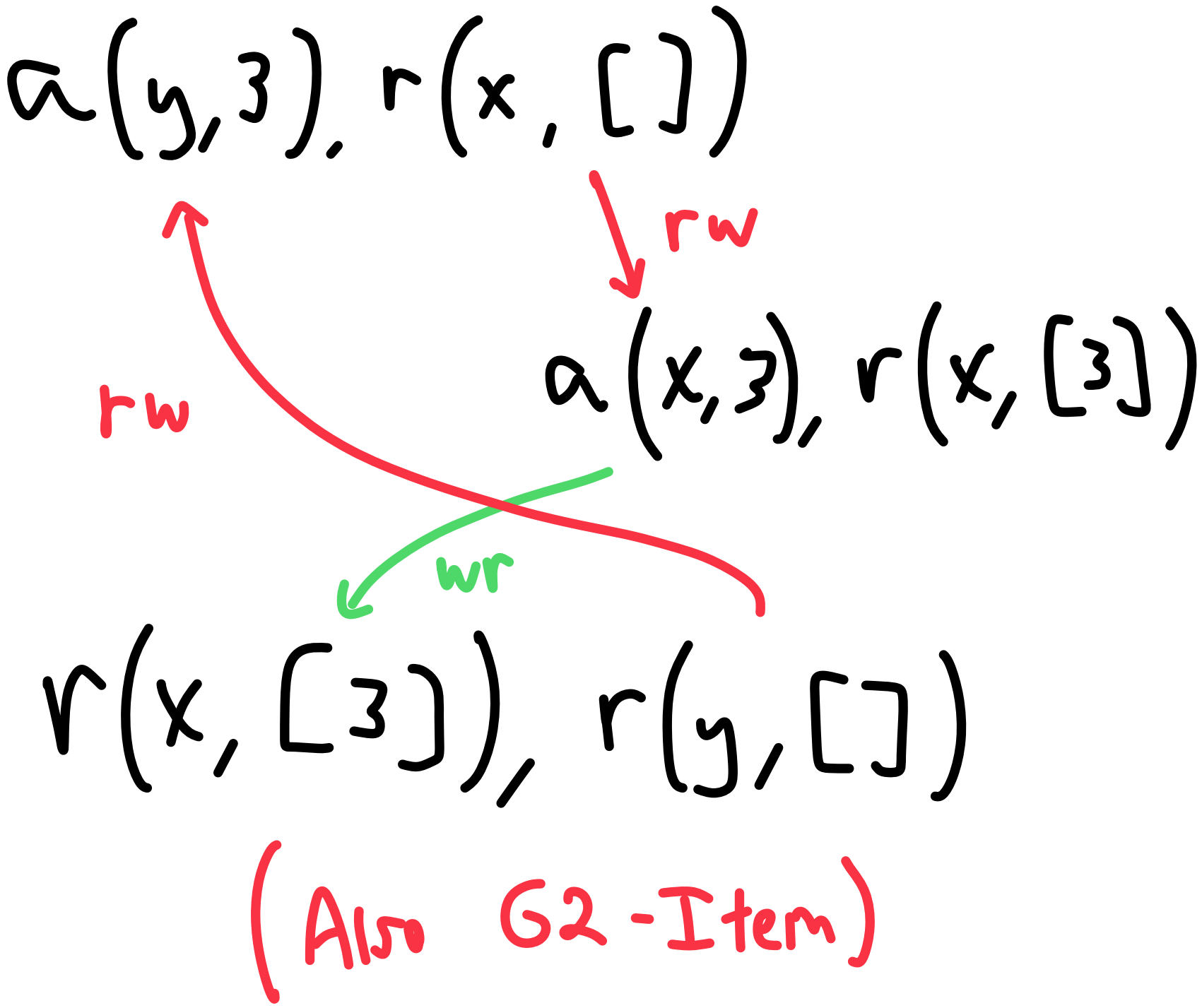
(a) Example 1: Simple Write Skew

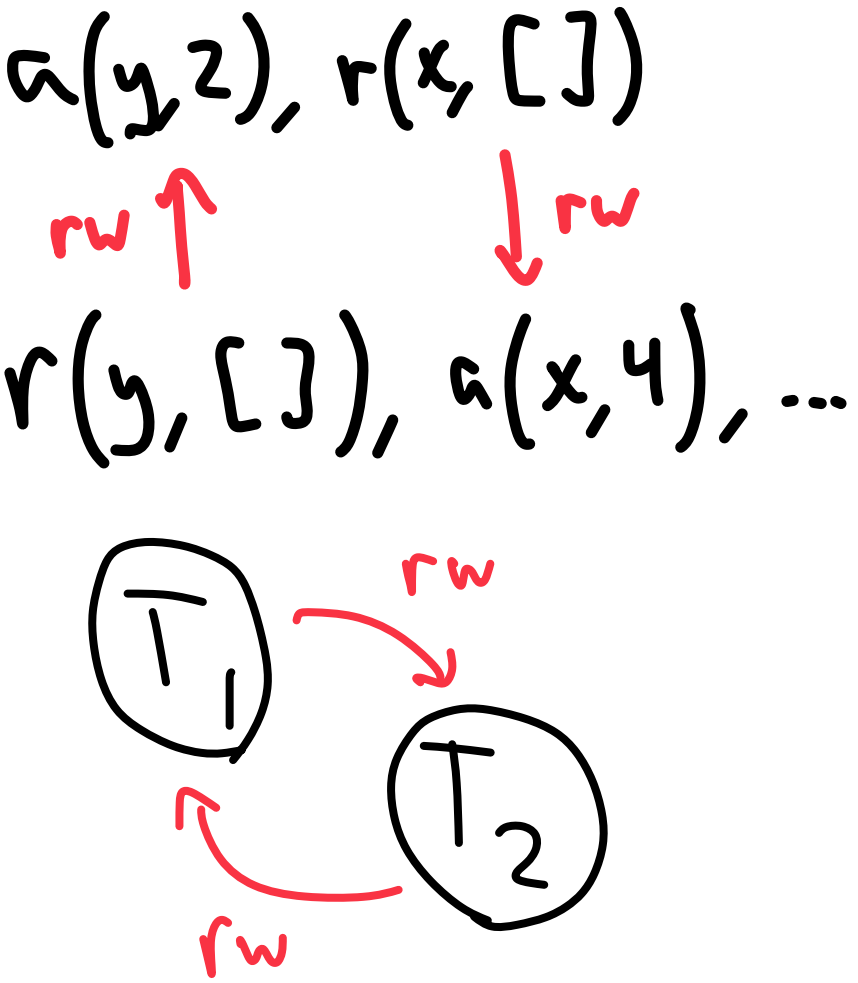


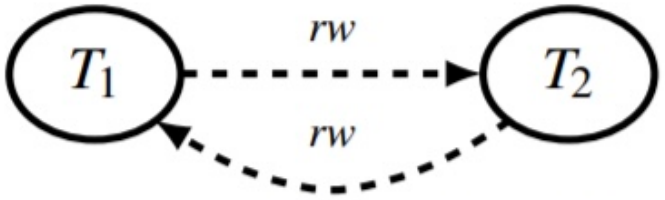
(b) Example 2: Batch Processing



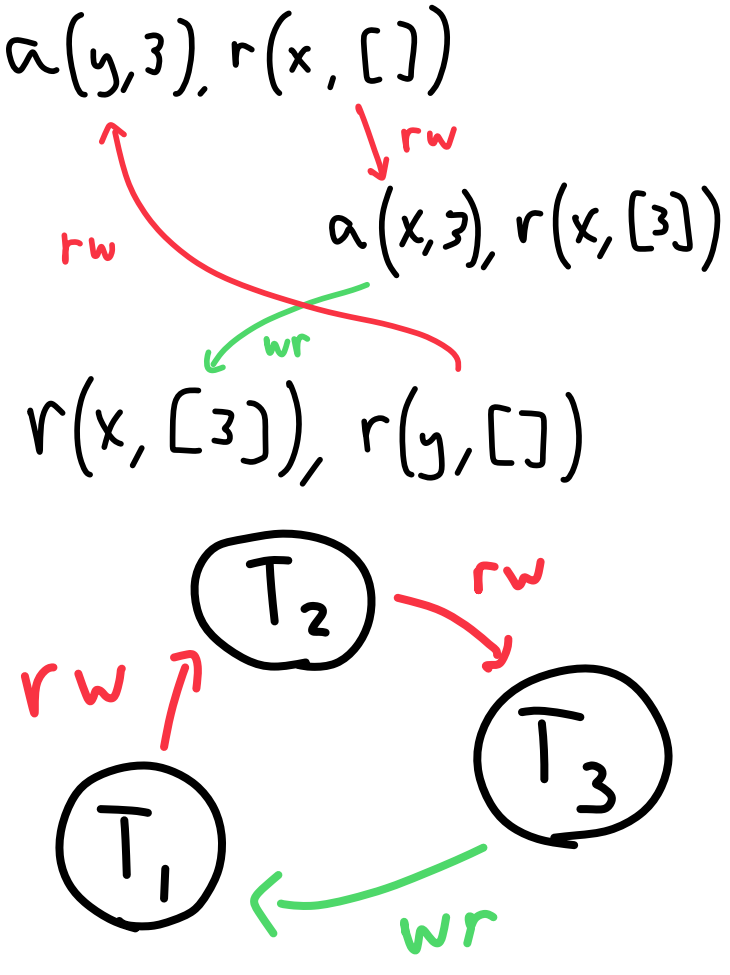
G2-Item

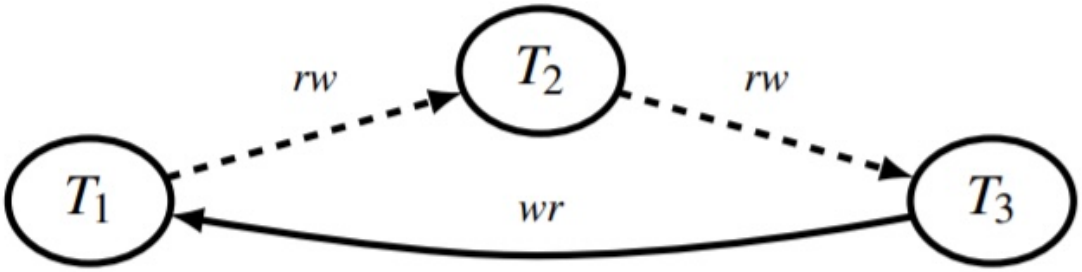

a(y,3), r(x,[])
rw
rw
a(x,3), r(x,[3])
wr
r(x,[3]), r(y,[])
(Also G2-Item)



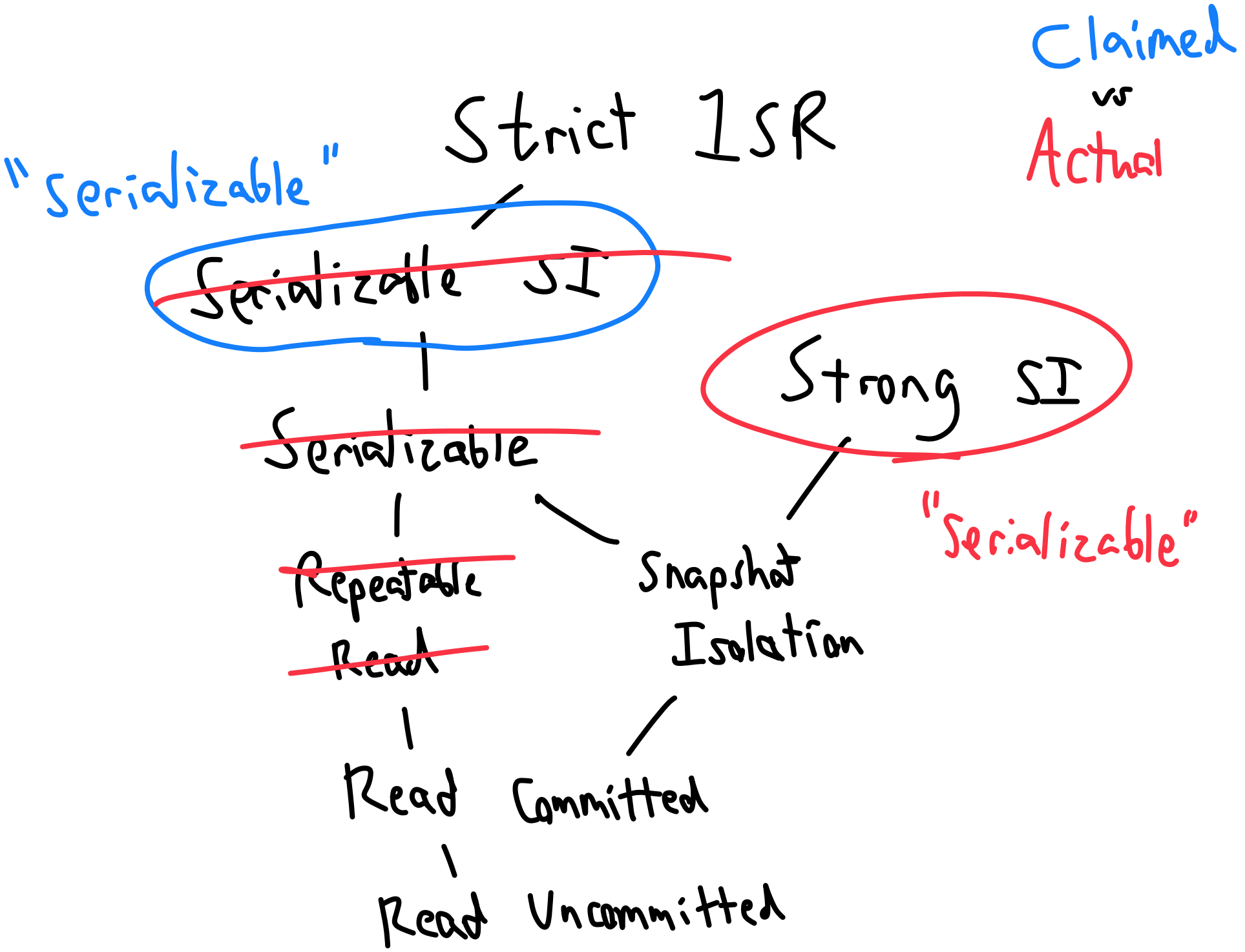


(a) Example 1: Simple Write Skew





(b) Example 2: Batch Processing

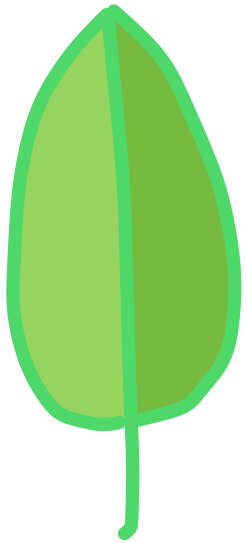

Claimed
vs
Actual
Strict 1SR
"Serializable"
Serializable SI
Strong SI
"Serializable"
Serializable
Repeatable Read
Snapshot Isolation
Read Committed
Read Uncommitted

Conflict detection mechanism incorrectly tagged one txn as writer of both old & new versions

→ txn could commit failing to observe another's writes!

- 9.5.22 → 9.5.23
- 10.13 → 19.14
- 11.8 → 11.9
- 12.3 → 12.4
- 13 (dev) → 13.0


"Serializable"
Best guess
Strict 1SR
Serializable SI
"Repeatable Read"
Strong SI
Serializable
Repeatable Read
Snapshot Isolation
Read Committed
Read Uncommitted
"Read Committed"
"Read Uncommitted"

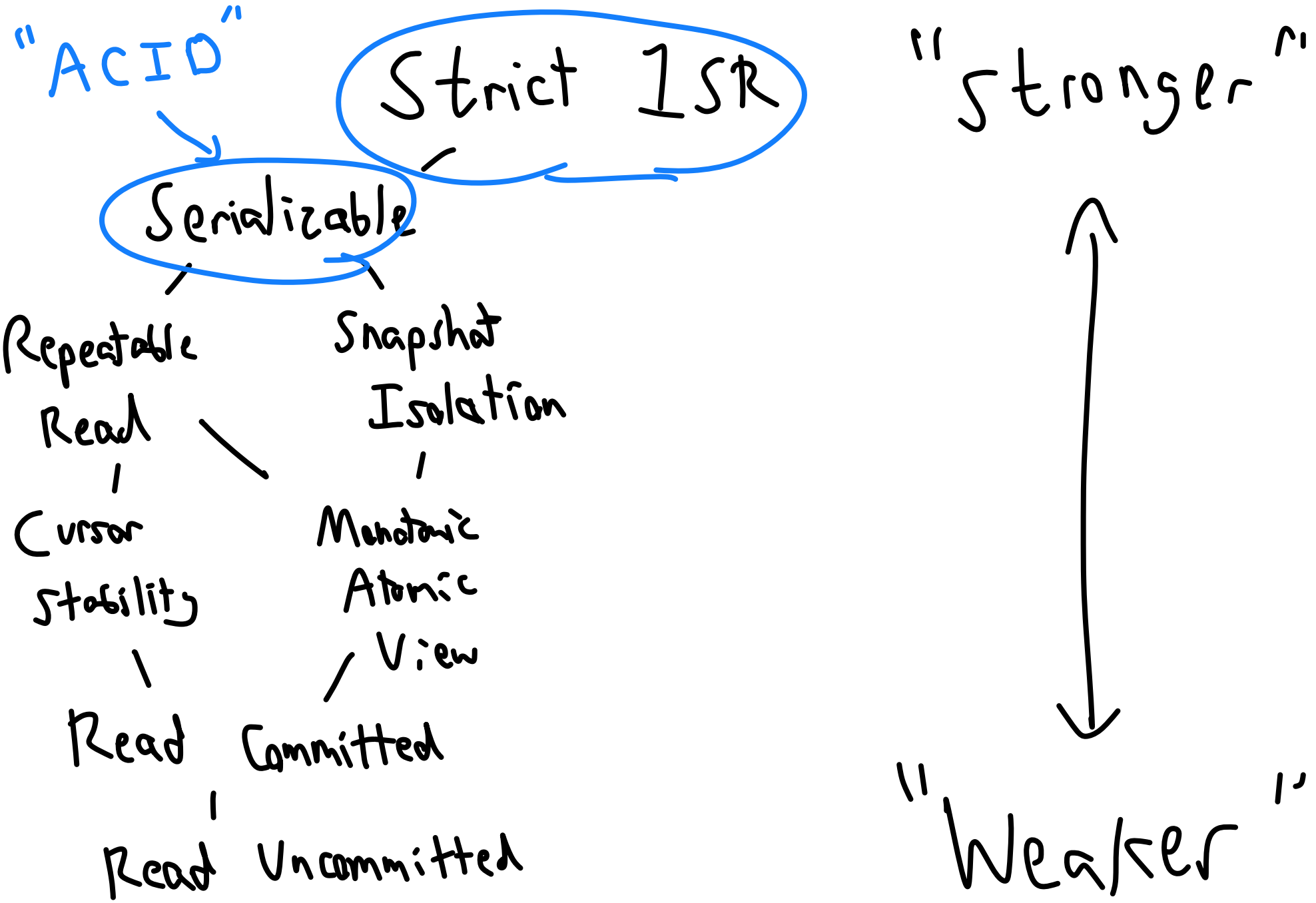


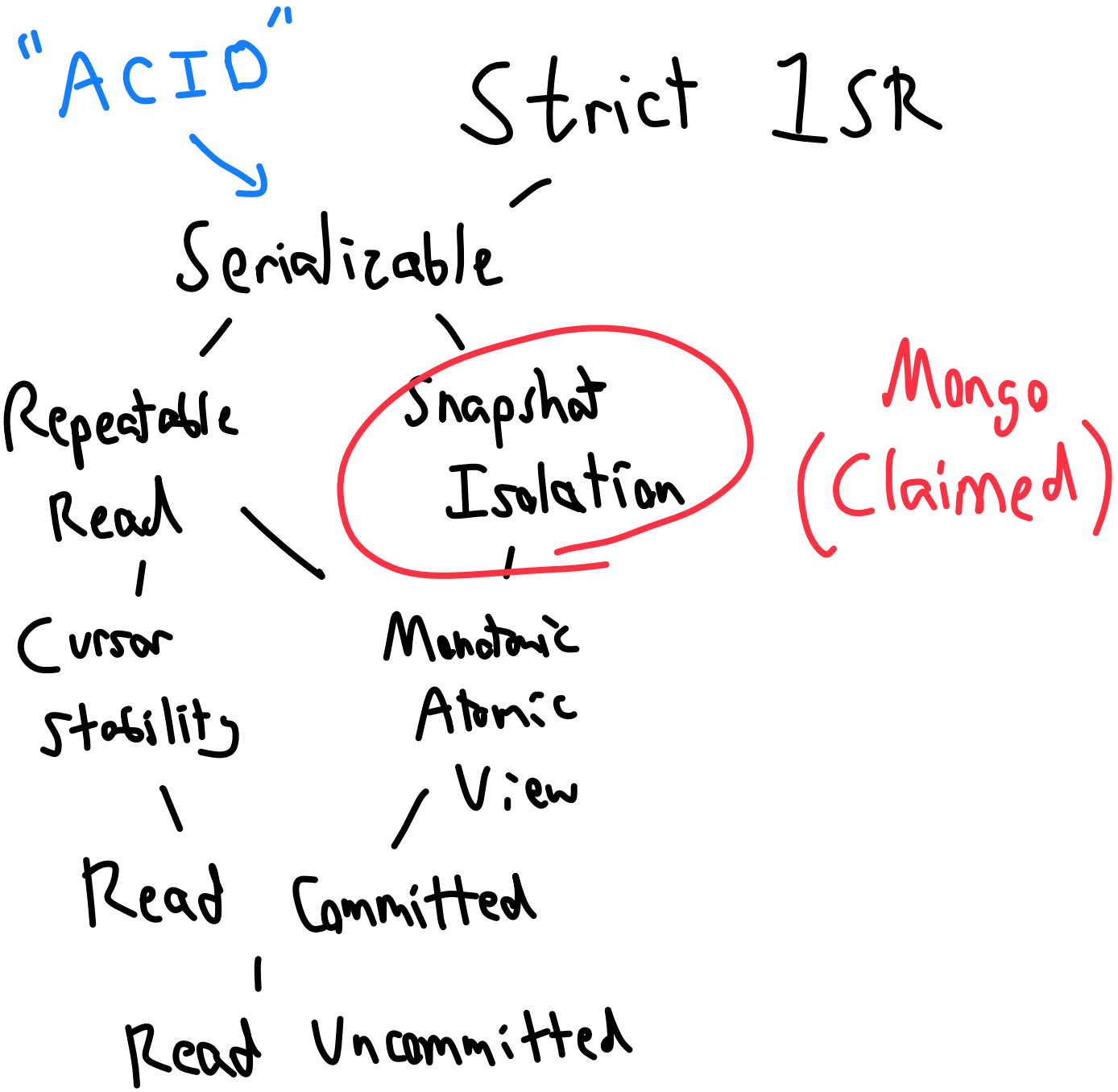
MongoDB

4.2.6

"... among the strongest data Consistency, Correctness, and Safety guarantees of any database available today"

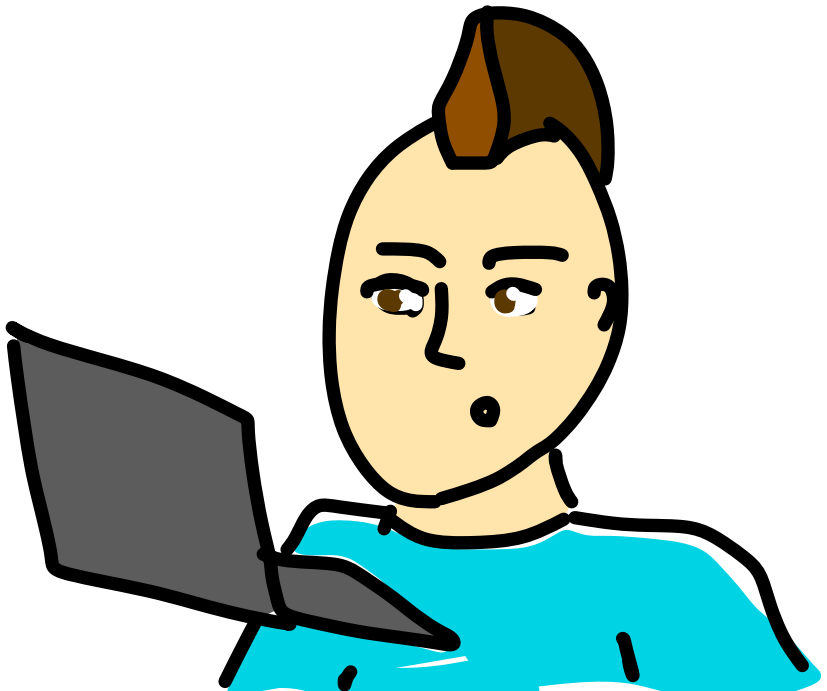
"Full ACID transactions"


"ACID"
Strict 1SR
Serializable
Repeatable Read
Snapshot Isolation
Cursor Stability
Monotonic Atomic View
Read Committed
Read Uncommitted
"Stronger"
"Weaker"


"ACID"
Strict 1SR
Serializable
Repeatable Read
Snapshot Isolation
Mongo (Claimed)
Cursor Stability
Monotonic Atomic View
Read Committed
Read Uncommitted

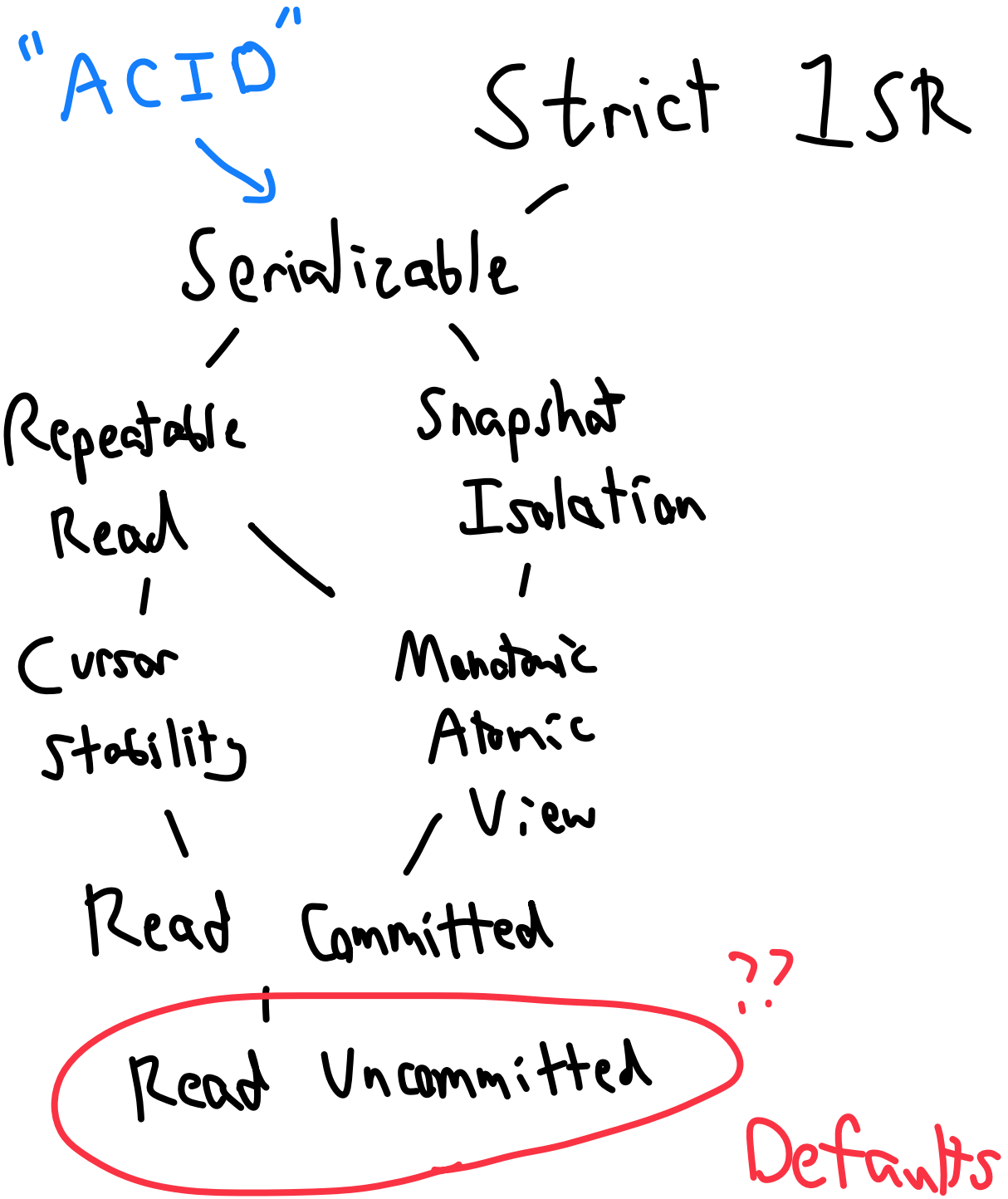
Assumes:

- Majority Writes
- Snapshot Reads



Defaults:

- "Committed" writes may be thrown away silently
- Reads "whatever a single node has locally"


"ACID"
Strict 1SR
Serializable
Repeatable Read
Snapshot Isolation
Cursor Stability
Monotonic Atomic View
Read Committed
Read Uncommitted
??
Defaults

GO: Write Cycle

T_1: $w(x_1)$ $w(y_2)$

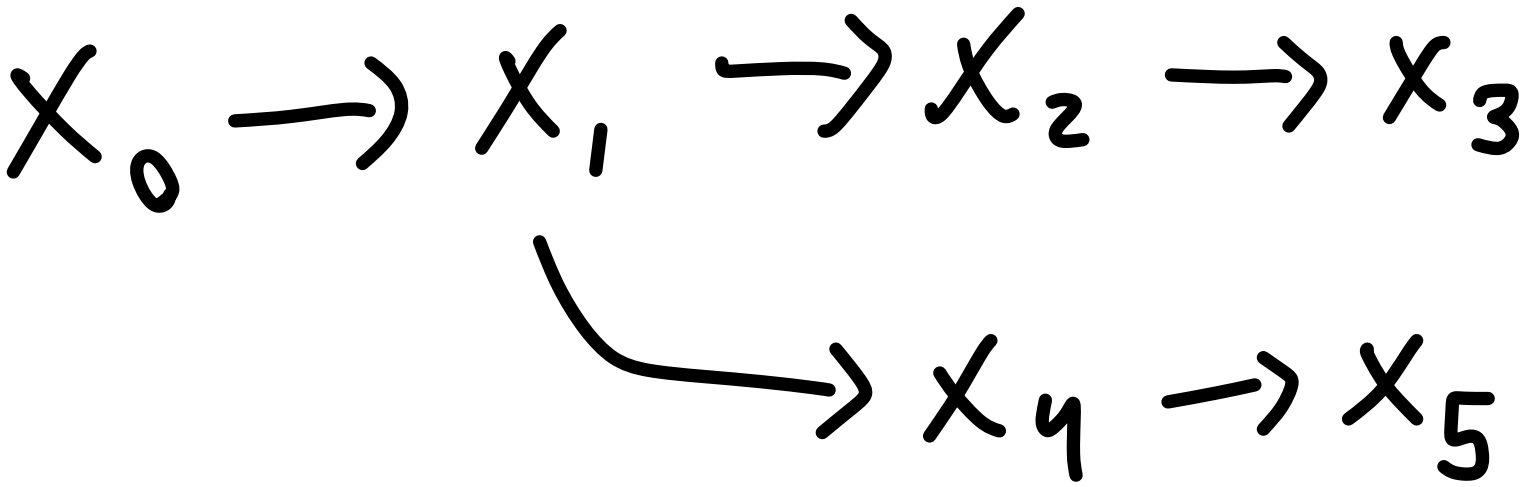
ww ↓ ↑ ww

T_2: $w(x_2)$ $w(y_1)$

Requires a total order over committed versions

$$x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow \cdots$$

Mongo Allows Split-Brain!



$x_3 \ll x_5$???


"ACID"
Strict 1SR
Serializable
Repeatable Read
Snapshot Isolation
Cursor Stability
Monotonic Atomic View
Read Committed
Read Uncommitted
Mongo DB defaults?

```
coll = db.withReadConcern(LINEARIZABLE)
       .withWriteConcern(MAJORITY)...

coll.updateOne(..., newVersion)
```

→ SI / Linearizable

single-doc ops

```
coll = db.withReadConcern(LINEARIZABLE)
        .withWriteConcern(MAJORITY)...
session = conn.startSession()
session.withTransaction(...
  [coll.updateOne(session..., newVersion)])
```

→ Lost Writes

Aborted Reads

Transactions silently

downgrade read concern

to LOCAL, by ignoring

DB & Collection-level

read concern.

```
TransactionOptions.Builder().
    .readConcern(SNAPSHOT)
```

→ Split-brain

Aborted reads

Definitely not Snapshot Isol.

Read Concern SNAPSHOT isn't actually a snapshot read unless you also use Write Concern MAJORITY

Even for read-only txns!?

r(x, [])

r(x, [1,5])

r(x, [1,5,6])

r(x, [])

r(x, [3])

r(x, [3,7])

uhhh...

so are we just pretending 1,5,6 never happened?

```
TransactionOptions.Builder().
    .readConcern(SNAPSHOT)
    .writeConcern(MAJORITY)
```

This is all in the docs!

It's just surprising!

Hosted MongoDB Users

80%. Default Write Concern

99.6%. Default Read Concern

(VLDB '19)

Mongo's own example used read concern LOCAL w/ write-only txns, or used only defaults.

Most third party blog

posts claimed SI/"ACID"

but inadvertently or intentionally

allowed write loss & aborted reads

How

ACID

is

— Snapshot Isolation —

anyway?

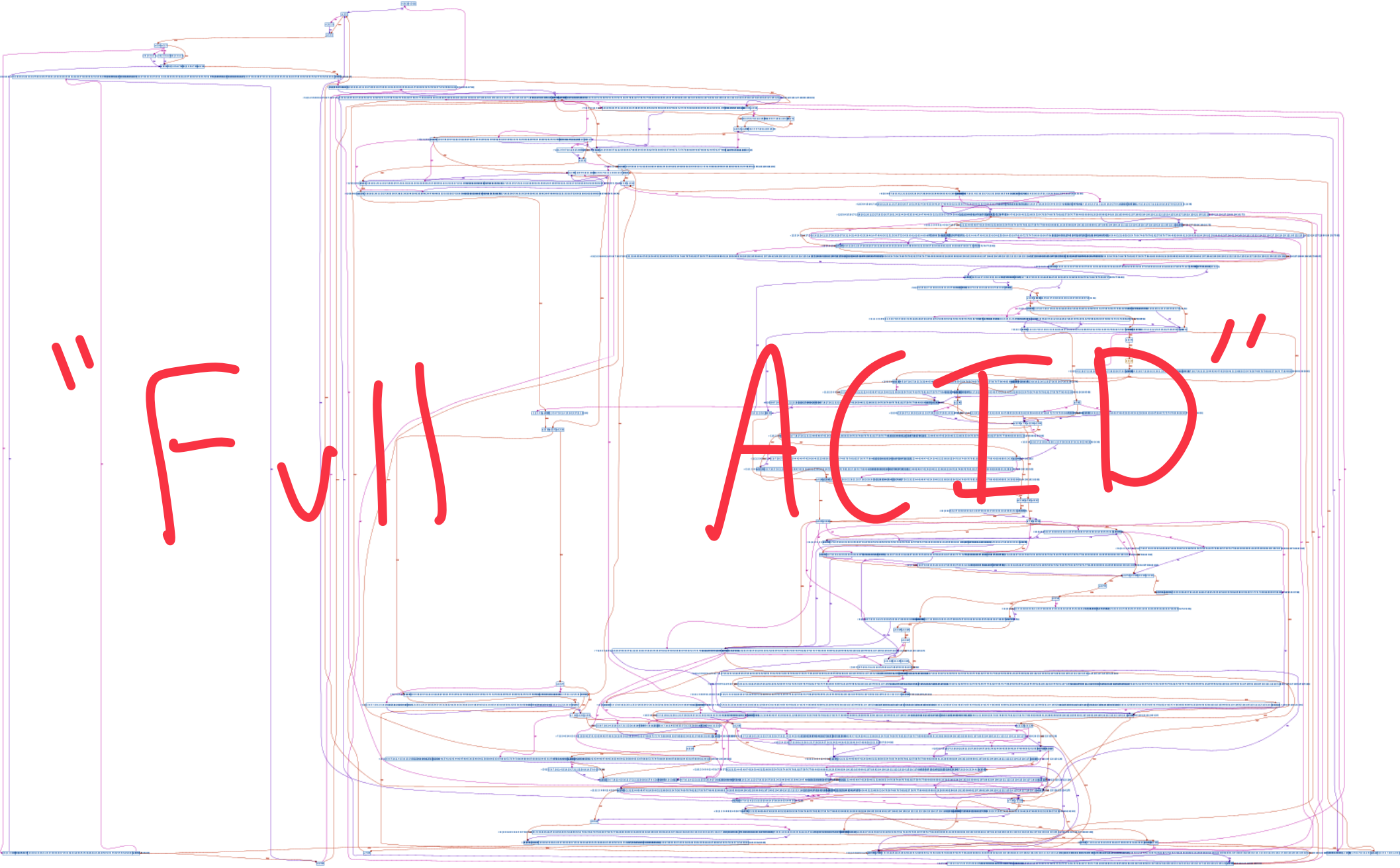
T_1: append(x, 1), read(y, [])

T_2: append(y, 1), read(x, [])

G2-item (write skew)

Legal under SI...

Definitely not isolated!



123 txns all depending on each other

~10% of txns violated isolation in healthy Mongo clusters.

- Does this matter?!?! Maybe...

Are MAJORITY/SNAPSHOT txns even Snapshot Isolated, though?

T_1: append(x, 6), ...

T_2: r(x, [2, 4, 1, 6, 8, 7, 6]),

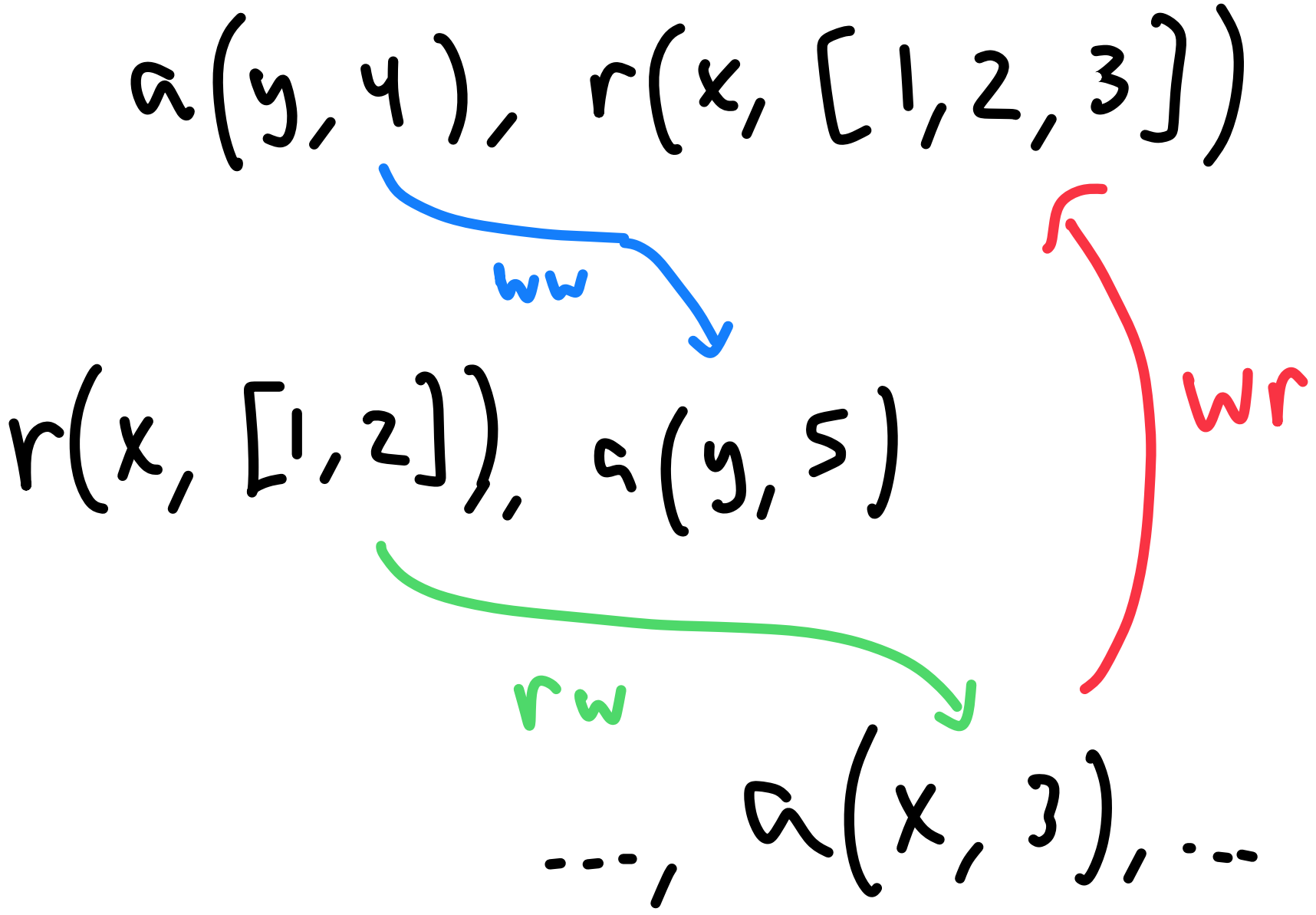
...

Maybe an improper txn retry?

Txns ignare the

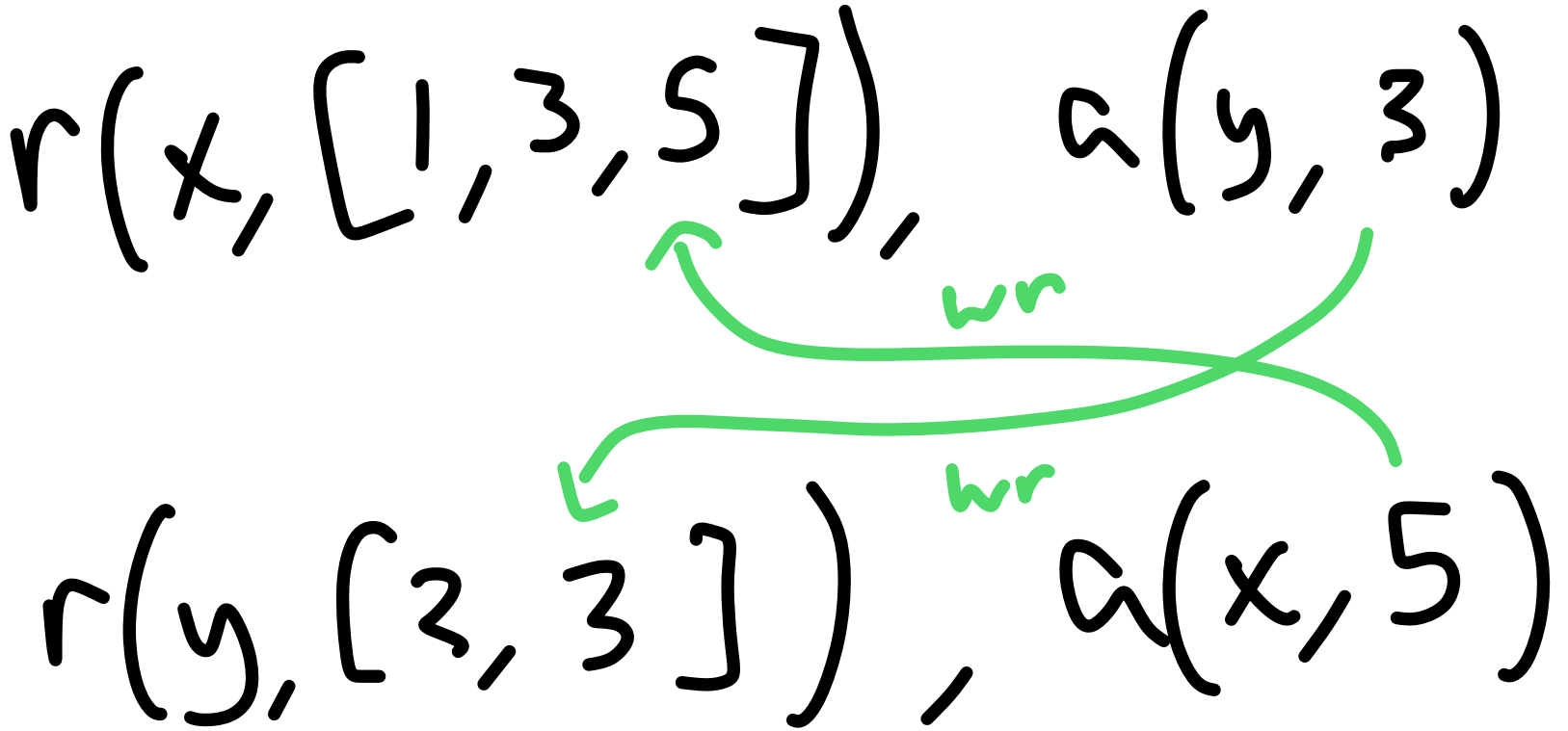
retryWrites setting $

retry regardless...


a(y, 4), r(x, [1, 2, 3])
ww
r(x, [1, 2]), a(y, 5)
wr
rw
..., a(x, 3), ...

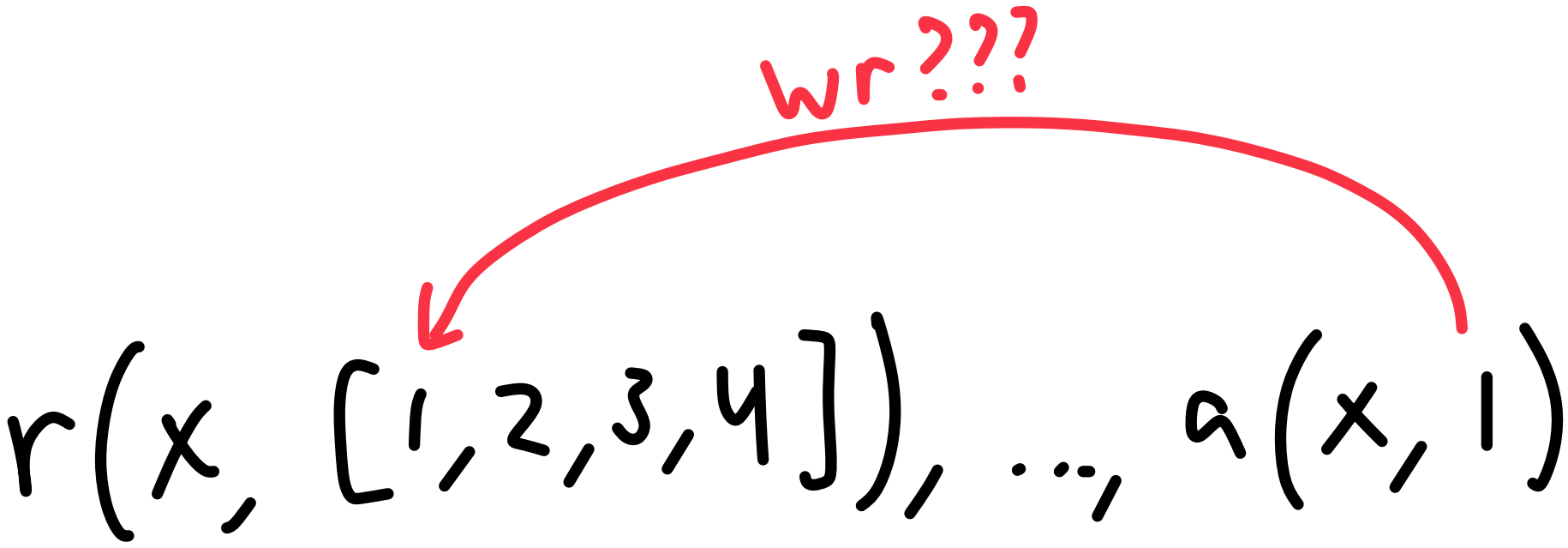
G-Single

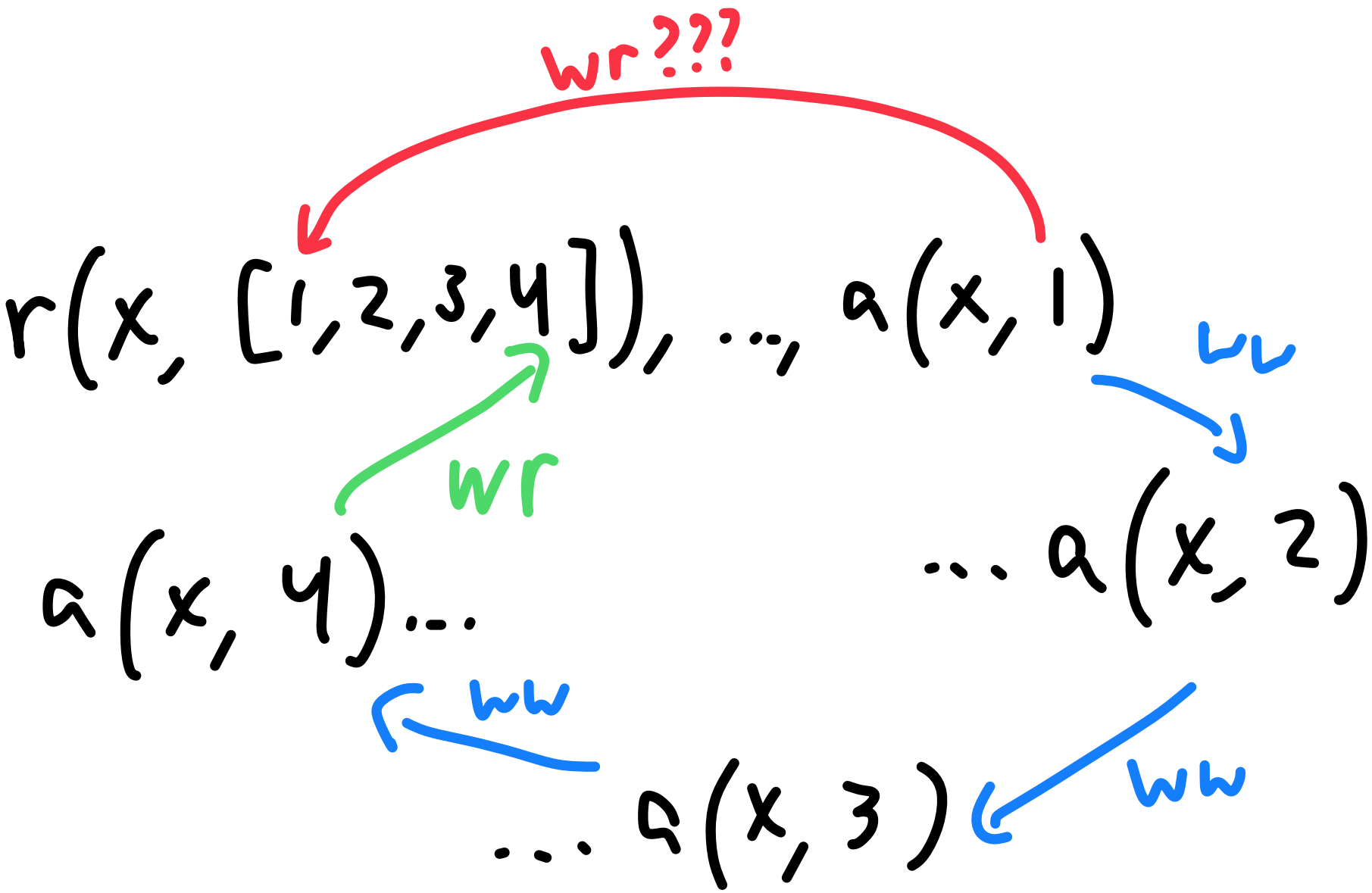
Read Skew


r(x, [1, 3, 5]), q(y, 3)
wr
wr
r(y, [2, 3]), q(x, 5)

G1C

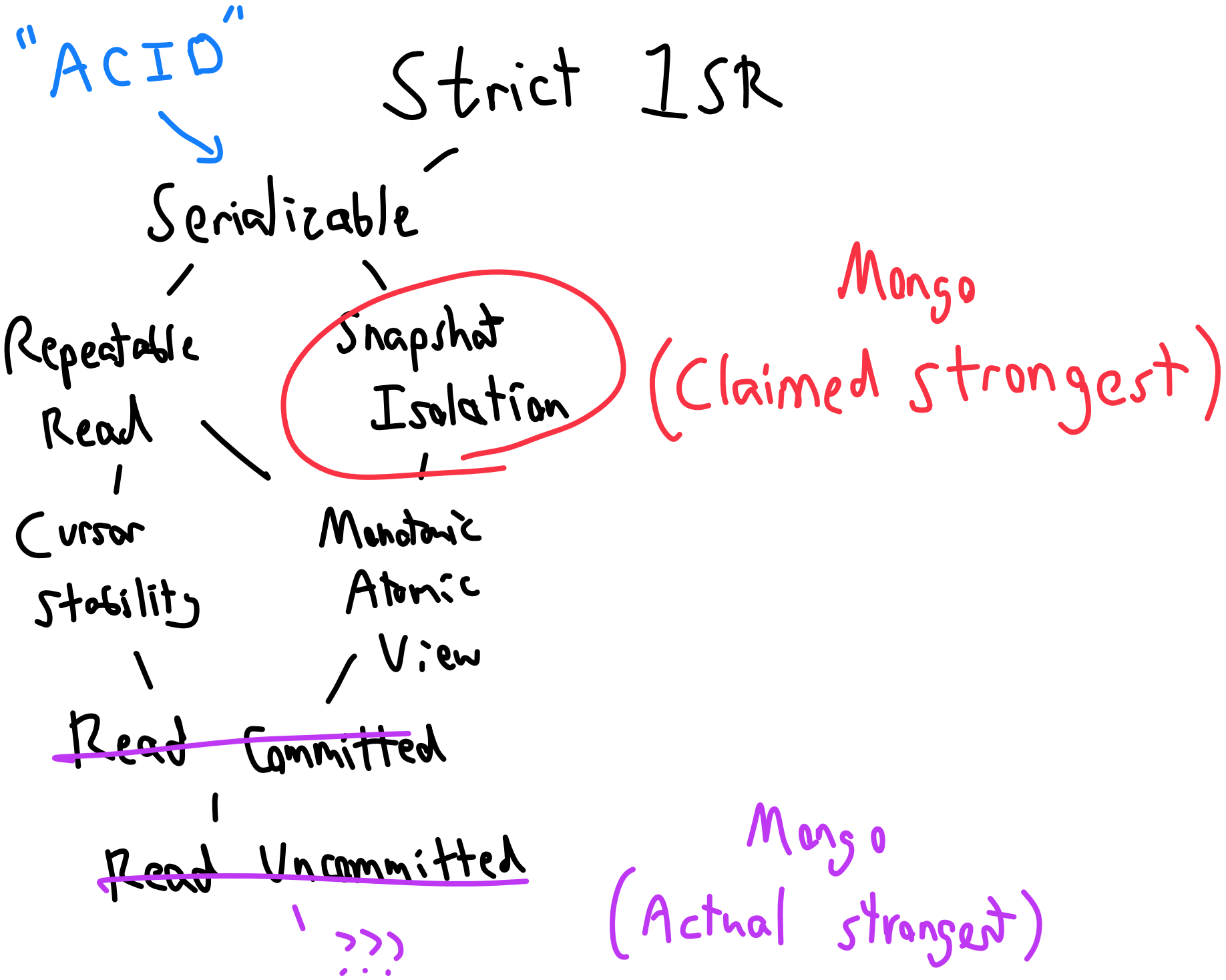
Cyclic Information Flow


wr???
r(x, [1,2,3,4]), ..., a(x, 1)


wr???
r(x, [1,2,3,4]), ..., a(x, 1)
ww
wr
a(x, 4)...
... a(x, 2)
ww
... a(x, 3)
ww

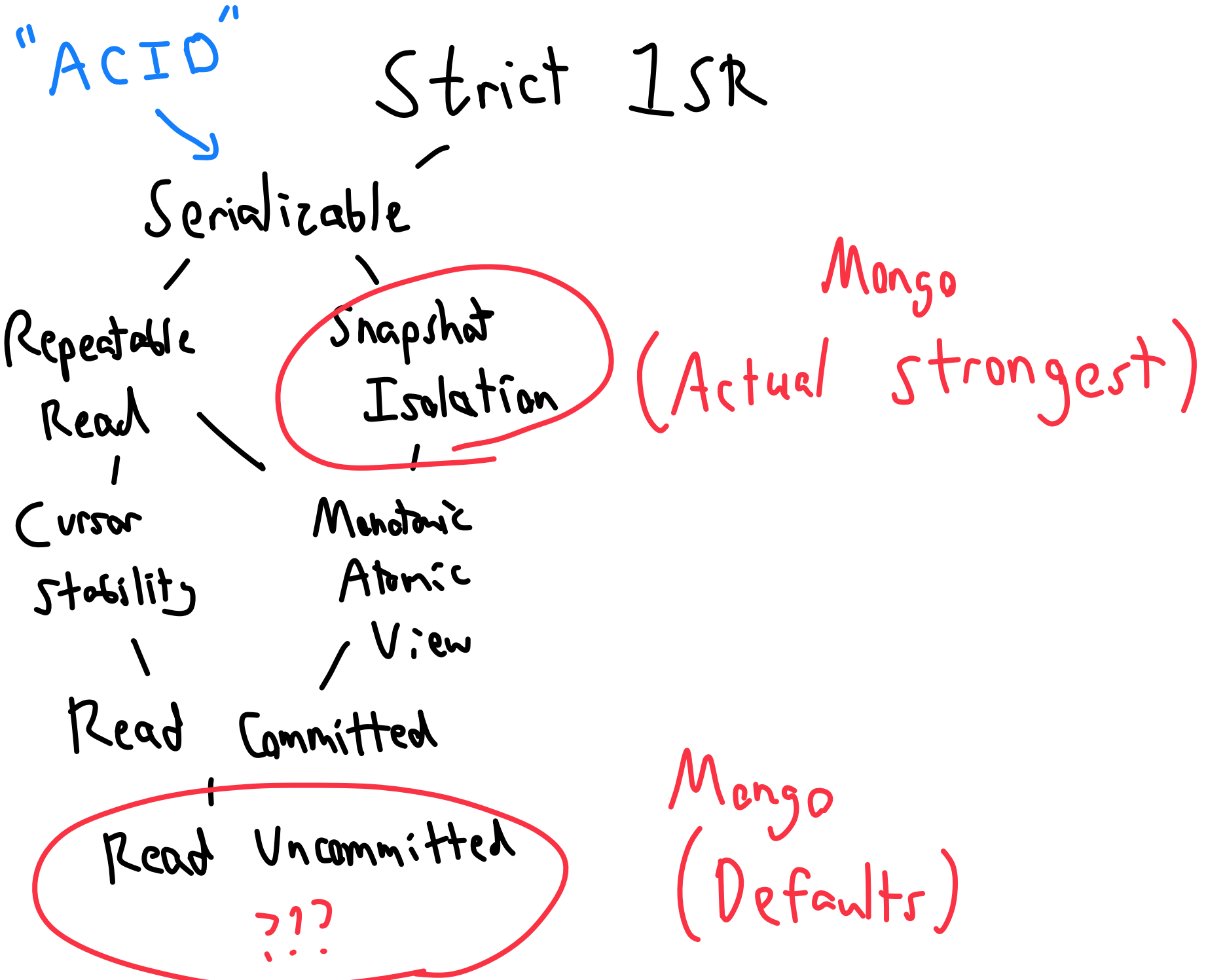
Internal Consistency

Violations


"ACID"
Strict 1SR
Serializable
Repeatable Read
Snapshot Isolation
Cursor Stability
Monotonic Atomic View
Read Committed
Read Uncommitted
???
Mongo
(Claimed strongest)
Mongo
(Actual strongest)

SERVER 48307

Committed txns could incorrectly appear to fail — fixed in 4.2.8

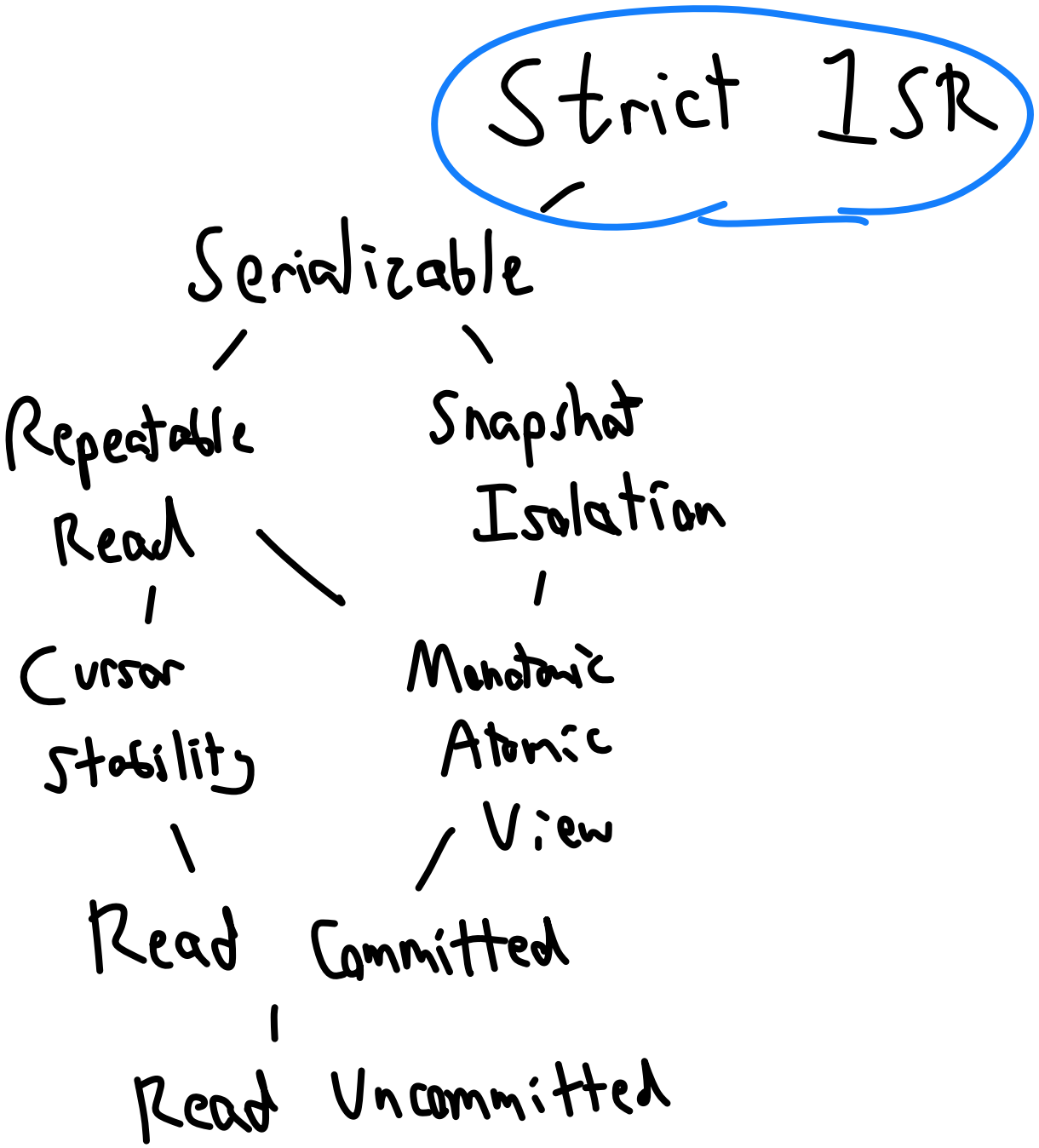

"ACID"
Strict 1SR
Serializable
Repeatable Read
Snapshot Isolation
Mongo (Actual strongest)
Cursor Stability
Monotonic Atomic View
Read Committed
Read Uncommitted
?!?
Mongo (Defaults)

redis

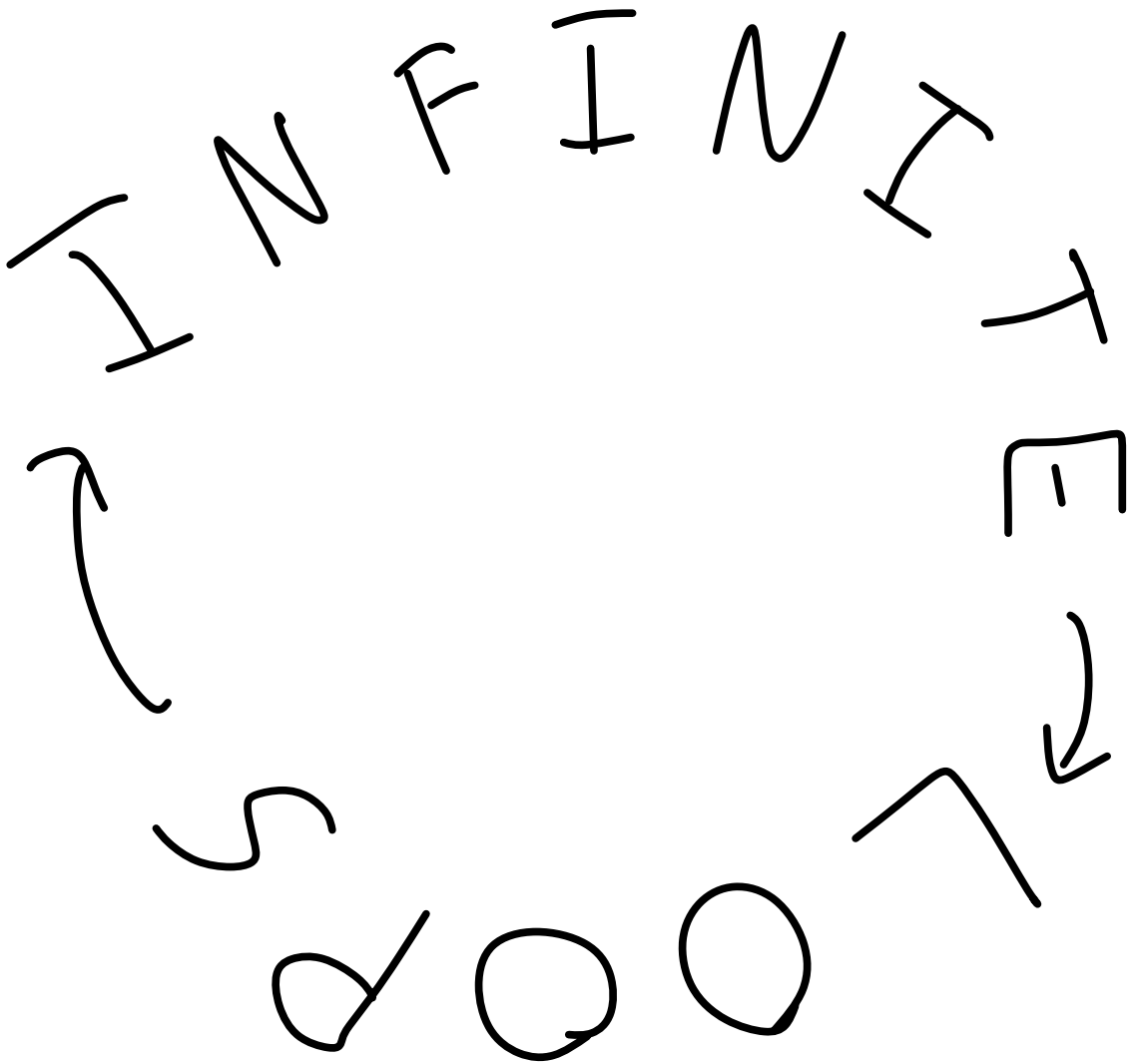
redis raft 163f6f6

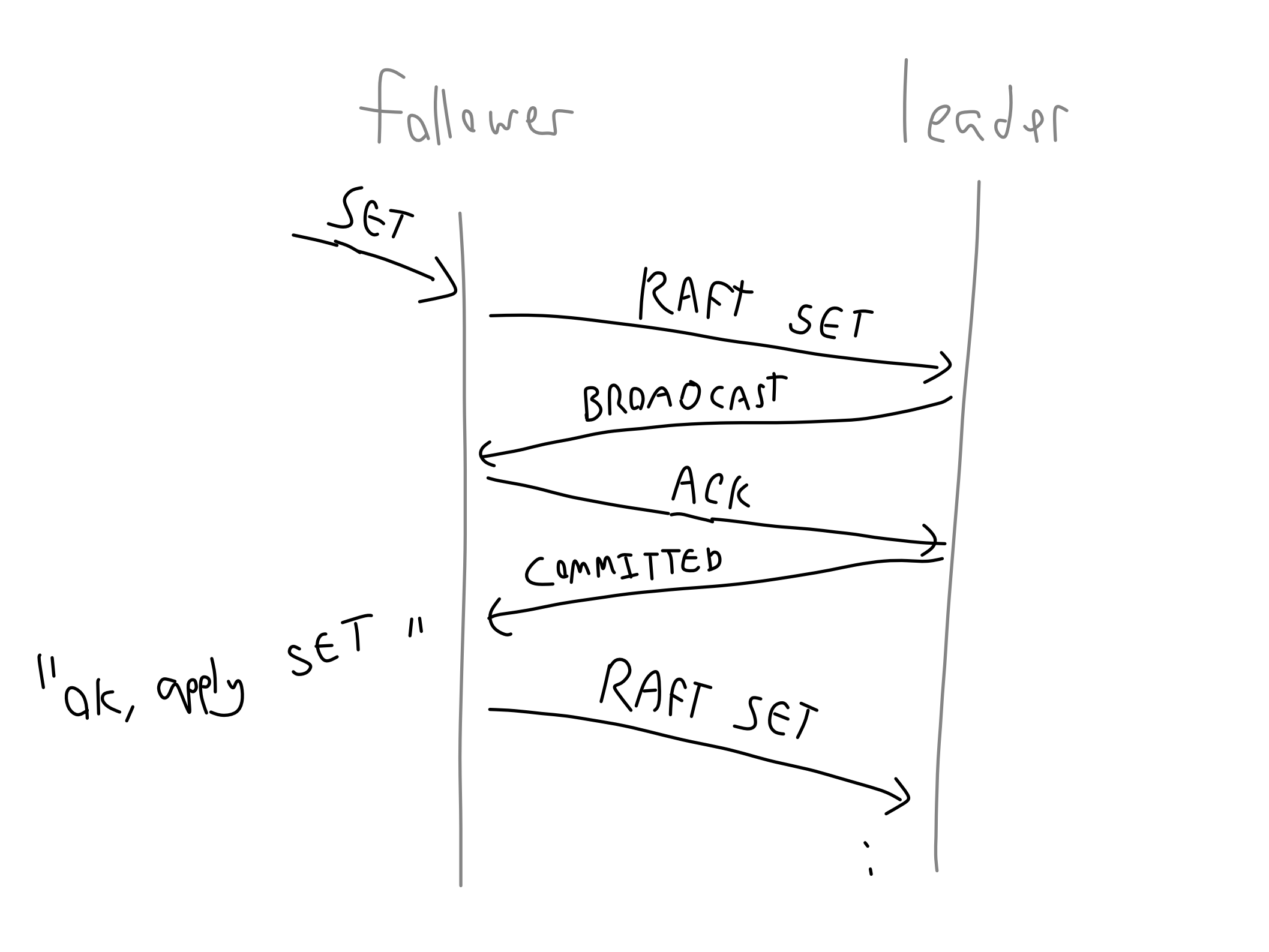
New replication system for Redis, based on Raft!

(Unreleased internal builds)


Strict 1SR
Serializable
Repeatable Read
Snapshot Isolation
Cursor Stability
Monotonic Atomic View
Read Committed
Read Uncommitted

#13



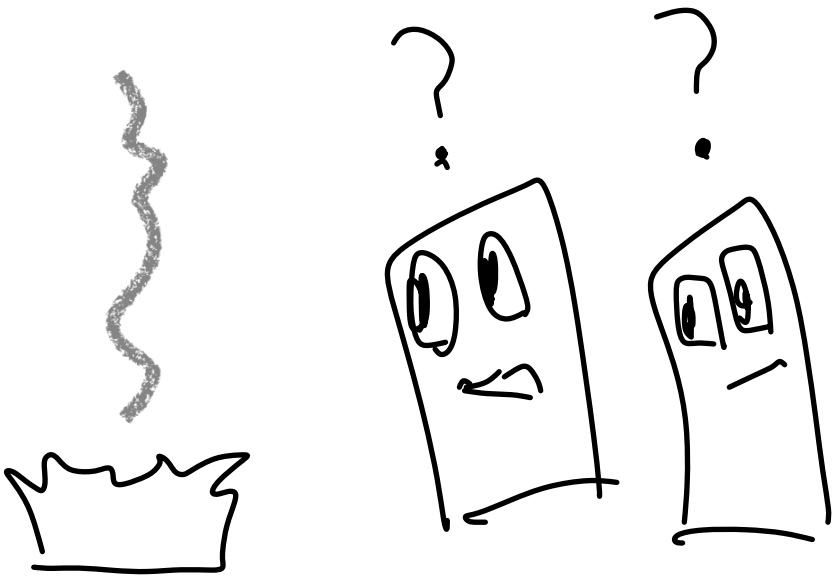

follower
leader
SET
RAFT SET
BROADCAST
ACK
COMMITTED
"ok, apply SET"
RAFT SET

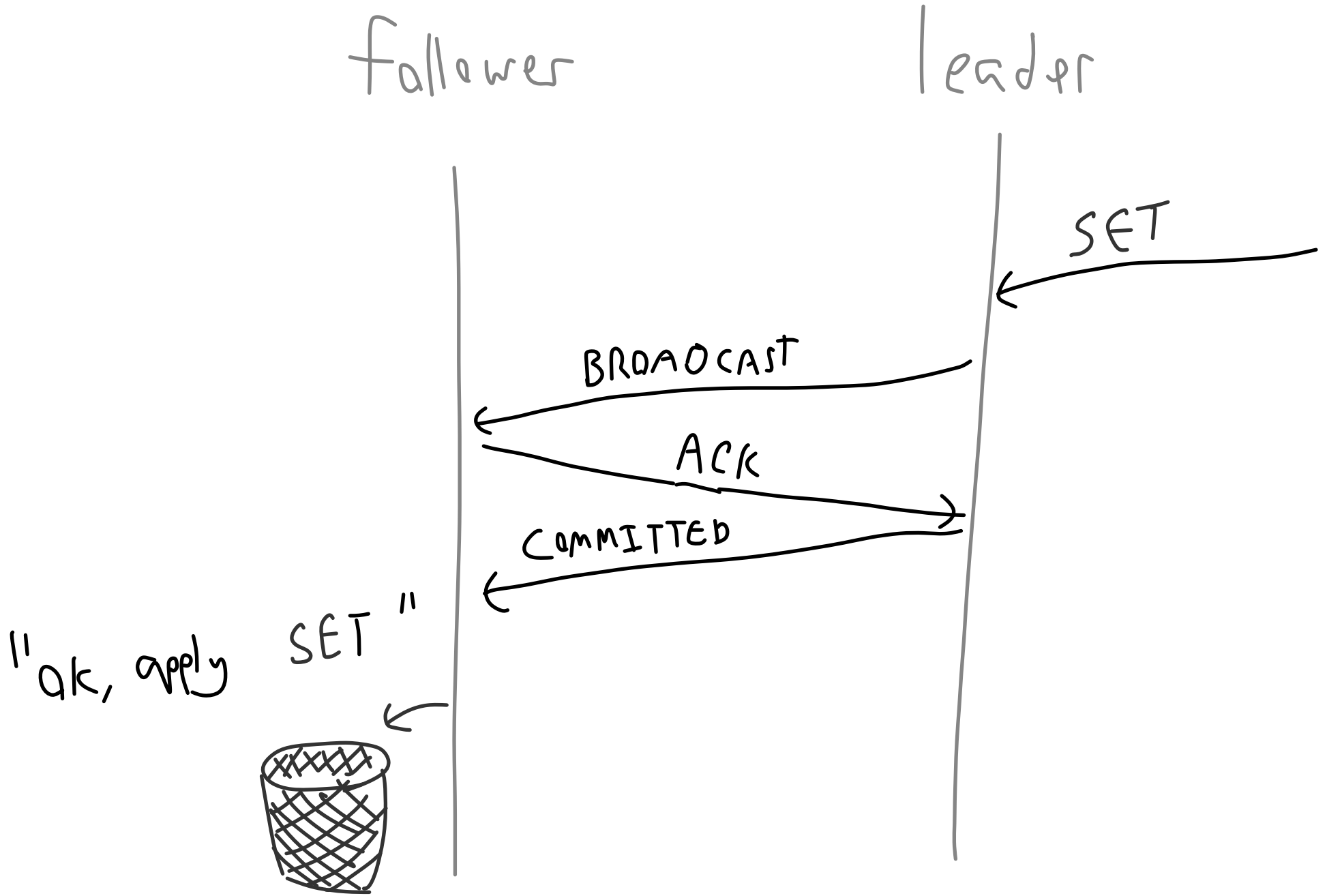
Fixed in d589127

via re-entrancy check

#14

Total Data Loss on Any Failover



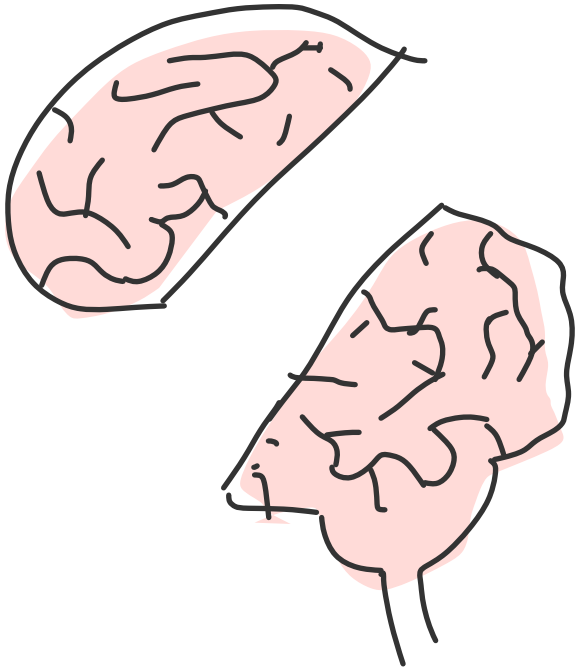

follower
leader
SET
BROADCAST
ACK
COMMITTED
"ok, apply SET"

Fixed in d589127

as well

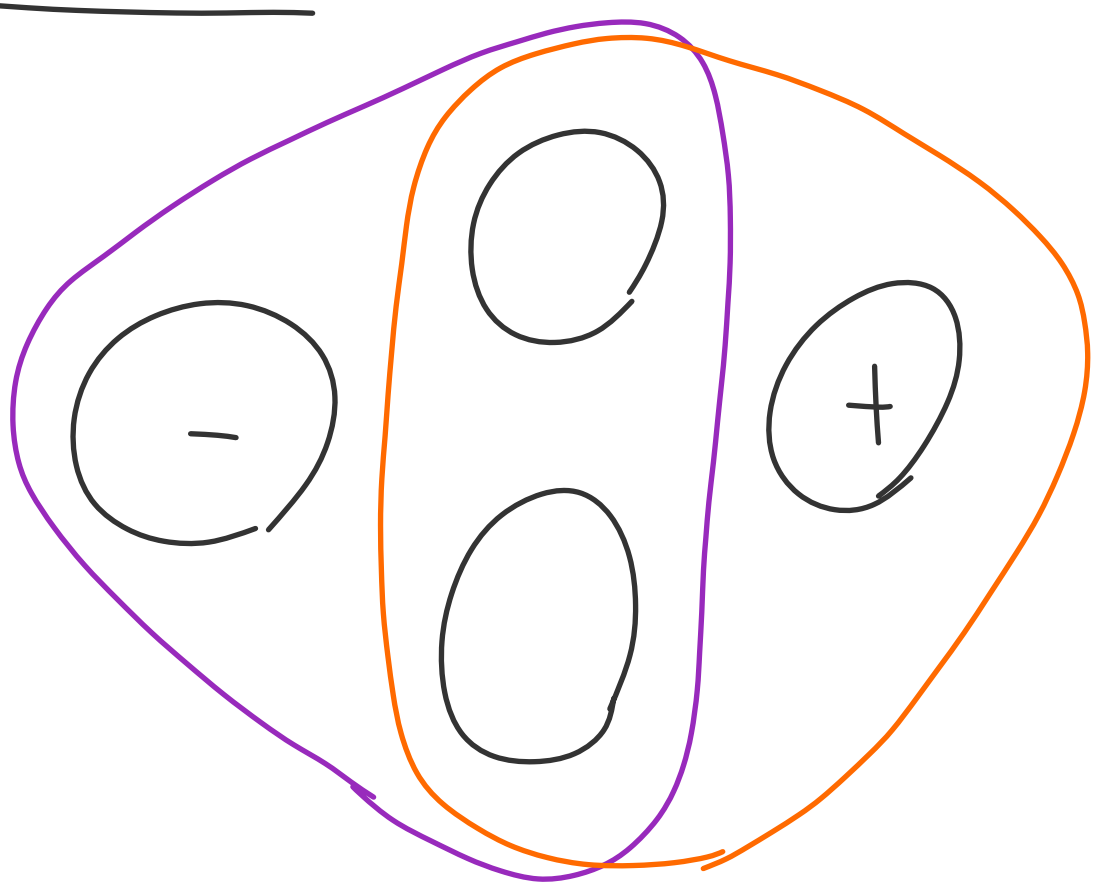
#17

Split - Brain



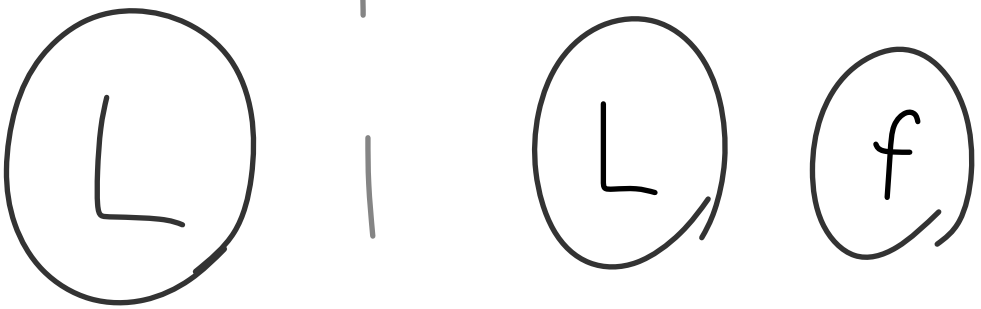
Last Updates

Joint Consensus

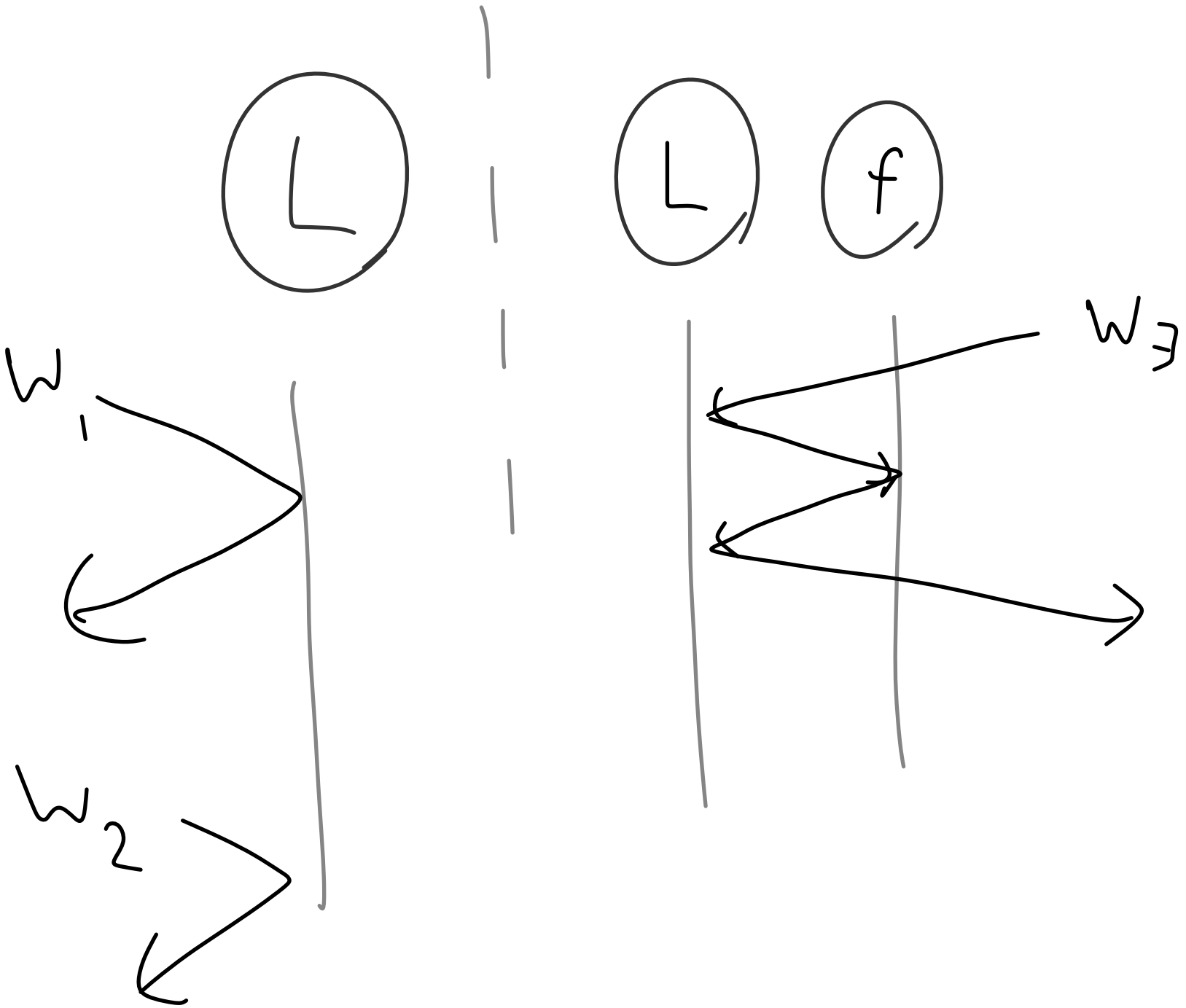


Old Cohort

New Cohort



"forget about
thave two...
yan don't need
their approval!"


L
L
f
W1
W2
W3

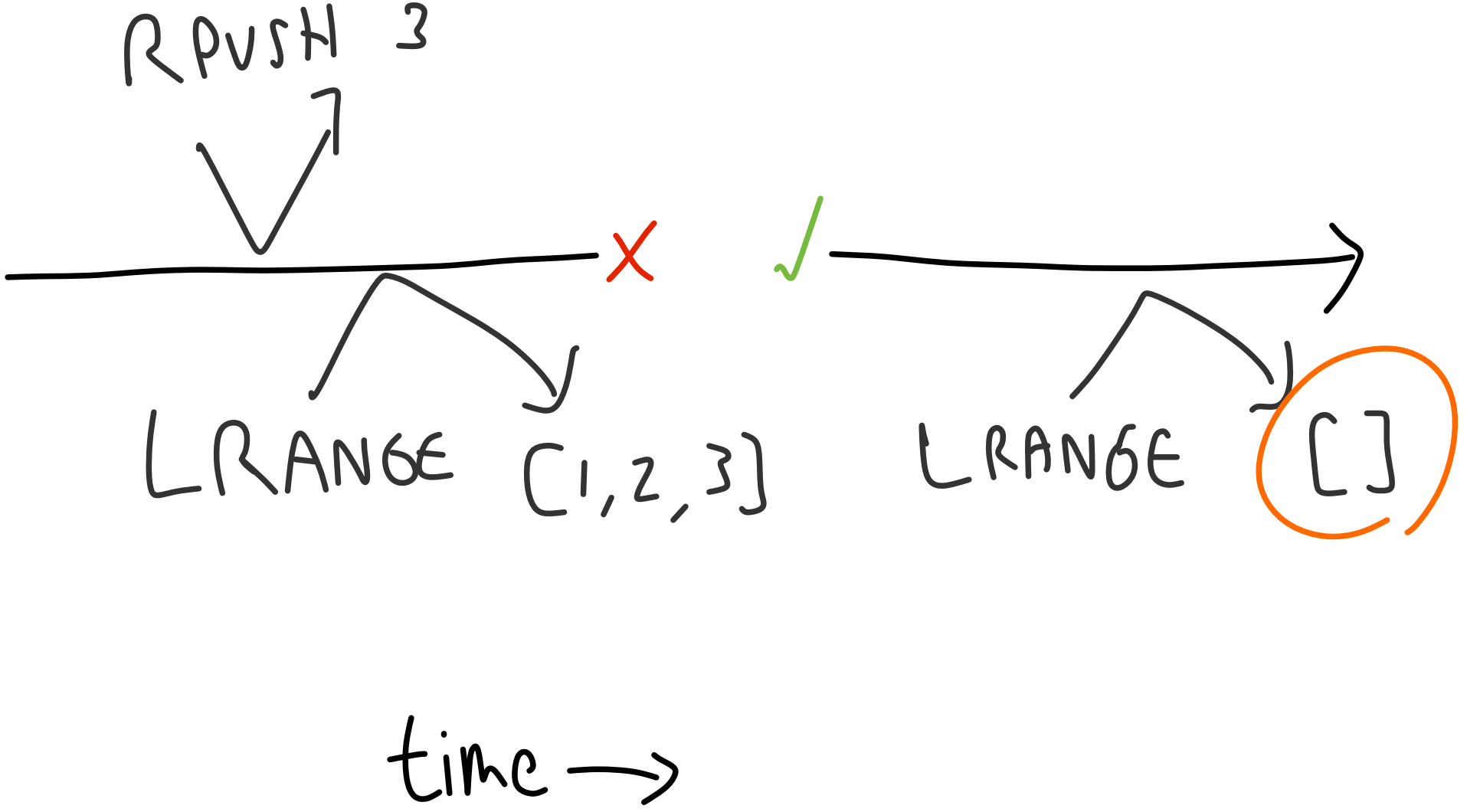
RAFT_LOGTYPE_REMOVE_NODE left out of set of msg types considered

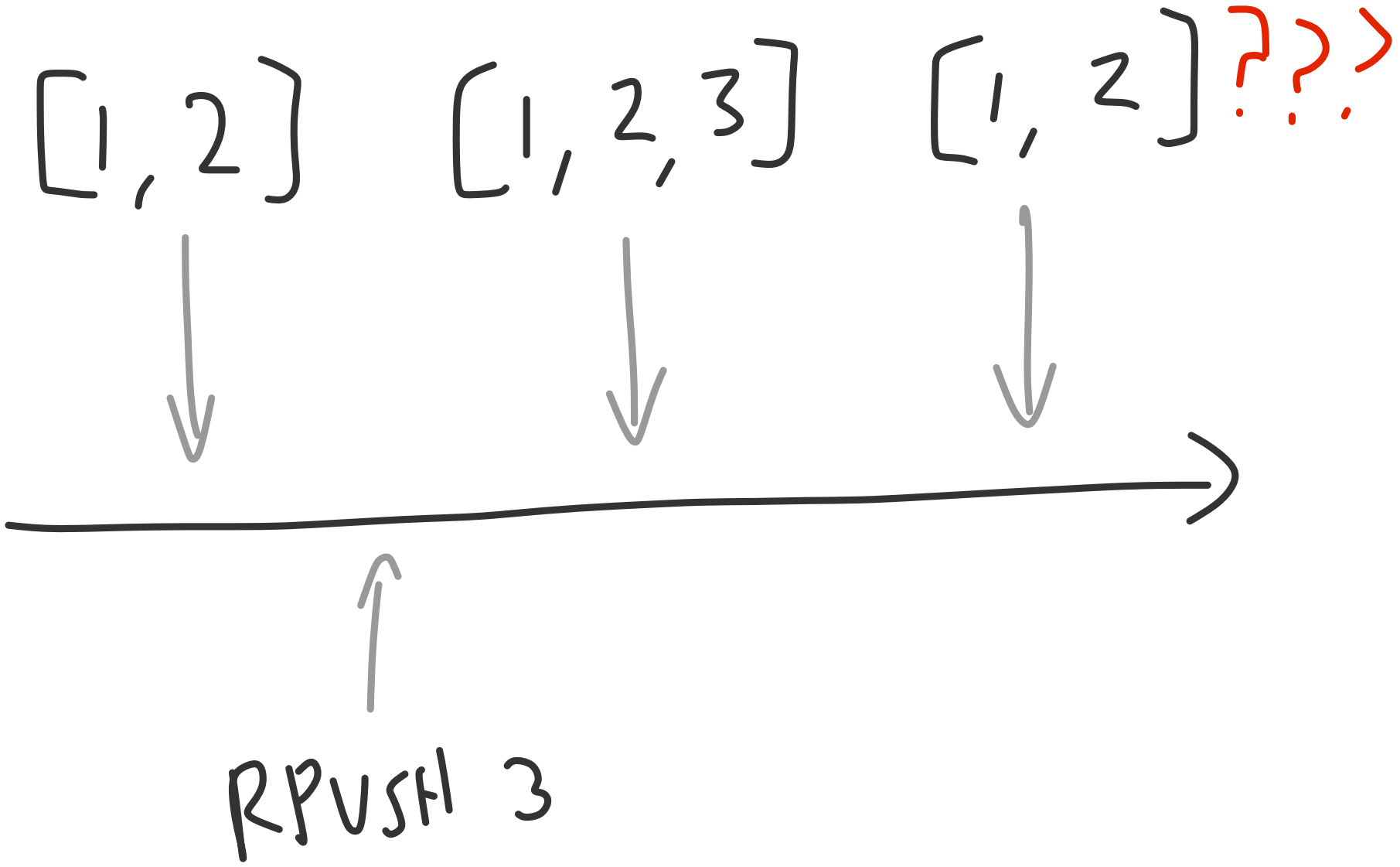
Voting config changes

Partial fix in 8da0c77, rest in #28

#18

Transient Stale Reads


RPUSH 3
X
✓
LRANGE
[1,2,3]
LRANGE
[]
time →

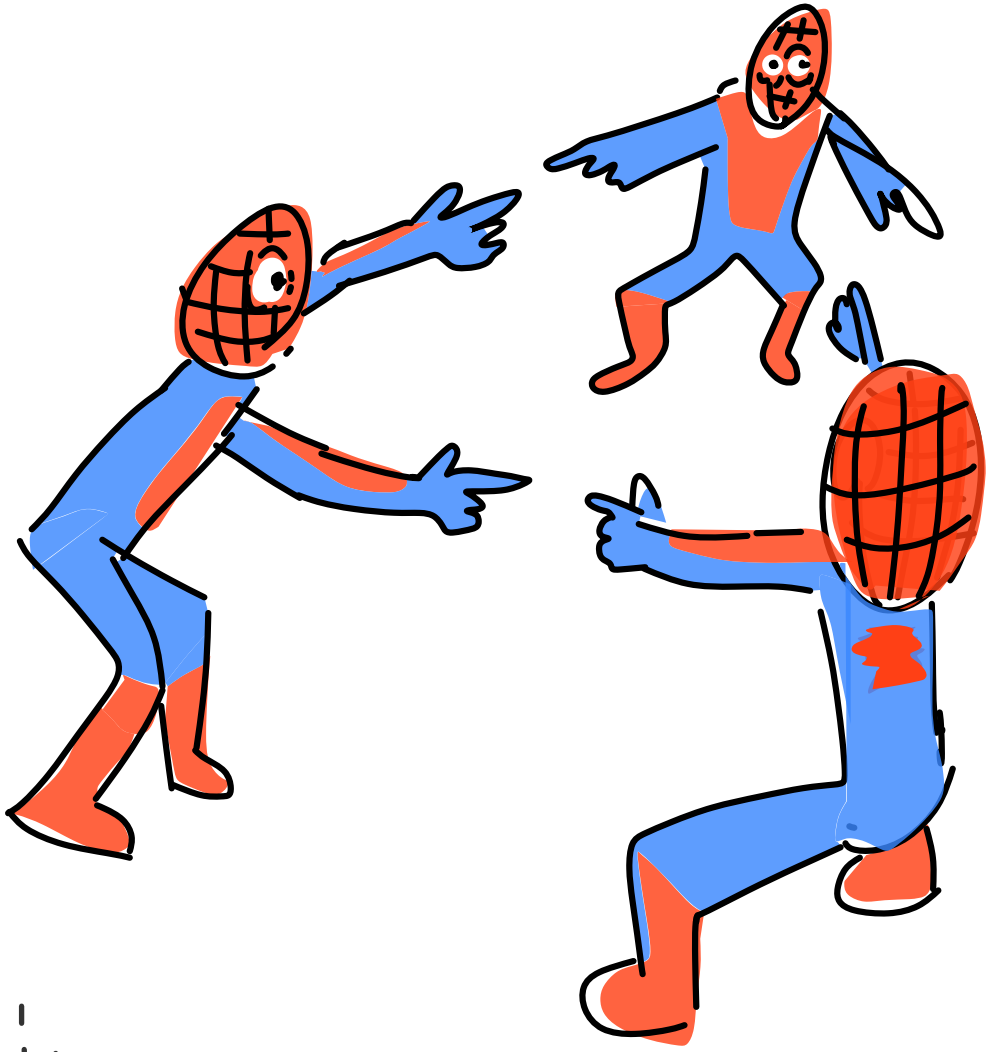

[1, 2]
[1, 2, 3]
[1, 2]???
RPUSH 3

Raft library failed to issue no-op log entries on coming to power — failed to establish commit point

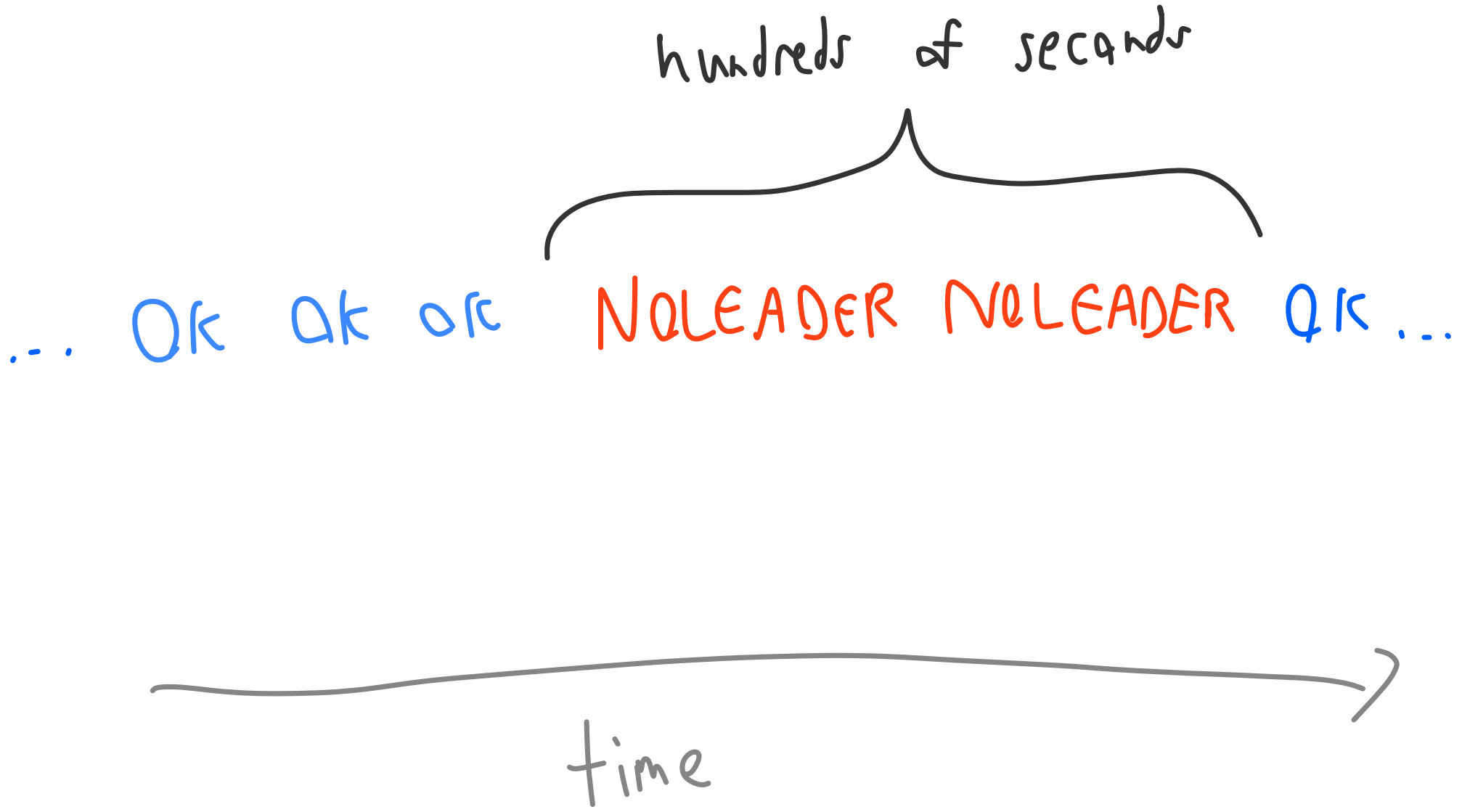
Fixed in dfd9104

(already addressed upstream)

#21



Spurious NOLEAPER in Healthy Clusters


hundreds of seconds
... OK OK OK NOLEADER NOLEADER OK ...
time

- Connection handling issues caused long stalls after re-election
- Aggressive default timeouts led to frequent elections

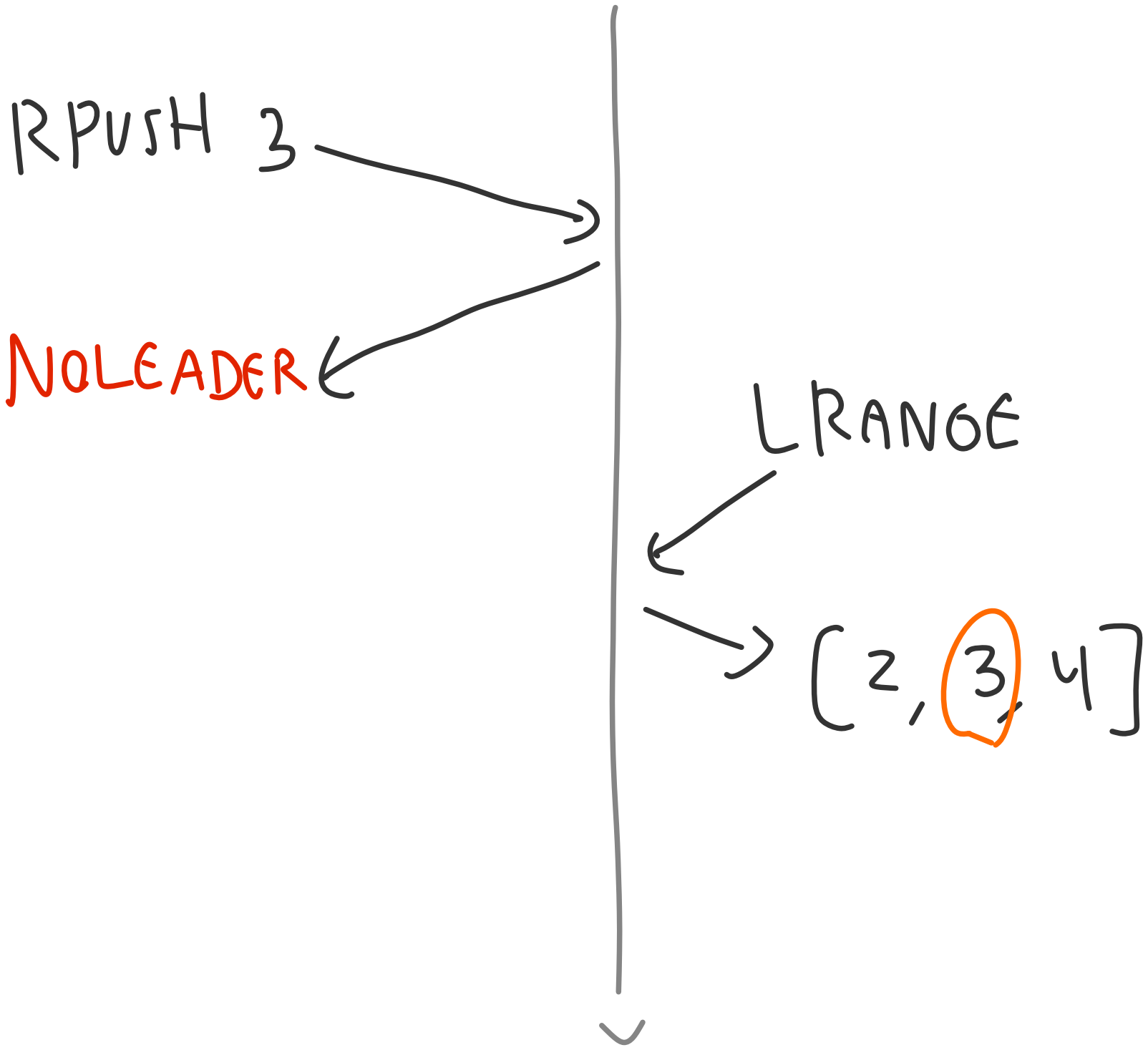
Fixed in recent builds

#23

Aborted Reads

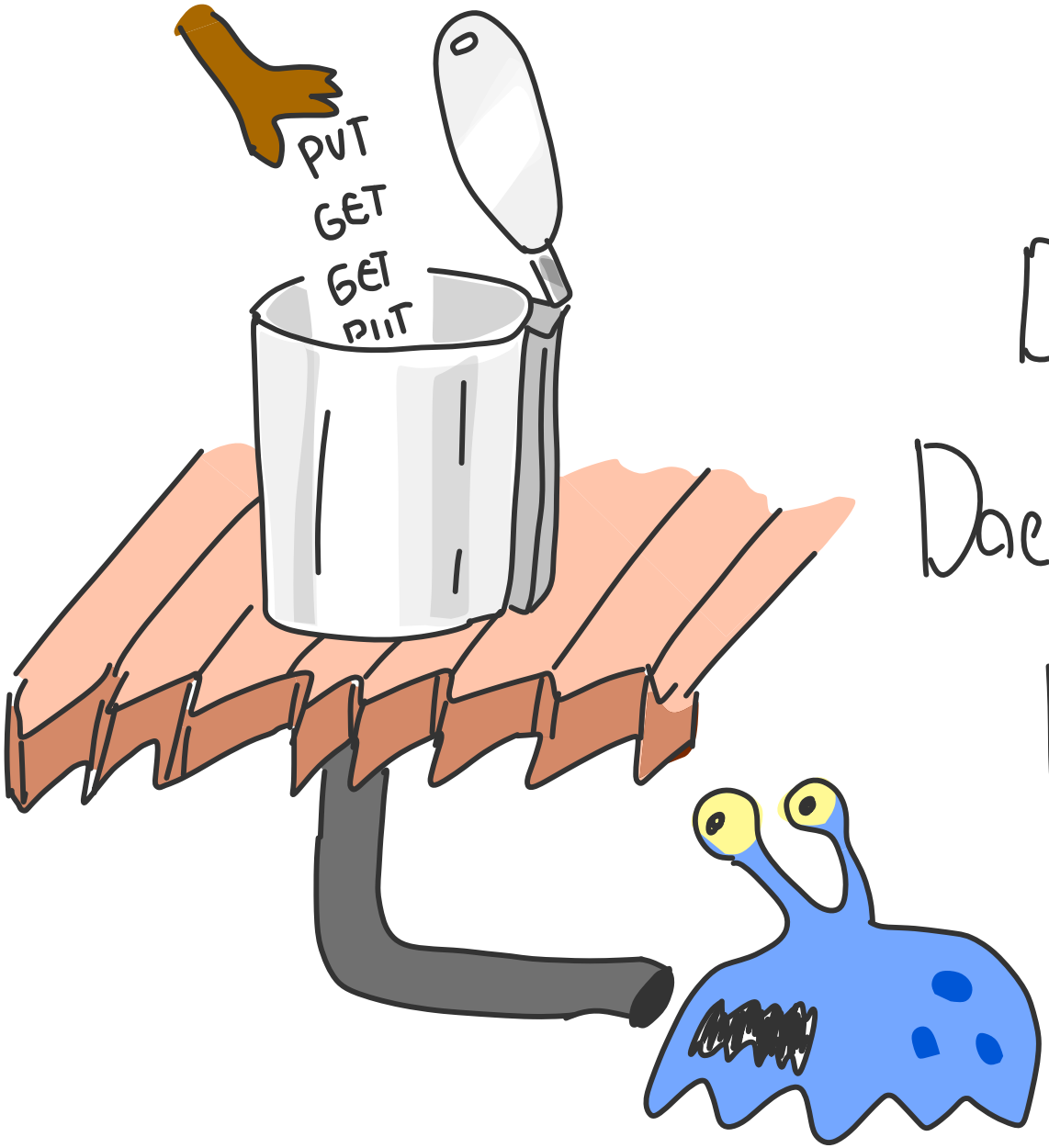
with

NoLEADER


RPUSH 3
NOLEADER
LRANGE
[2, 3, 4]

Addressed by a suite of protocol-level fixes in @657423

#25



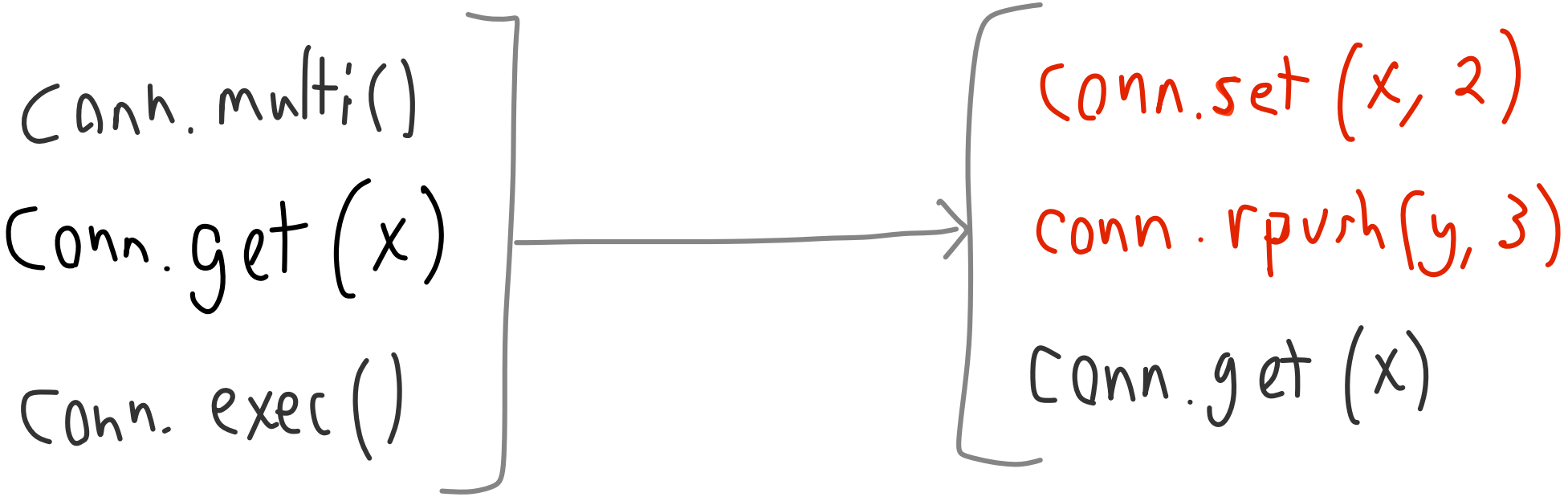
DISCARD

Daesn't Always

Discard

```
try {
    conn.multi()
    conn.set(x, 2)
    conn.rpush(y, 3)
    conn.exec()
}catch (Exception e) {
    conn.discard()
}
```

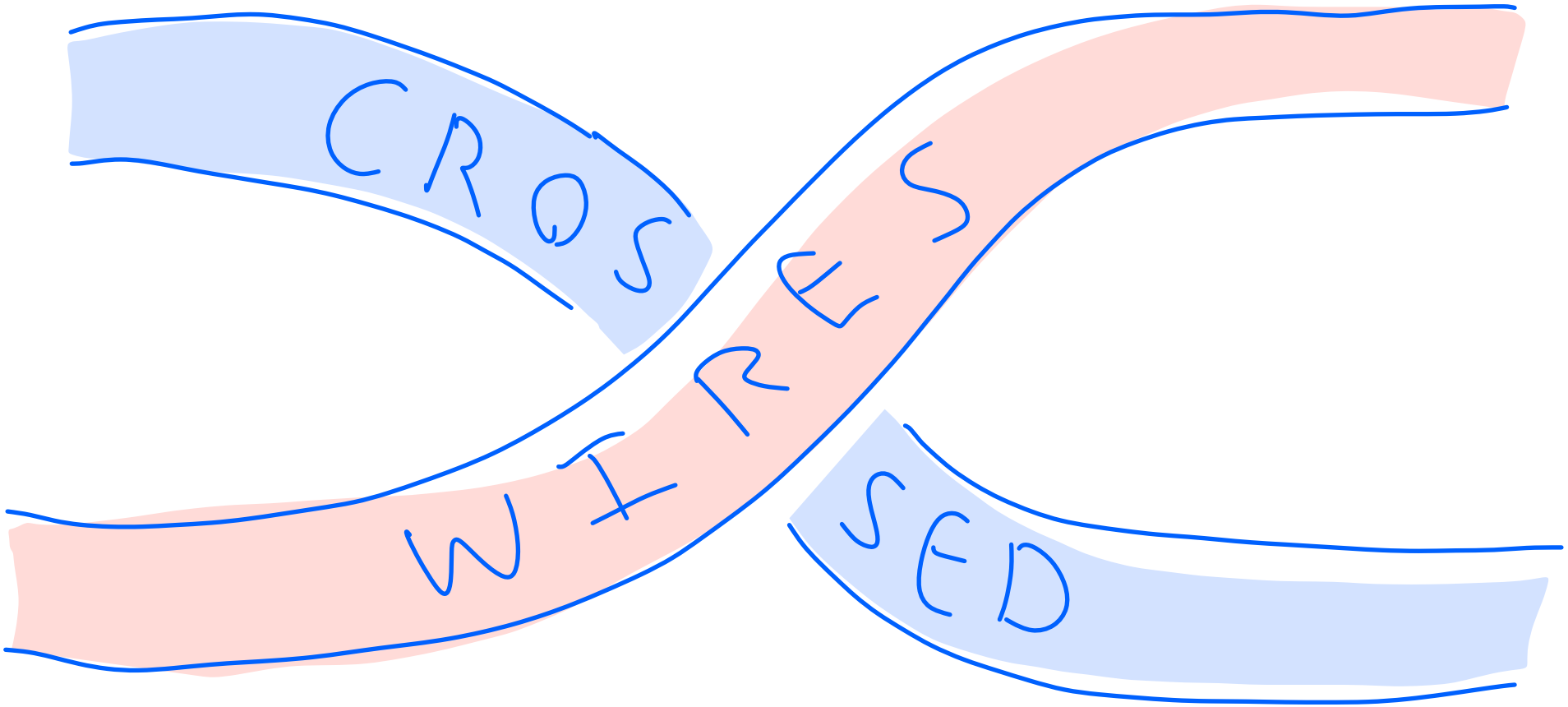
```
try {
    conn.multi()
    conn.set(x, 2)
    conn.rpush(y, 3)
    conn.exec()
} catch (Exception e) {
    conn.discard()
}
```


Conn.multi()
Conn.get(x)
Conn.exec()
Conn.set(x, 2)
Conn.rpush(y, 3)
Conn.get(x)

DISCARD went through Raft log

- Could fail!
- Leaving Conn in MULTI state
- Fixed in f4bb49f
- Now a local state machine

#26



Network partitions + crashes caused Redis-Raft to send responses to wrong clients

⇒ Type errors

⇒ Silent data corruption

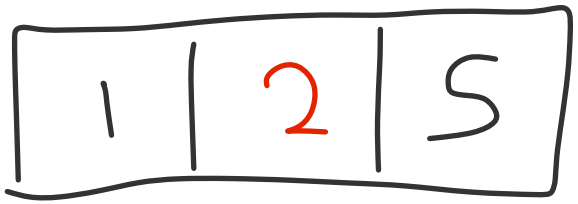
- Replies for proxied commands used Redis-Raft ctx, not request ctx
- Leaders re-bundled MULTI txns
- Error handling around async context cleanups

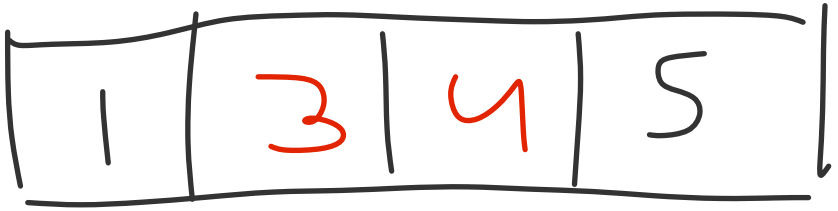
Fixed by e657423

28

More Split-Brain & Lost Updates

Membership changes w/ crashes could create weird split-brain states





Bug in Raft Library!

- Nodes were supposed to be demoted THEN removed
- Node left running, w/ data files, after removal

Fixed in bc9552f

#30

More Transient Empty Reads

Nodes which restart after membership change temporarily exhibited empty reads

Very rare!

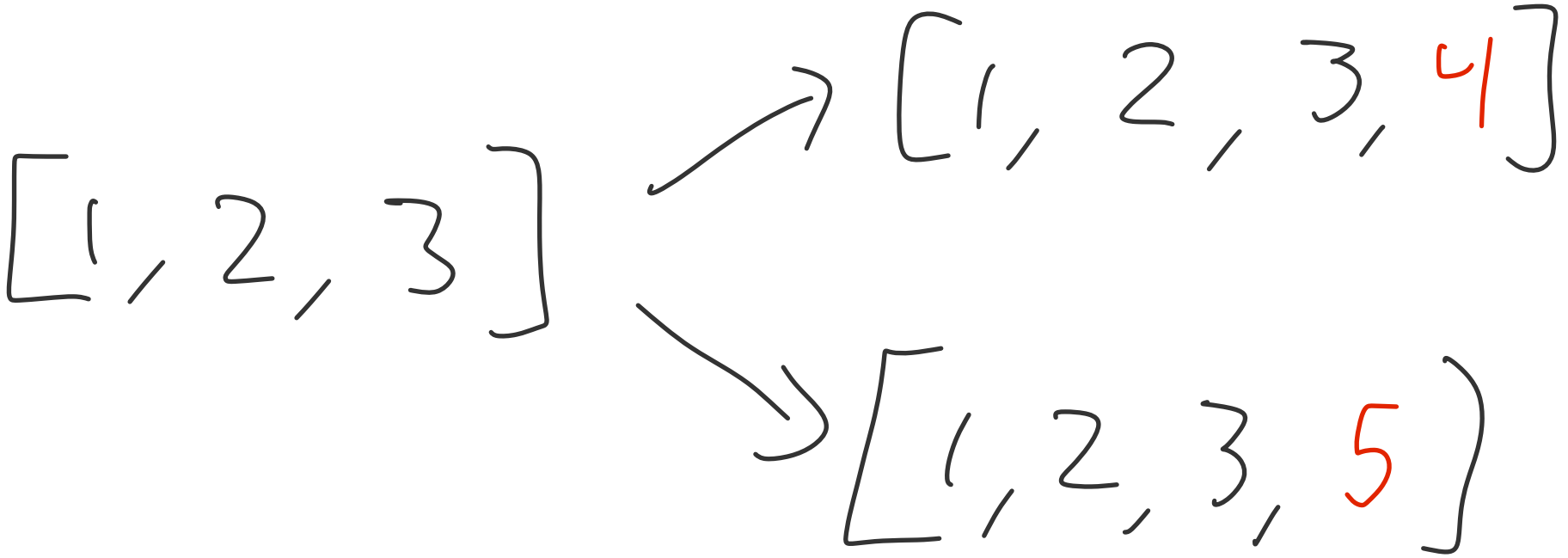
Bugs in log loading process

1. Used all nodes, not voting nodes, to determine log entry committed state
2. Could treat former cluster members as if they were active nodes!

Fixed in 73ad833

Even Marc

Split Brain

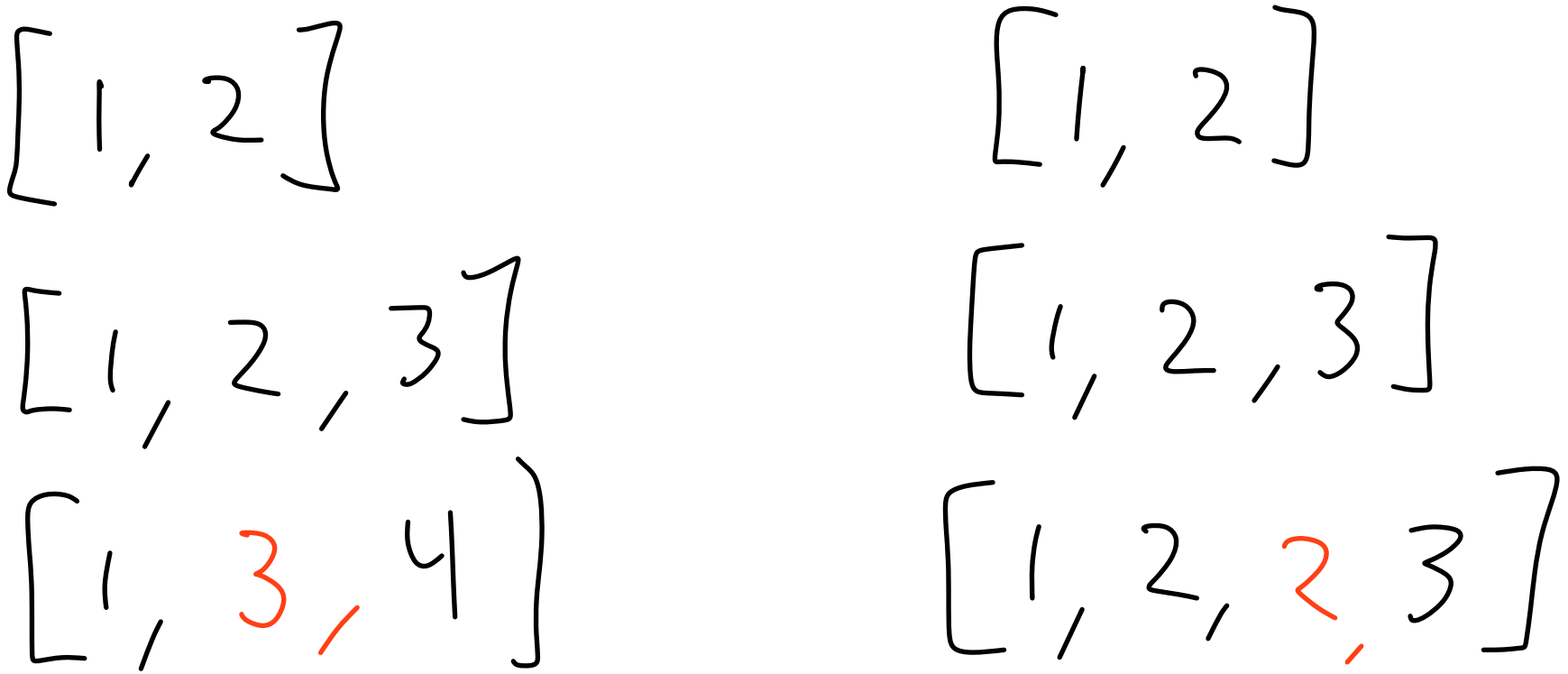

[1, 2, 3]
[1, 2, 3, 4]
[1, 2, 3, 5]

Due to a bug in the snapshot loading process — fine in memory, but next snapshot would be missing nodes!

Fixed in 2d1cf30

#52

Dueling Histories


[1, 2]
[1, 2, 3]
[1, 3, 4]
[1, 2]
[1, 2, 3]
[1, 2, 2, 3]

Happens when we

kill processes

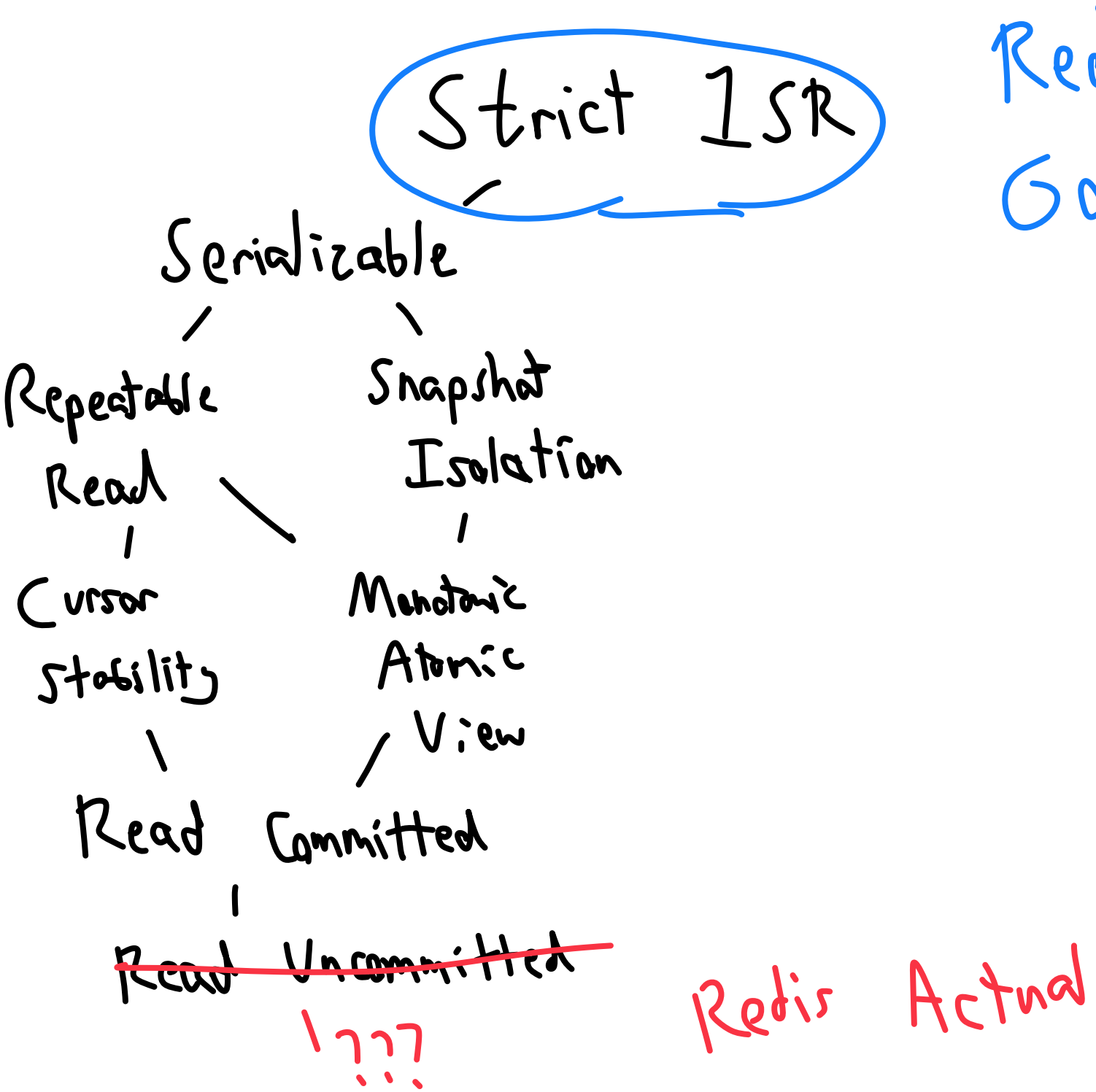
fixed in

e0123a9


fine
Assorted
Crashes

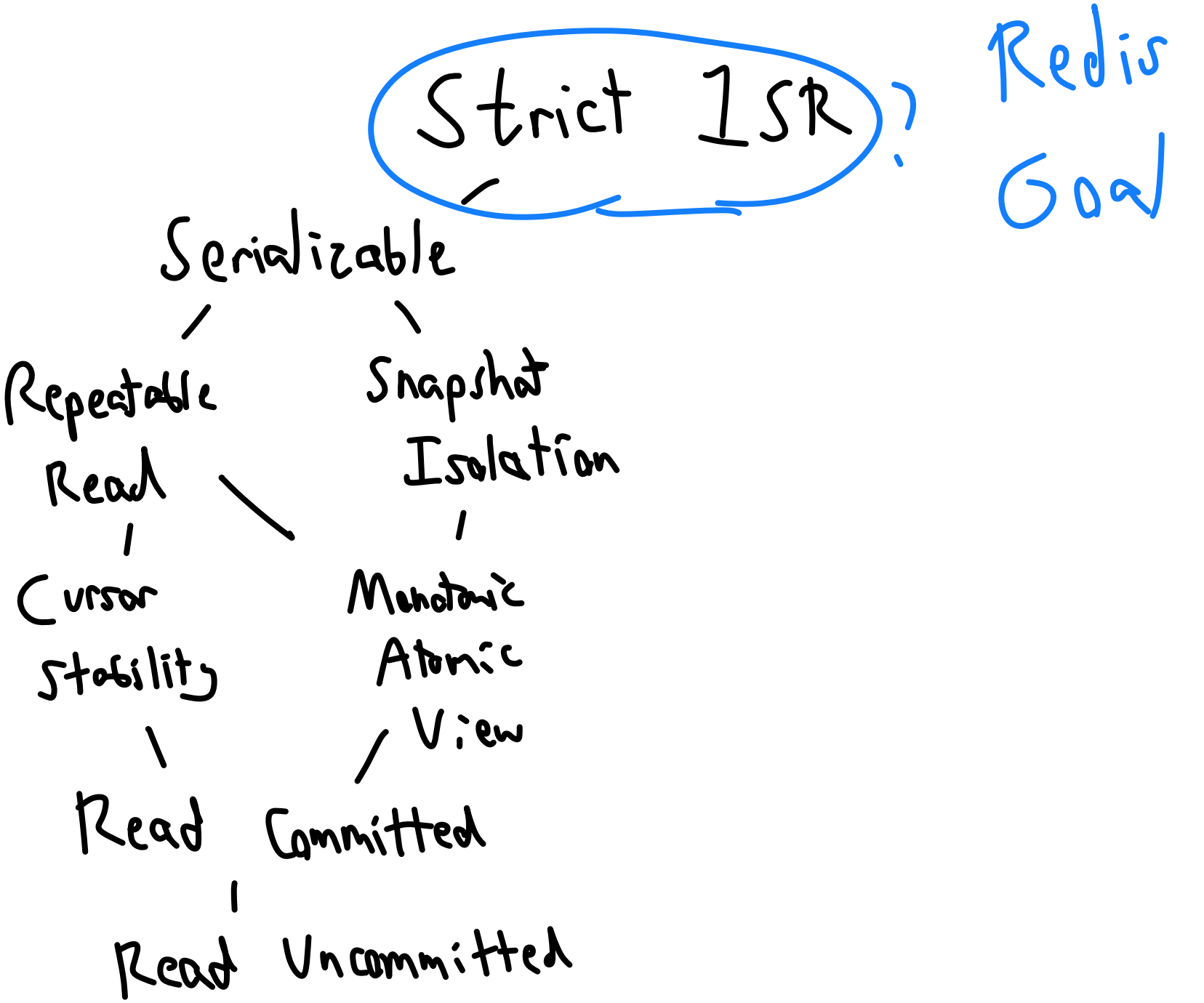
- Mix of segfaults & asserts
- Some point to Raft invariant violations
- Most not associated w/ observable safety issues

All fixed!


Strict 1SR
Redis Goal
Serializable
Repeatable Read
Snapshot Isolation
Cursor Stability
Monotonic Atomic View
Read Committed
~~Read Uncommitted~~
!???
Redis Actual

All of these were in early dev builds

→ No users to affect!


Strict 1SR?
Redis Goal
Serializable
Repeatable Read
Snapshot Isolation
Cursor Stability
Monotonic Atomic View
Read Committed
Read Uncommitted

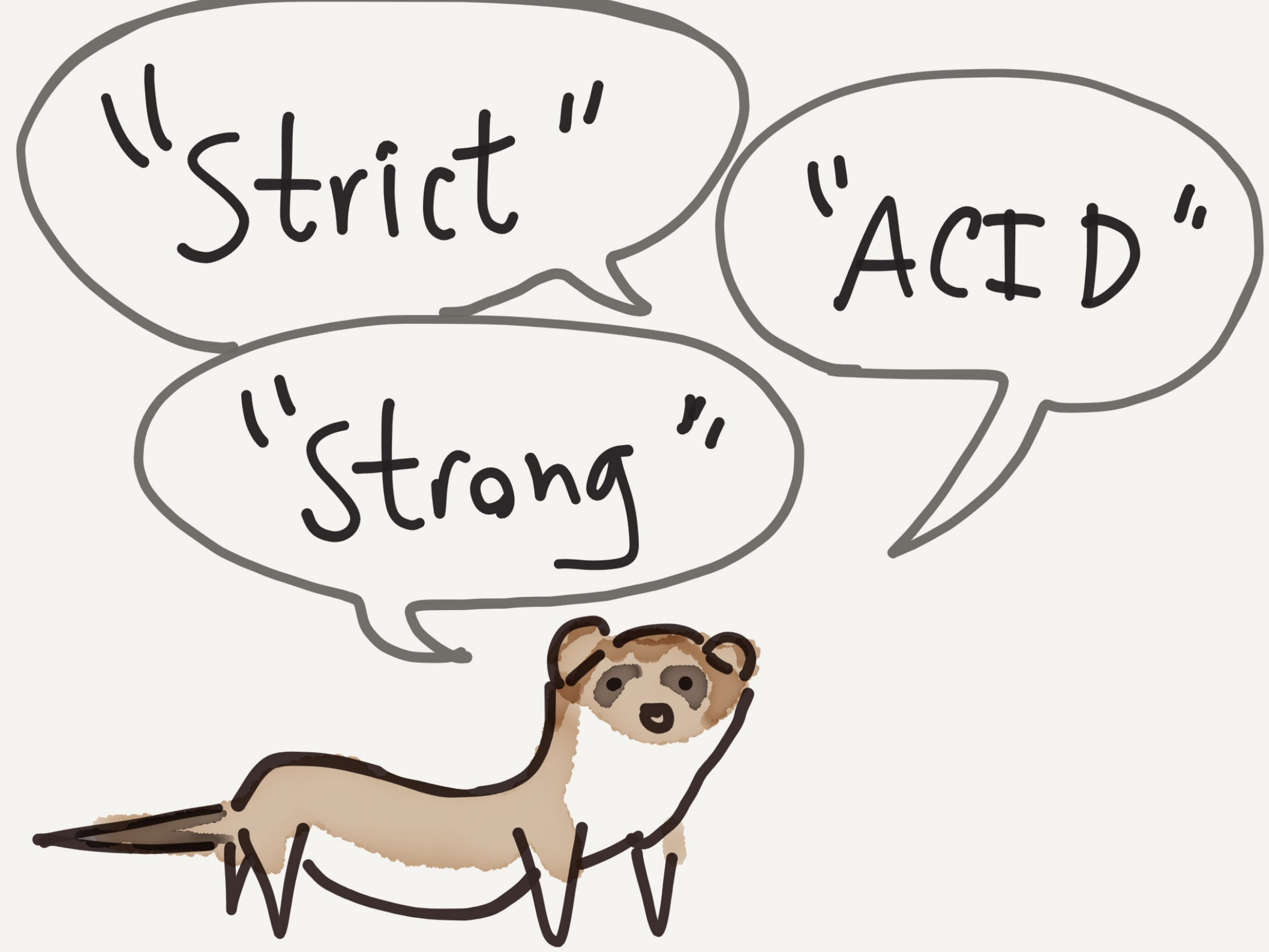
Recap


Read the docs!
CAREFULLY

Then test it

— for —

YOURSELF


"Strict"
"ACID"
"Strong"

Be Formal

&

Be Specific!

Figure out the invariants your system needs

USE
proven
algorithms

PROOFS

aren't

ENOUGH

Consider your failure modes

Process Crash

```
#kill -9 1234
```

Node failure

- AWS terminate
- Physical power switch

Clock Skew

```
# date 10280000
# faketime ...
```

GC/IO Pause

```
# killall -s STOP foo
# killall -s CONT foo
```

Network Partition

iptables -j DROP

tc qdisc ... delay ...

drop ...

- Property testing
- High-level invariants
- With distsys failure modes

If you look for

perfection

you will never be

content

Thanks

- Peter Alvaro
- Kit Patella
- Andres Freund
- Peter Geoghegan
- Félix Gerzaguet

Thanks

- Thomas Munroe
- Daniel Verite
- Maxime Bengnet
- Redis Labs $
- Yiftach Shoolman

Thanks

-Yossi Gottlieb

https://jepsen.io