Programming at Fast Scale: Consistency + Lock Freedom

cs378

Today

Questions?

Administrivia

- Project Proposal Due Today!

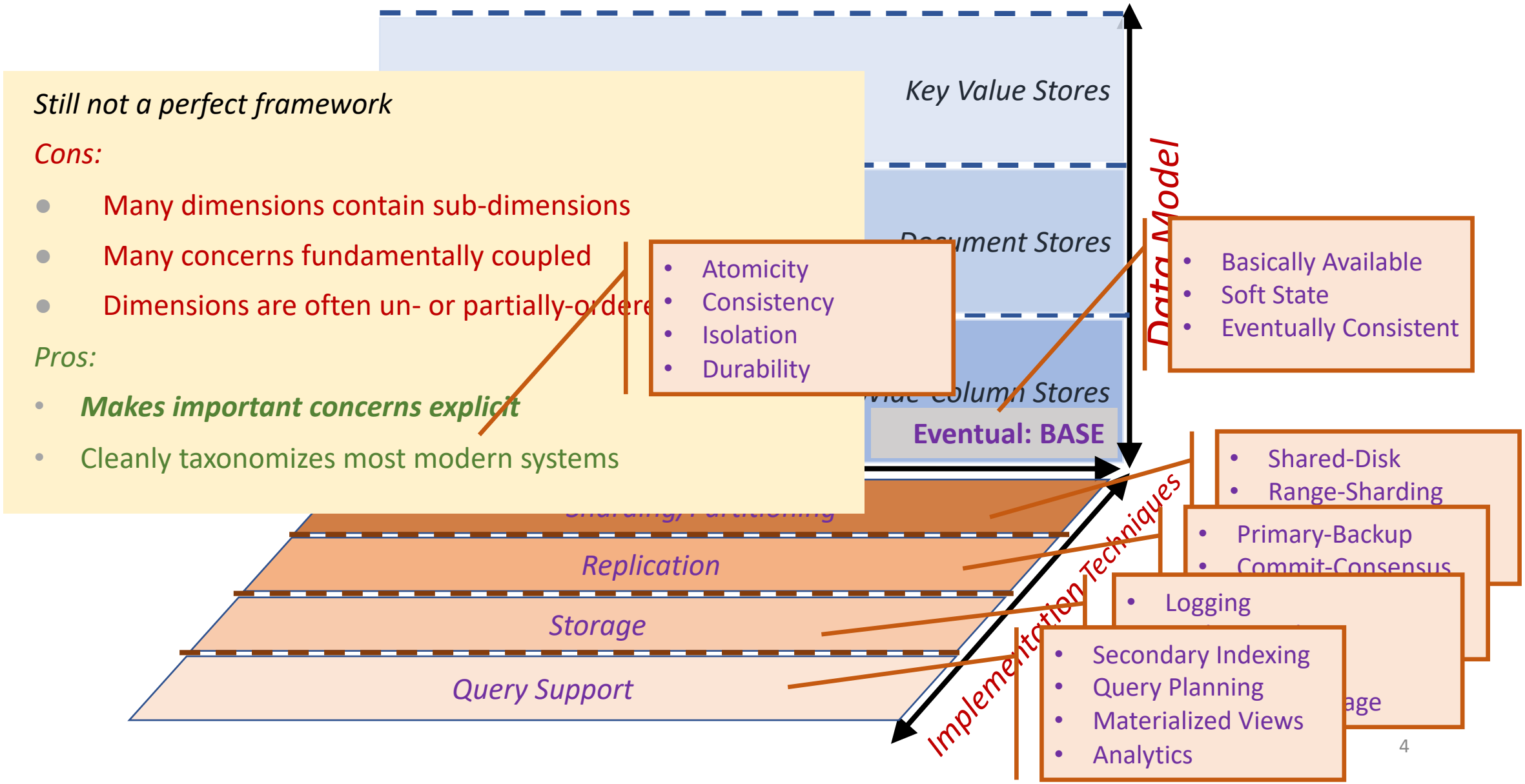
Agenda:

- Consistency
- Lock Freedom

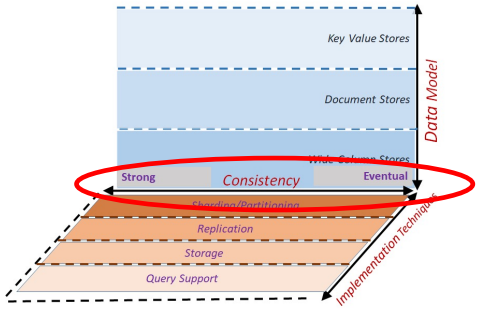
Faux Quiz Questions

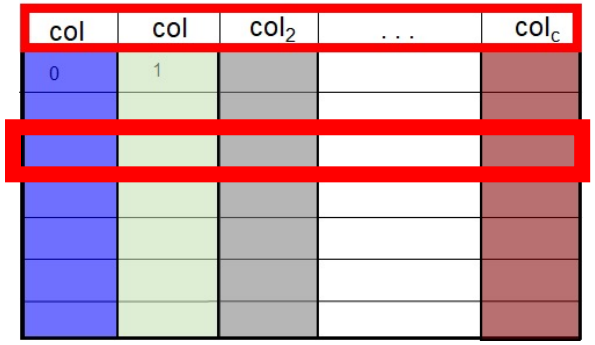
- What is the CAP theorem? What does “PACELP” stand for and how does it relate to CAP?
- What is the difference between ACID and BASE?
- Why do NoSQL systems claim to be more horizontally scalable than RDMBSes? List some features NoSQL systems give up toward this goal?
- What is eventual consistency? Give a concrete example of how of why it causes a complex programming model (relative to a strongly consistent model).
- Compare and contrast Key-Value, Document, and Wide-column Stores
- Define and contrast the following consistency properties:
  - strong consistency, eventual consistency, consistent prefix, monotonic reads, read-my-writes, bounded staleness
- What is causal consistency?
- What is chain replication?
- What is obstruction freedom, wait freedom, lock freedom?
- How can one compose lock free data structures?
- Why should I want a lock free hash table instead of a fine-grain lock-based one?
- What is the difference between linearizability and strong consistency? Between linearizability and serializability?
- What is the ABA problem? Give an example.
- How do lock-free data structures deal with the “inconsistent view” problem?

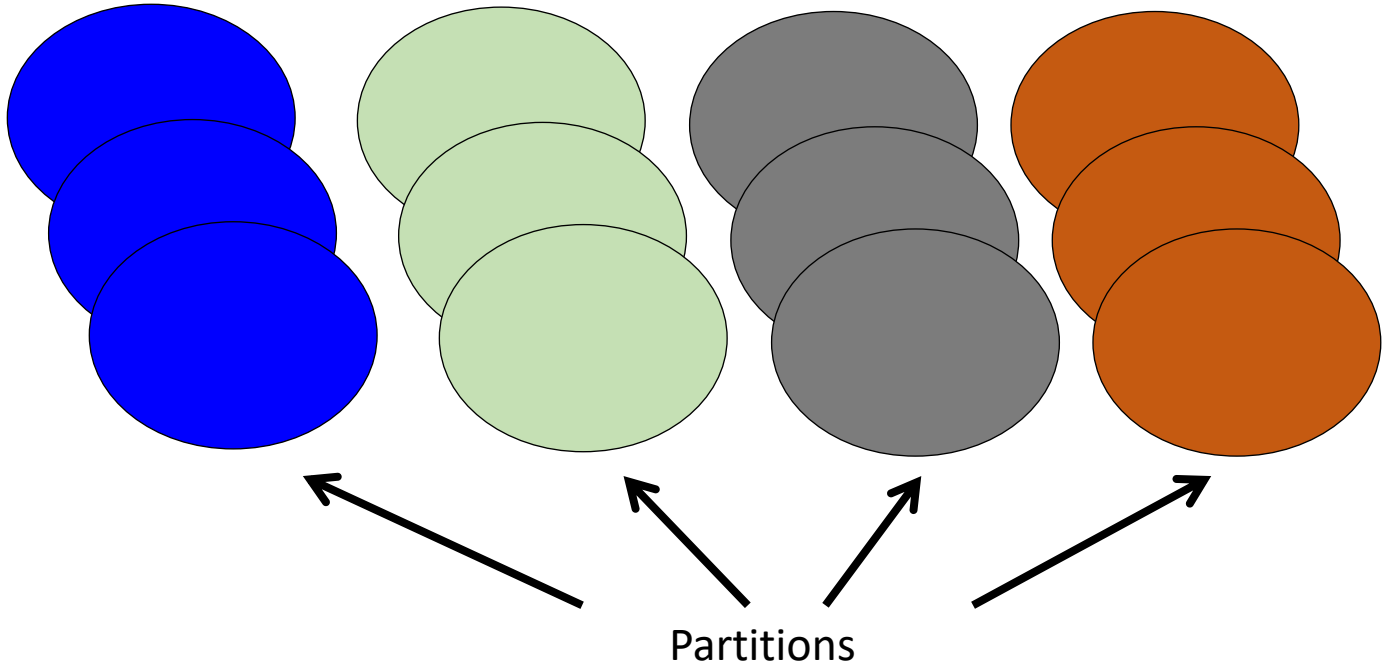
Another Framework



Consistency



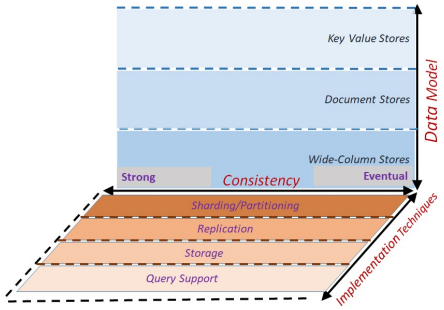


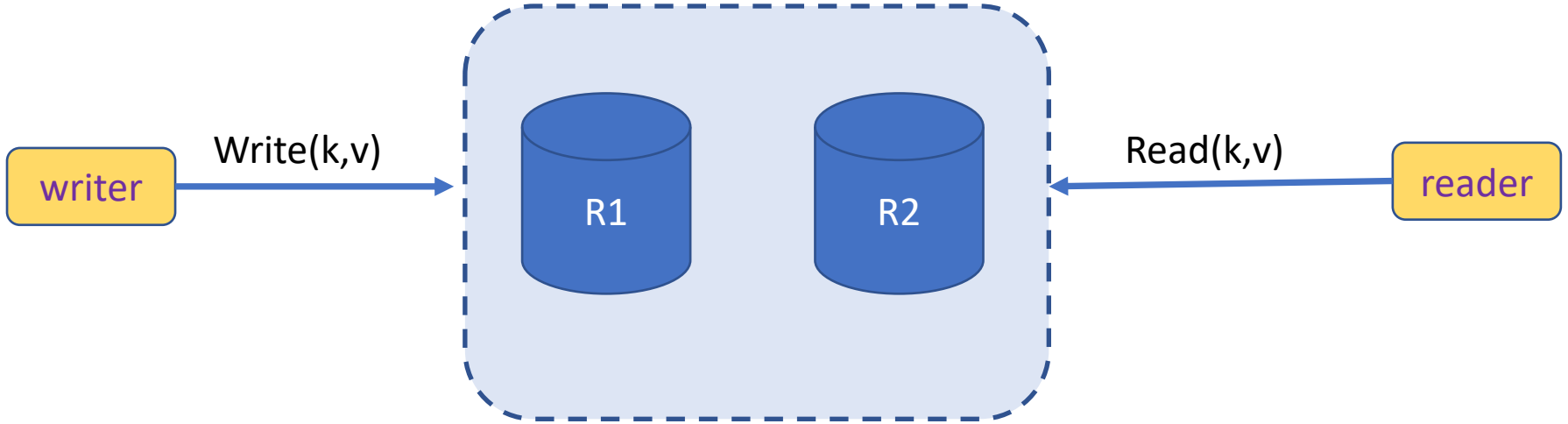


How to keep data in sync?

- Partitioning → single row spread over multiple machines
- Redundancy → single datum spread over multiple machines

Consistency: the core problem

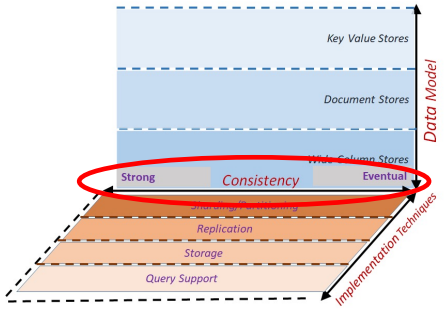




- Clients perform reads and writes
- Data is replicated among a set of servers
- Writes must be performed at all servers
- Reads return the result of one or more past writes

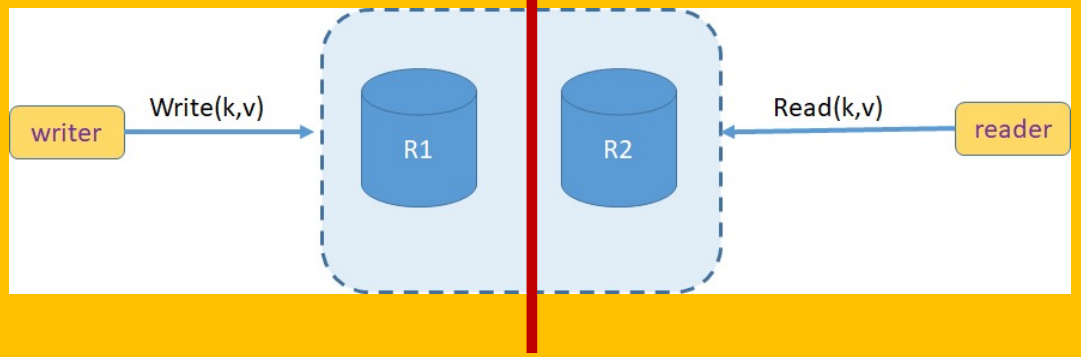
- How should we *implement* write?
- How to *implement* read?

Consistency: CAP Theorem



- A distributed system can satisfy at most 2/3 guarantees of:
  1. **Consistency**:
     - all nodes see same data at any time
     - or reads return latest written value by any client
  2. **Availability**:
     - system allows operations all the time,
     - and operations return quickly
  3. **Partition-tolerance**:
     - system continues to work in spite of netwo

Why is this "theorem" true?

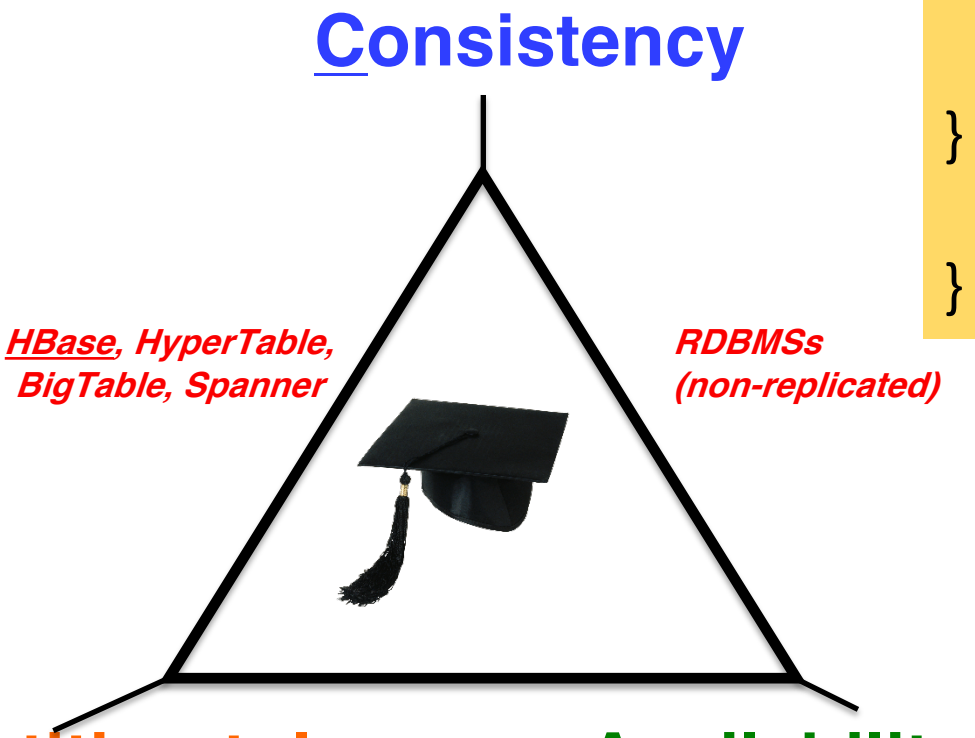


if(partition) { keep going } → !consistent && available

if(partition) { stop } → consistent && !available

CAP Implications

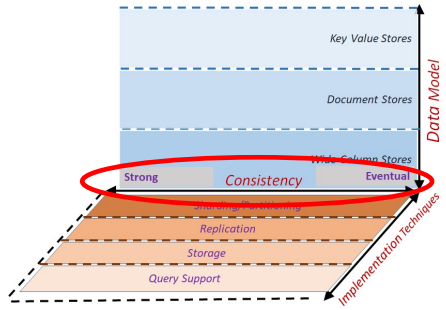
- A distributed storage system can achieve at most two of C, A, and P.
- When partition-tolerance is important, you have to choose between consistency and availability

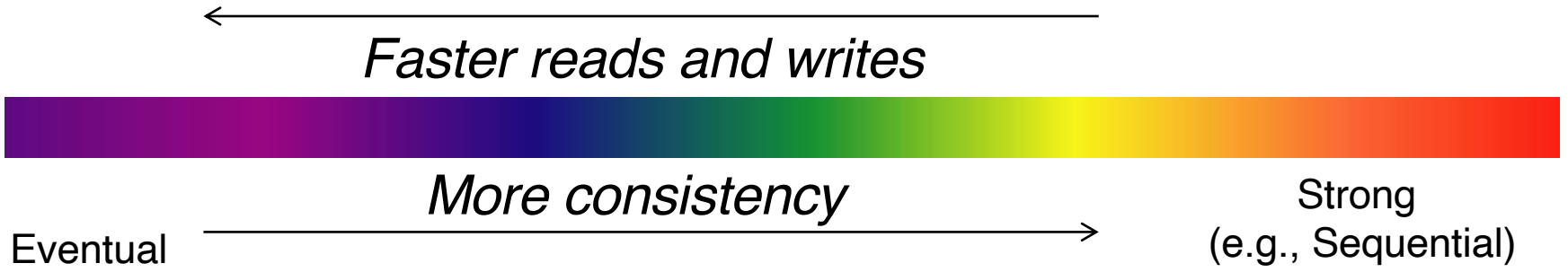


PACELC:

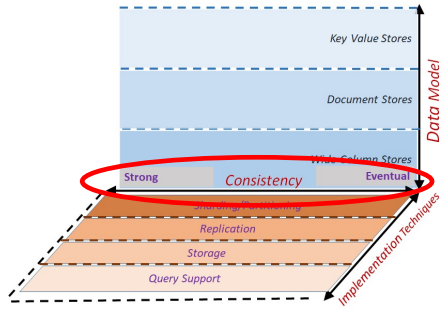
```
if(partition) {
   choose A or C
} else {
   choose latency or consistency
}
```

Consistency Spectrum

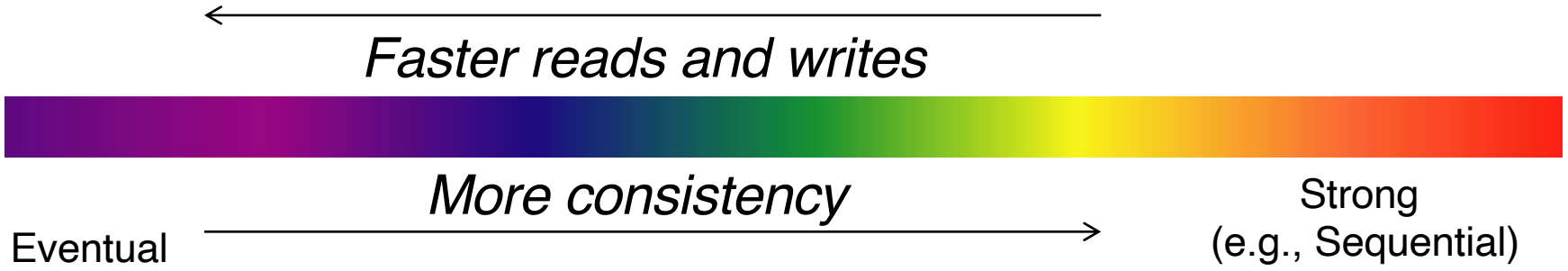




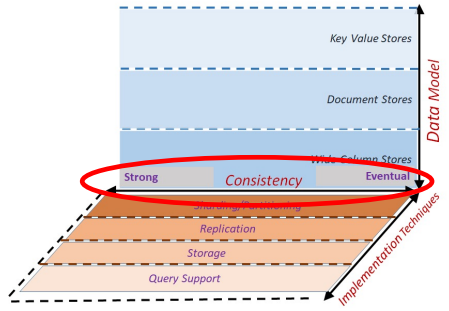
Spectrum Ends: Eventual Consistency



- Eventual Consistency
  - If writes to a key stop, all replicas of key will converge
  - Originally from Amazon's Dynamo and LinkedIn's Voldemort systems

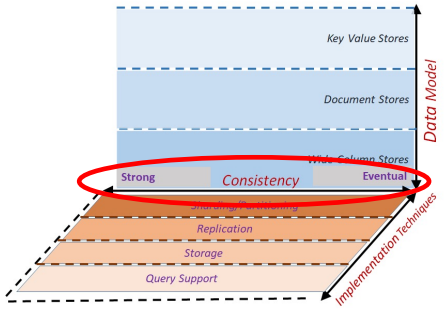


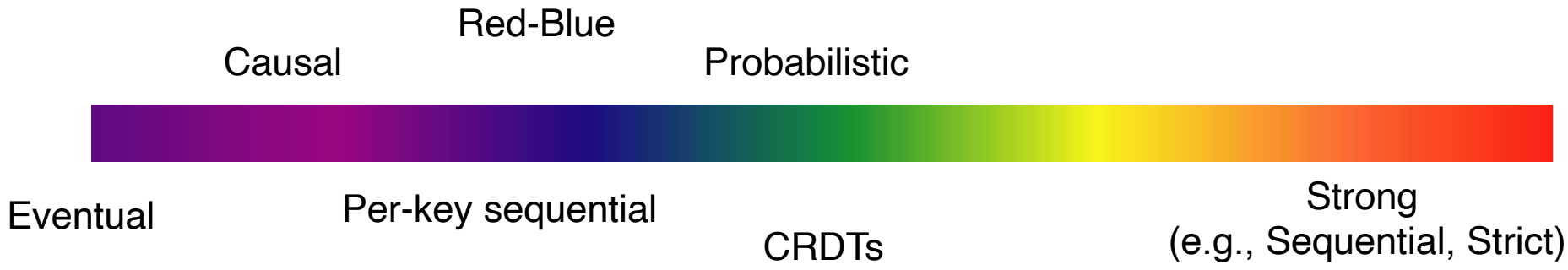
Spectrum Ends: Strong Consistency



- **Strict:**
  - Absolute time ordering of all shared accesses, reads always return last write
- **Linearizability**:
  - Each operation is visible (or available) to all other clients in real-time order
- **Sequential Consistency** [Lamport]:
  - *"... the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*
  - After the fact, find a "reasonable" ordering of the operations (can re-order operations) that obeys sanity (consistency) at all clients, and across clients.
- **ACID** properties

Many *Many* Consistency Models



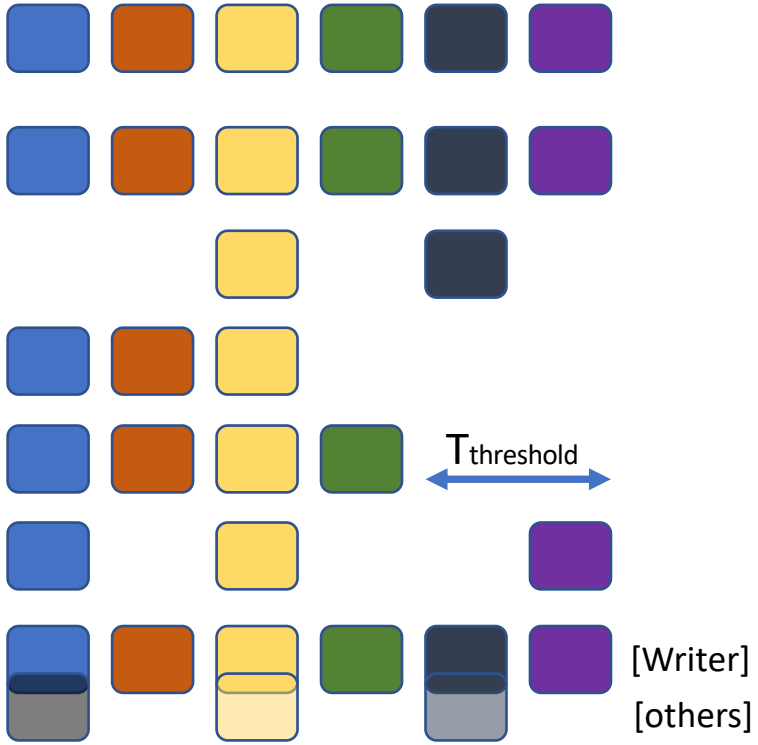


- Amazon S3 – **eventual** consistency
- Amazon Simple DB – **eventual** or strong
- Google App Engine – **strong** or eventual
- Yahoo! PNUTS – **eventual** or strong
- Windows Azure Storage – **strong** (or eventual)
- Cassandra – **eventual** or strong (if R+W > N)
- …

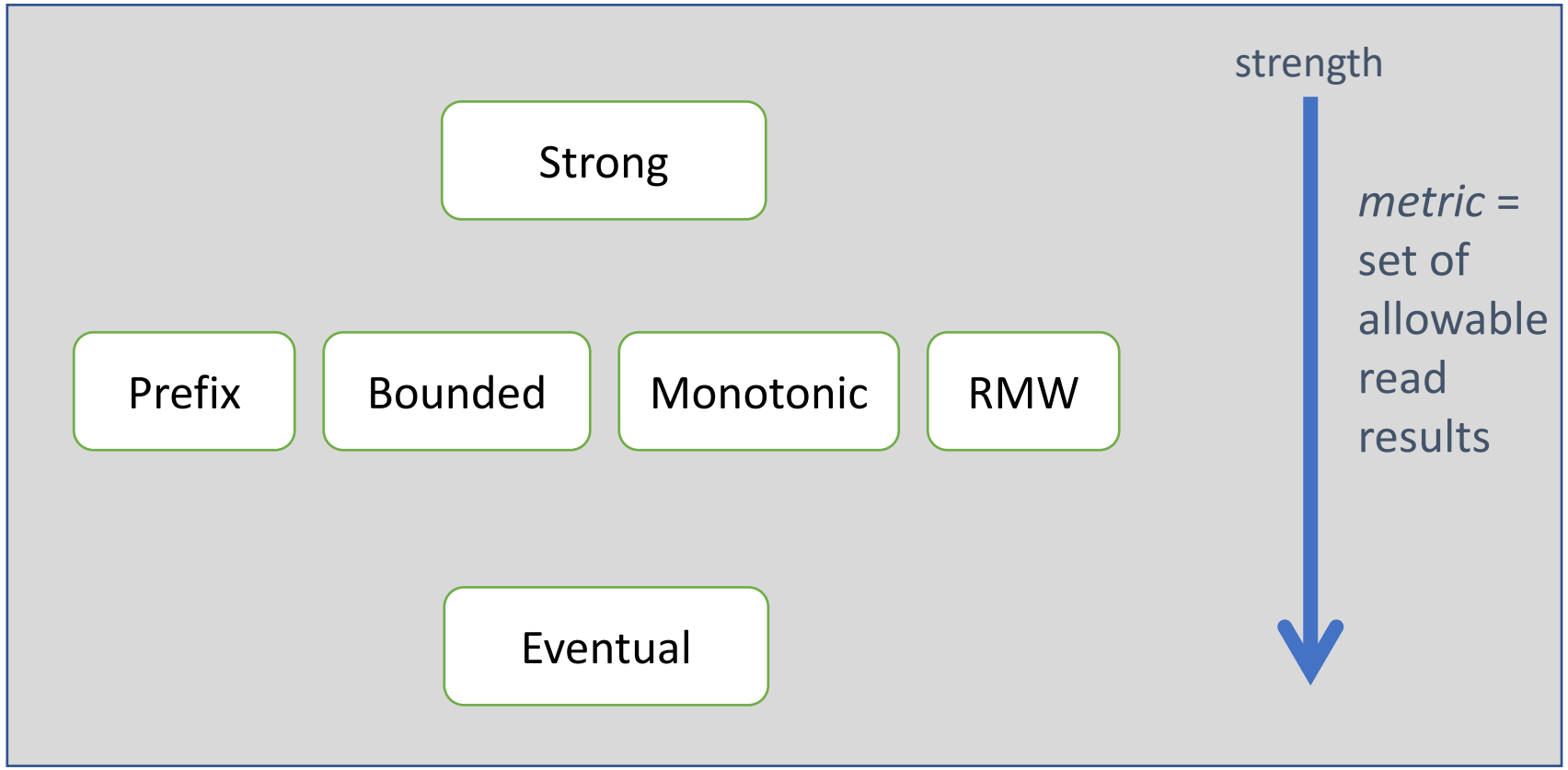
Question: How to choose what to use or support?

Some Consistency Guarantees

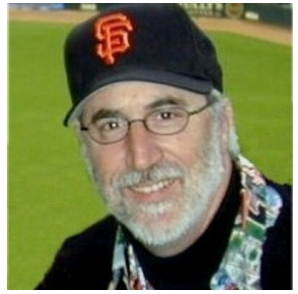
Strong Consistency	See all previous writes.
Eventual Consistency	See subset of previous writes.
Consistent Prefix	See initial sequence of writes.
Bounded Staleness	See all “old” writes.
Monotonic Reads	See increasing subset of writes.
Read My Writes	See all writes performed by reader.



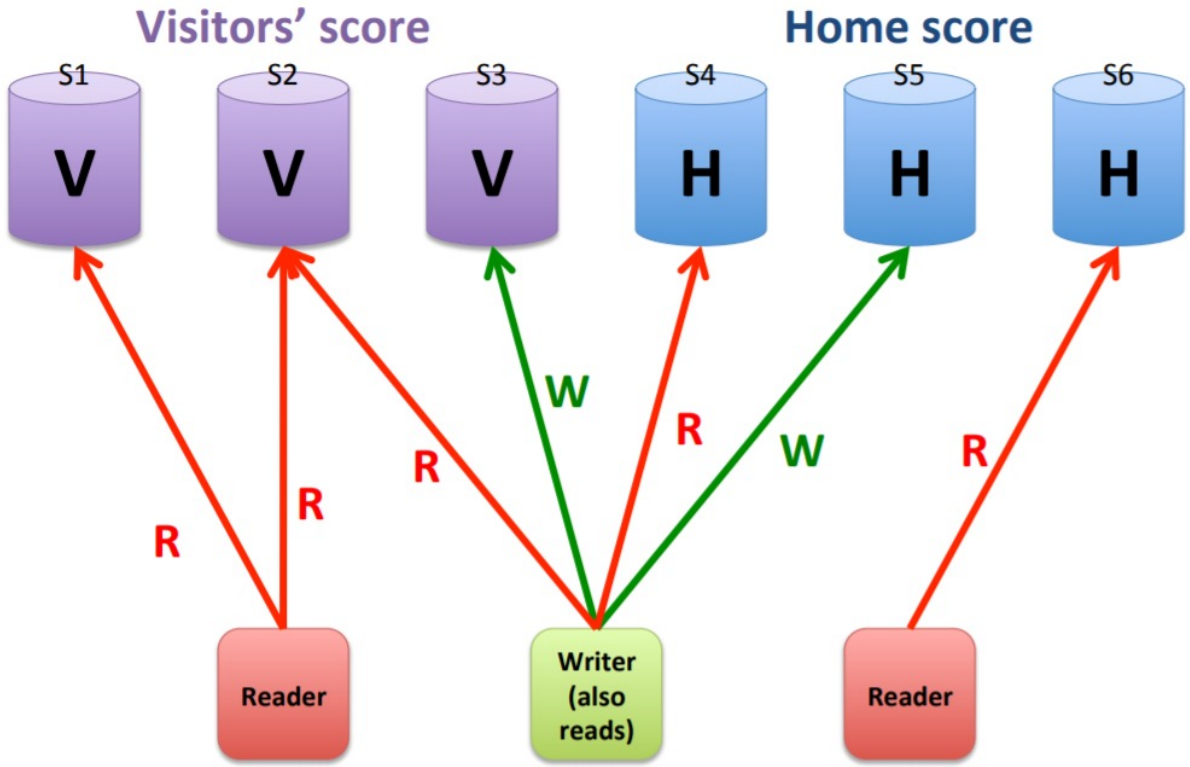
Some Consistency Guarantees



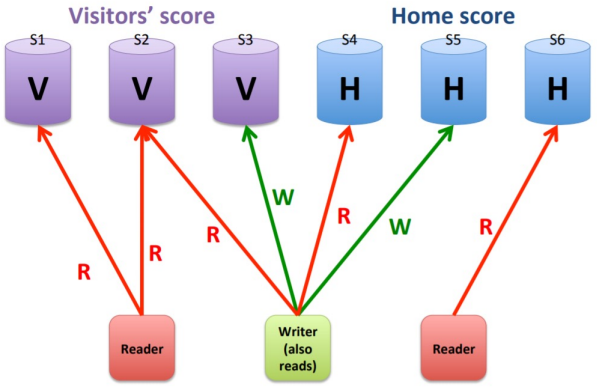
The Game of Soccer



```
for half = 1 .. 2  {
   while half not over {
        kick-the-ball-at-the-goal
        for each goal {
          if visiting-team-scored {
             score = Read ("visitors");
             Write ("visitors", score + 1);
          } else {
             score = Read ("home");
             Write ("home", score + 1);
          } } }
hScore = Read("home");
vScore = Read("visit");
if (hScore == vScore)
    play-overtime
```



Official Scorekeeper



score = **Read** ("visitors");
Write ("visitors", score + 1);

```
Write ("home", 1);
Write ("visitors", 1);
Write ("home", 2);
Write ("home", 3);
Write ("visitors", 2);
Write ("home", 4);
Write ("home", 5);
```

```
Visitors = 2
Home = 5
```

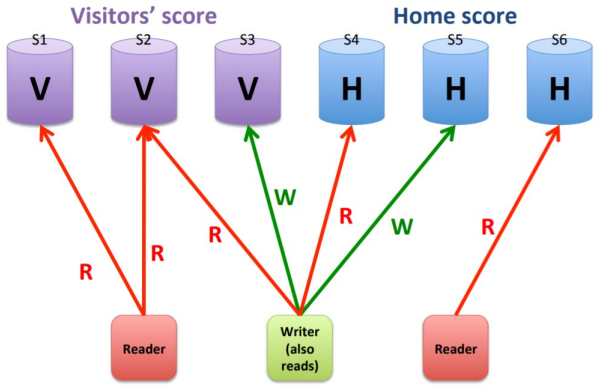
Desired consistency?

Strong

= Read My Writes!

Strong Consistency	See all previous writes.
Eventual Consistency	See subset of previous writes.
Consistent Prefix	See initial sequence of writes.
Monotonic Reads	See increasing subset of writes.
Read My Writes	See all writes performed by reader.
Bounded Staleness	See all "old" writes.

Referee



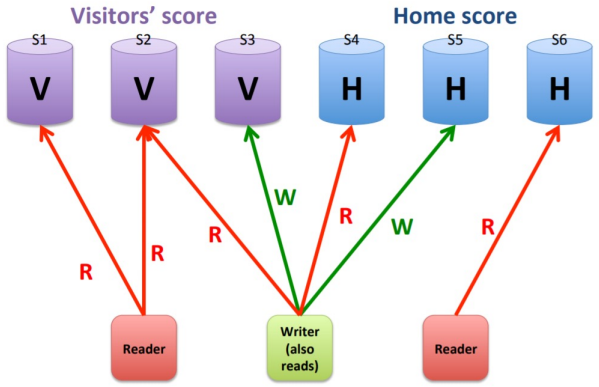

```
vScore = Read ("visitors");
hScore = Read ("home");
if vScore == hScore
      play-overtime
```

Desired consistency?

Strong consistency

Strong Consistency	See all previous writes.
Eventual Consistency	See subset of previous writes.
Consistent Prefix	See initial sequence of writes.
Monotonic Reads	See increasing subset of writes.
Read My Writes	See all writes performed by reader.
Bounded Staleness	See all "old" writes.

Radio Reporter




```
do {
    BeginTx();
        vScore = Read ("visitors");
        hScore = Read ("home");
    EndTx();
    report vScore and hScore;
    sleep (30 minutes);
}
```

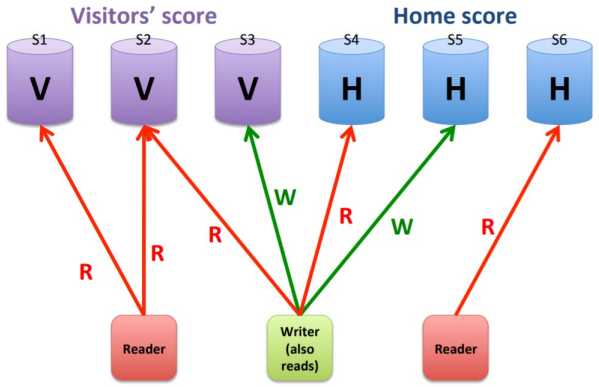
Desired consistency?

Consistent Prefix

or Bounded Staleness

Strong Consistency	See all previous writes.
Eventual Consistency	See subset of previous writes.
Consistent Prefix	See initial sequence of writes.
Monotonic Reads	See increasing subset of writes.
Read My Writes	See all writes performed by reader.
Bounded Staleness	See all "old" writes.

Sportswriter




```
While not end of game {
    drink beer;
    smoke cigar;
}
go out to dinner;
vScore = Read ("visitors");
hScore = Read ("home");
write article;
```

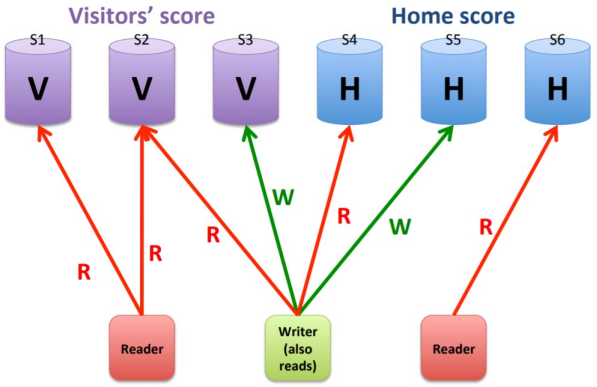
Desired consistency?

Eventual

Bounded Staleness

Strong Consistency	See all previous writes.
Eventual Consistency	See subset of previous writes.
Consistent Prefix	See initial sequence of writes.
Monotonic Reads	See increasing subset of writes.
Read My Writes	See all writes performed by reader.
Bounded Staleness	See all "old" writes.

Statistician



Wait for end of game;

score = **Read** ("home");

stat = **Read** ("season-goals");

Write ("season-goals", stat + score);

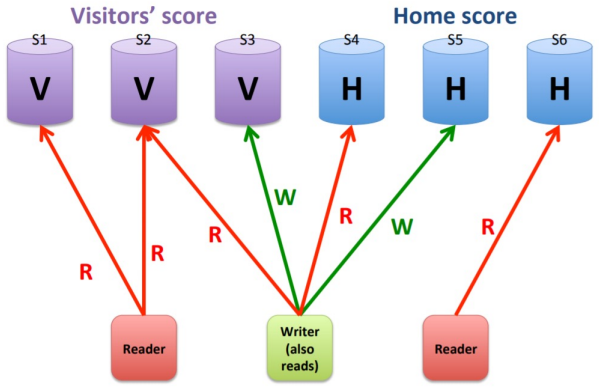
Desired consistency?

Strong Consistency (1st read)

Read My Writes (2nd read)

Strong Consistency	See all previous writes.
Eventual Consistency	See subset of previous writes.
Consistent Prefix	See initial sequence of writes.
Monotonic Reads	See increasing subset of writes.
Read My Writes	See all writes performed by reader.
Bounded Staleness	See all "old" writes.

Stat Watcher

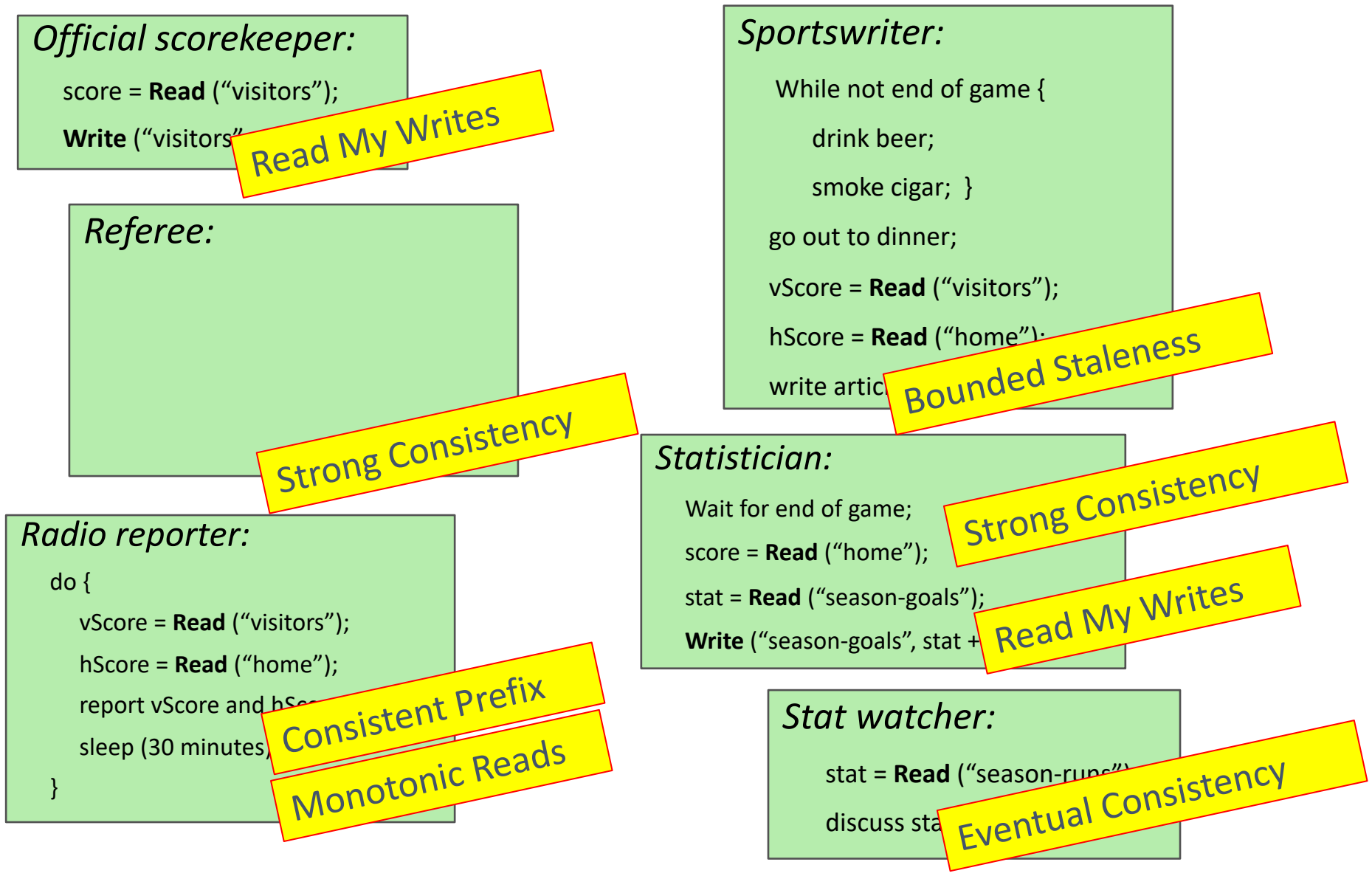



```
do {
    stat = Read ("season-goals");
    discuss stats with friends;
    sleep (1 day);
}
```

Desired consistency?

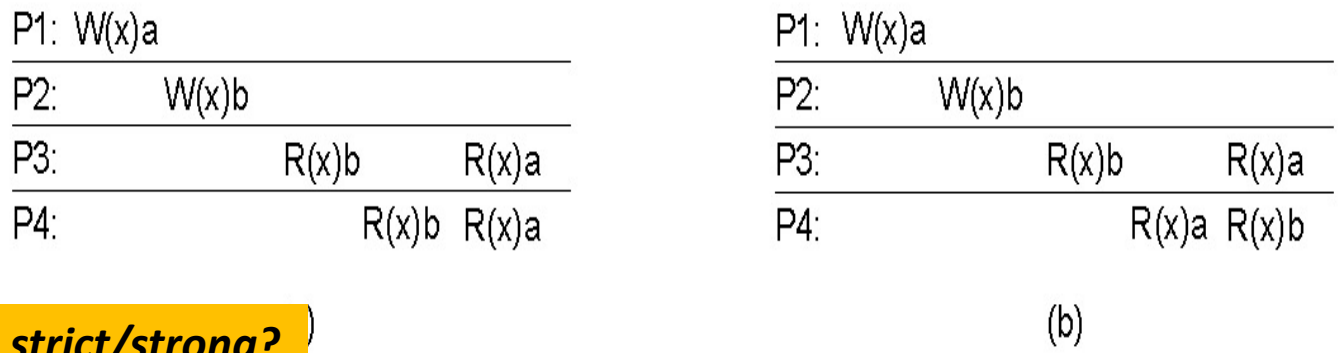
Eventual Consistency

Strong Consistency	See all previous writes.
Eventual Consistency	See subset of previous writes.
Consistent Prefix	See initial sequence of writes.
Monotonic Reads	See increasing subset of writes.
Read My Writes	See all writes performed by reader.
Bounded Staleness	See all "old" writes.


Official scorekeeper:
score = Read ("visitors");
Write ("visitors"
Read My Writes
Referee:
Strong Consistency
Radio reporter:
do {
vScore = Read ("visitors");
hScore = Read ("home");
report vScore and hSc
sleep (30 minutes)
}
Consistent Prefix
Monotonic Reads
Sportswriter:
While not end of game {
drink beer;
smoke cigar; }
go out to dinner;
vScore = Read ("visitors");
hScore = Read ("home");
write artic
Bounded Staleness
Statistician:
Wait for end of game;
score = Read ("home");
stat = Read ("season-goals");
Write ("season-goals", stat +
Strong Consistency
Read My Writes
Stat watcher:
stat = Read ("season-runs")
discuss sta
Eventual Consistency

Sequential Consistency

- weaker than strict/strong consistency
  - All operations are executed in *some* sequential order
  - each process issues operations in program order
    - Any valid interleaving is allowed
    - All agree on the same interleaving
    - Each process preserves its program order



- ***Why is this weaker than strict/strong?***
- ***Nothing is said about "most recent write"***

Linearizability

- Assumes sequential consistency *and*
  - If TS(x) < TS(y) then OP(x) should precede OP(y) in the sequence
  - Stronger than sequential consistency
  - Difference between linearizability and serializability?
    - Granularity: reads/writes versus transactions
- Example:
  - Stay tuned…relevant for lock free data structures
  - Importantly: *a property of concurrent objects*

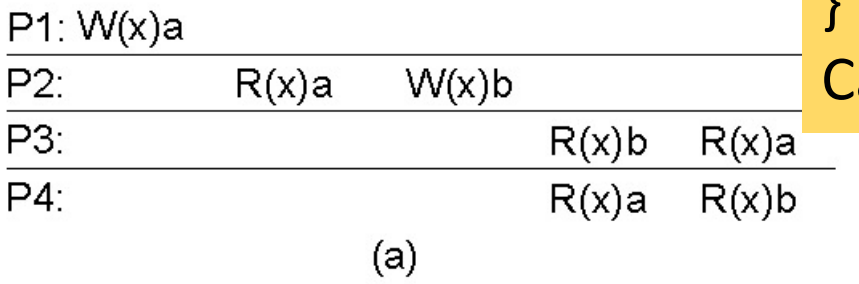
Causal consistency

- Causally related writes seer
  - *Causally?*
  - *Concurrent* writes may be se machines

Causal:
If a write produces a value that
causes another write, they are causally related

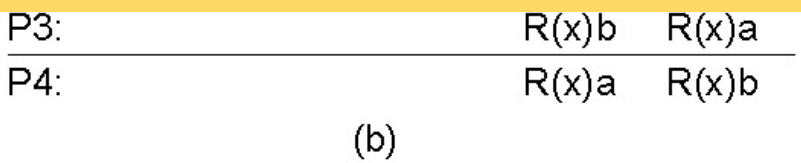
```
X = 1
if(X > 0) {
   Y = 1
}
```

Causal consistency → all see X=1, Y=1 in same order



(a)

Not permitted



(b)

Permitted

Consistency models summary

Consistency	Description
Strict	Absolute time ordering of all shared accesses matters.
Linearizability	All processes must see all shared accesses in the same order. Accesses are furthermore ordered according to a (nonunique) global timestamp
Sequential	All processes see all shared accesses in the same order. Accesses are not ordered in time
Causal	All processes see causally-related shared accesses in the same order.
FIFO	All processes see writes from each other in the order they were used. Writes from different processes may not always be seen in that order

Consistency	Description
Weak	Shared data can be counted on to be consistent only after a synchronization is done
Release	Shared data are made consistent when a critical region is exited
Entry	Shared data pertaining to a critical region are made consistent when a critical region is entered.

Non-Blocking Synchronization

Locks: a litany of problems

- Deadlock
- Priority inversion
- Convoys
- Fault Isolation
- Preemption Tolerance
- Performance

Solution: don’t use locks

Lock-free programming

- Subset of a broader class: ***Non-blocking Synchronization***
- Thread-safe access shared mutable state without mutual exclusion
- Possible without HW support
  - e.g. Lamport's Concurrent Buffer
  - …but not really practical wo HW
- Built on atomic instructions like CAS + clever algorithmic tricks
- Lock-free *algorithms* are hard, so
- General approach: encapsulate lock-free algorithms in data structures
  - Queue, list, hash-table, skip list, etc.
  - New LF data structure → research result

Basic List Append

```
struct Node
{
  int data;
  struct Node *next;
};
```

```
void append(Node** head_ref, int new_data) {
    Node* new_node = mknode(new_data, head_ref);
    if (*head_ref == NULL) {
        *head_ref = new_node;
        return;
    }
    while (last->next != NULL)
        last = last->next;
    last->next = new_node;
}
```

- Is this thread safe?
- What can go wrong?

Example: List Append

```
void append(Node** head_ref, int new_data) {
    Node* new_node = mknode(new_data);
    new_node->next = NULL;
    while(TRUE) {
        Node * last = *head_ref;
        if(last == NULL) {
            if(cas(head_ref, new_node, NULL))
                break;
        }
        while(last->next != NULL)
            last = last->next;
        if(cas(&last->next, new_node, NULL))
            break;
    }
}
```

```
struct Node
 data;
uct Node *next;
```

e?

nsure?

- Can we ensure consistent view (invariants hold) sans mutual exclusion?
- Key insight: allow inconsistent view and fix it up algorithmically

Example: SP-SC Queue

```
next(x):
    if(x == Q_size-1) return 0;
    else return x+1;

Q_get(data):
    t = Q_tail;
    while(t == Q_head)
      ;
    data = Q_buf[t];
    Q_tail = next(t);
```

```
Q_put(data):
    h = Q_head;
    while(next(h) == Q_tail)
      ;
    Q_buf[h] = data;
    Q_head = next(h);
```

- Single-producer single-consumer
- Why/when does this work?

1. Q_head is last write in Q_put, so Q_get never gets “ahead”.
2. *single* p,c only (as advertised)
3. Requires fence before setting Q head
4. Devil in the details of “wait”
5. No lock → “optimistic”

Lock-Free Stack

```
struct Node
{
  int data;
  struct Node *next;
};
```

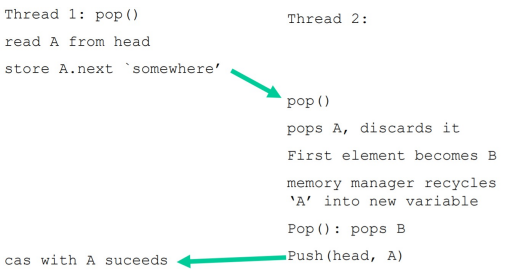
```
void push(int t) {
    Node* node = new Node(t);
    do {
        node->next = head;
    } while (!cas(&head, node, node->next));
}

bool pop(int& t) {
   Node* current = head;
   while(current) {
       if(cas(&head, current->next, current)) {
           t = current->data;
           return true;
       }
       current = head;
       }
   return false;
}
```

- Why does is it work?
- Does it enforce all invariants?

Lock-Free Stack: ABA Problem

```
Node* pop() {
    Node* current = head;
    while(current) {
        if(cas(&head, current->next, current))
            return current;
        current = head;
    }
    return false;
}
```

```
Node * node = pop();
delete node;
node = new Node(blah_blah);
push(node);
```




```
Node* pop() {
    Node* current = head;
    while(current) {
        if(cas(&head, current->next, current))
            return current;
        current = head;
    }
    return false;
}
```

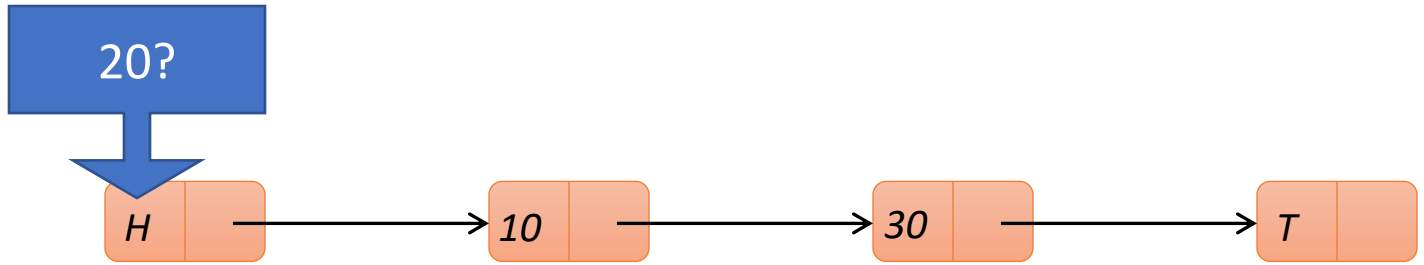
ABA Problem

- Thread 1 observes shared variable → ‘A’
- Thread 1 calculates using that value
- Thread 2 changes variable to B
  - if Thread 1 wakes up now and tries to CAS, CAS fails and Thread 1 retries
- Instead, Thread 2 changes variable back to A!
  - CAS succeeds despite mutated state
  - Very bad if the variables are pointers

- Keep update count → DCAS
- Avoid re-using memory
- Multi-CAS support → HTM

Correctness: Searching a sorted list

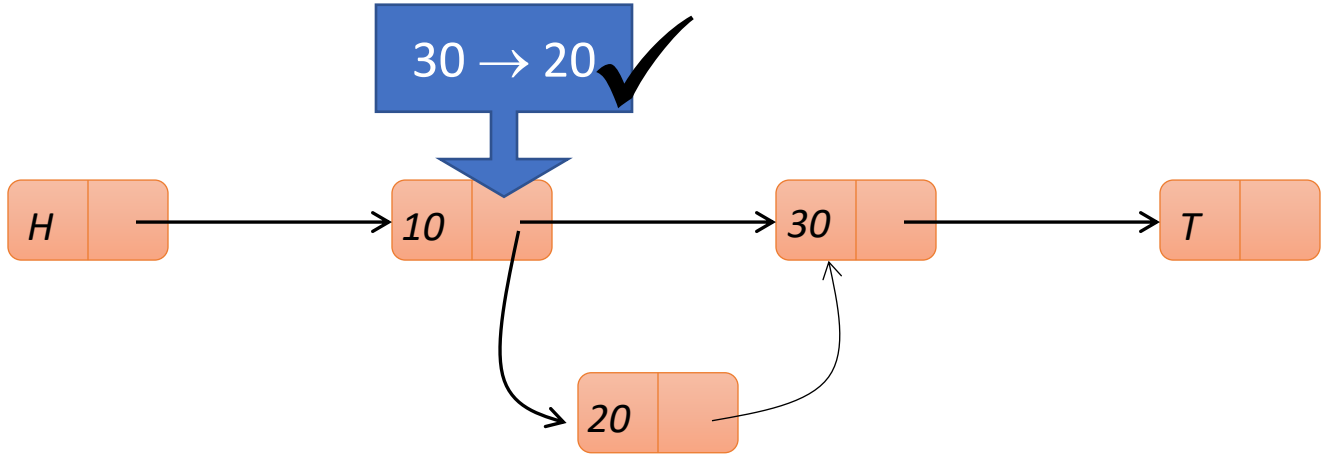
- find(20):



find(20) -> false

Inserting an item with CAS

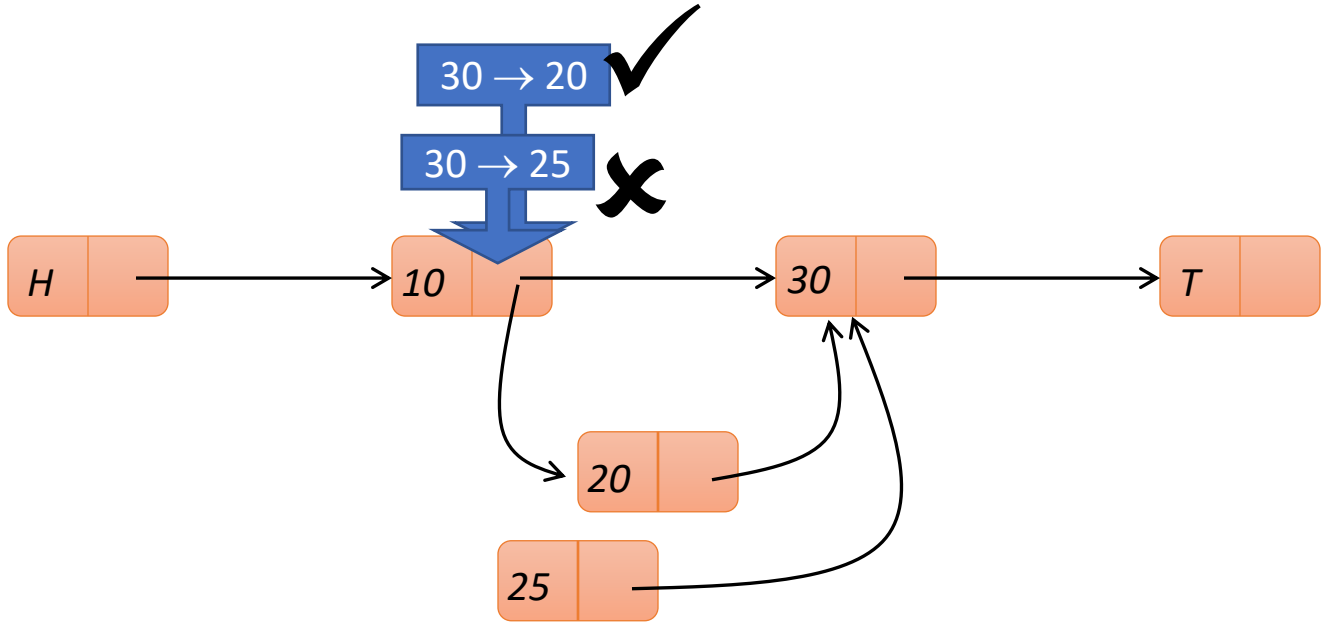
- insert(20):



insert(20) -> true

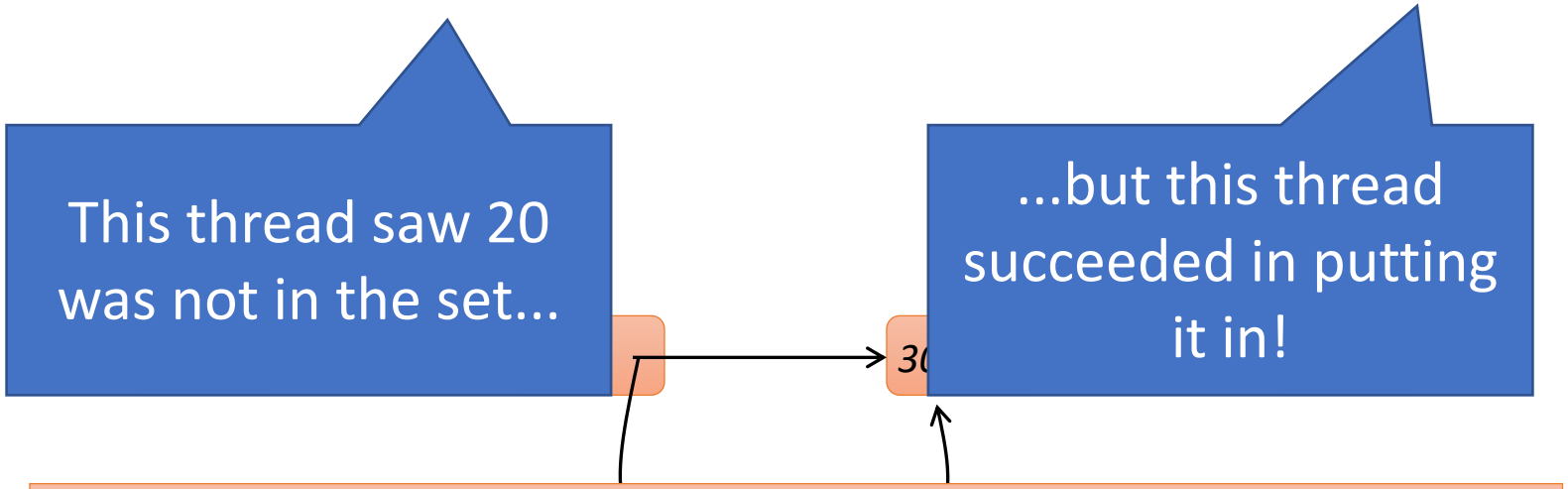
Inserting an item with CAS

- insert(20):
- insert(25):



Searching and finding together

- find(20) -> false
- insert(20) -> true



- Is this a correct implementation?
- Should the programmer be surprised if this happens?
- What about more complicated mixes of operations?

Correctness criteria

Informally:

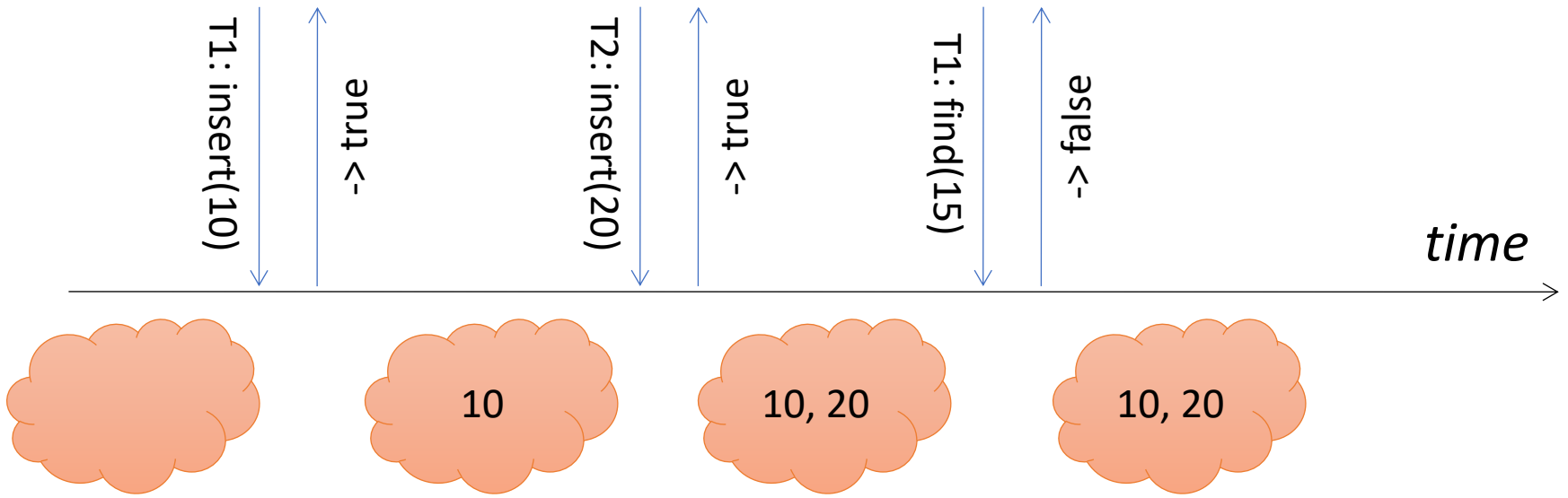
Look at the behavior of the data structure

- what operations are called on it
- what their results are

If behavior is indistinguishable from atomic calls to a sequential implementation then the concurrent implementation is correct.

Sequential history

- No overlapping invocations



Linearizability: concurrent behaviour should be similar

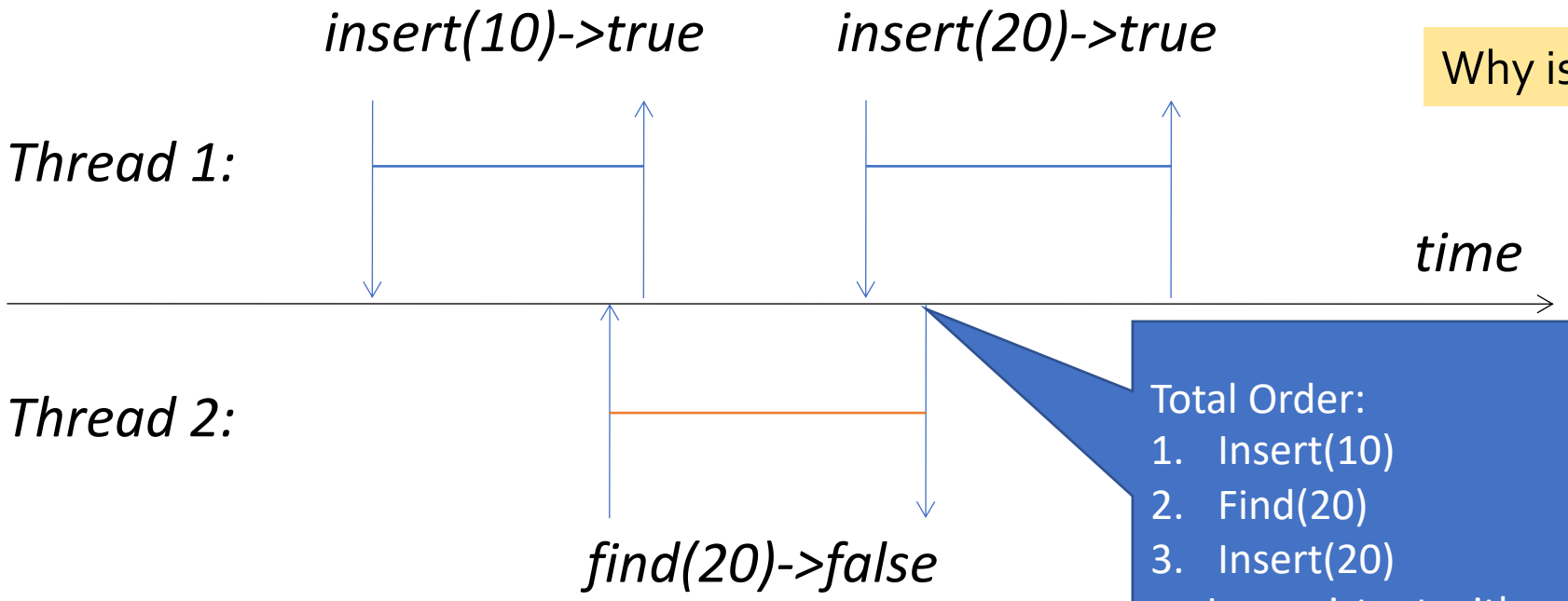
- even when threads can see intermediate state
- Recall: mutual exclusion precludes overlap

Concurrent history

Allow *overlapping* invocations

Linearizability:

- Is there a correct sequential history:
  - Same results as the concurrent one
  - Consistent with the timing of the invocations/responses?
  - Start/end impose ordering constraints



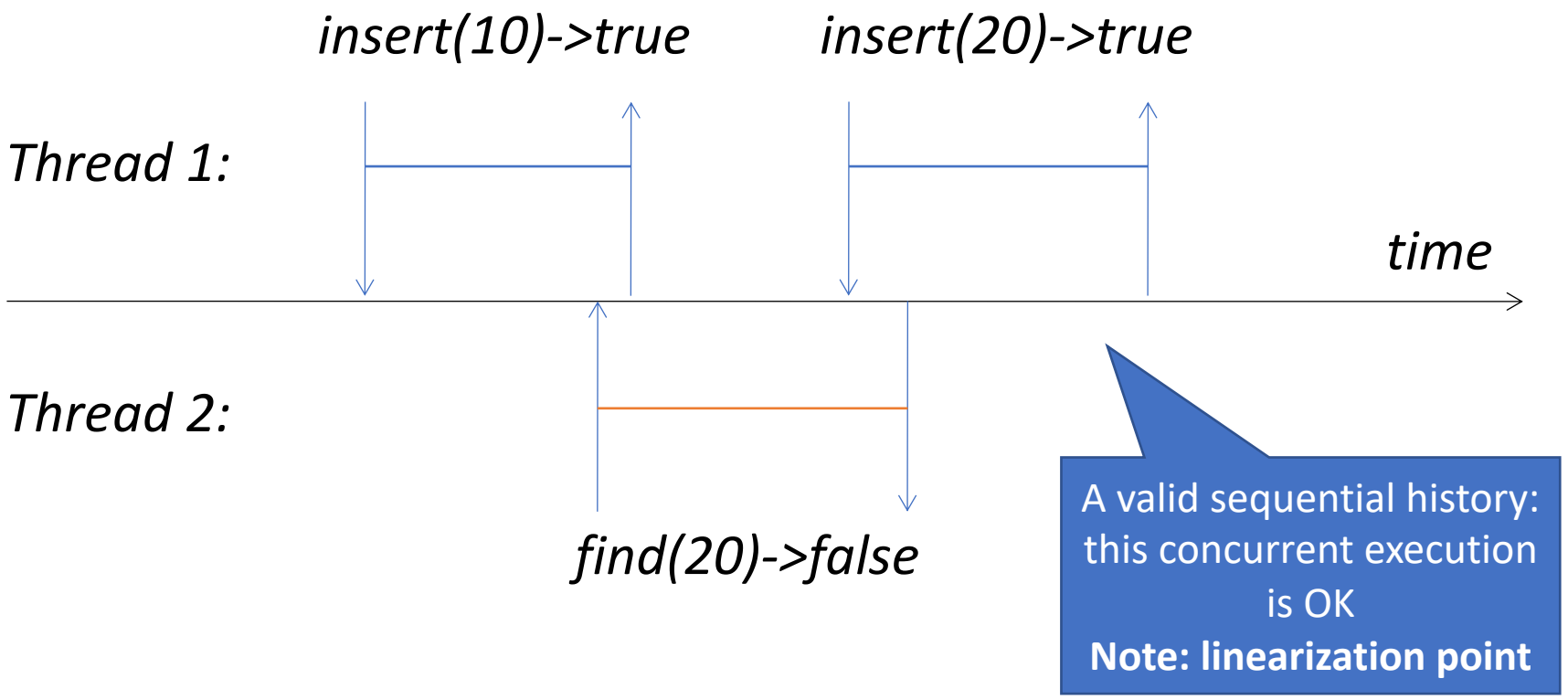
Why is this one OK?

Total Order:

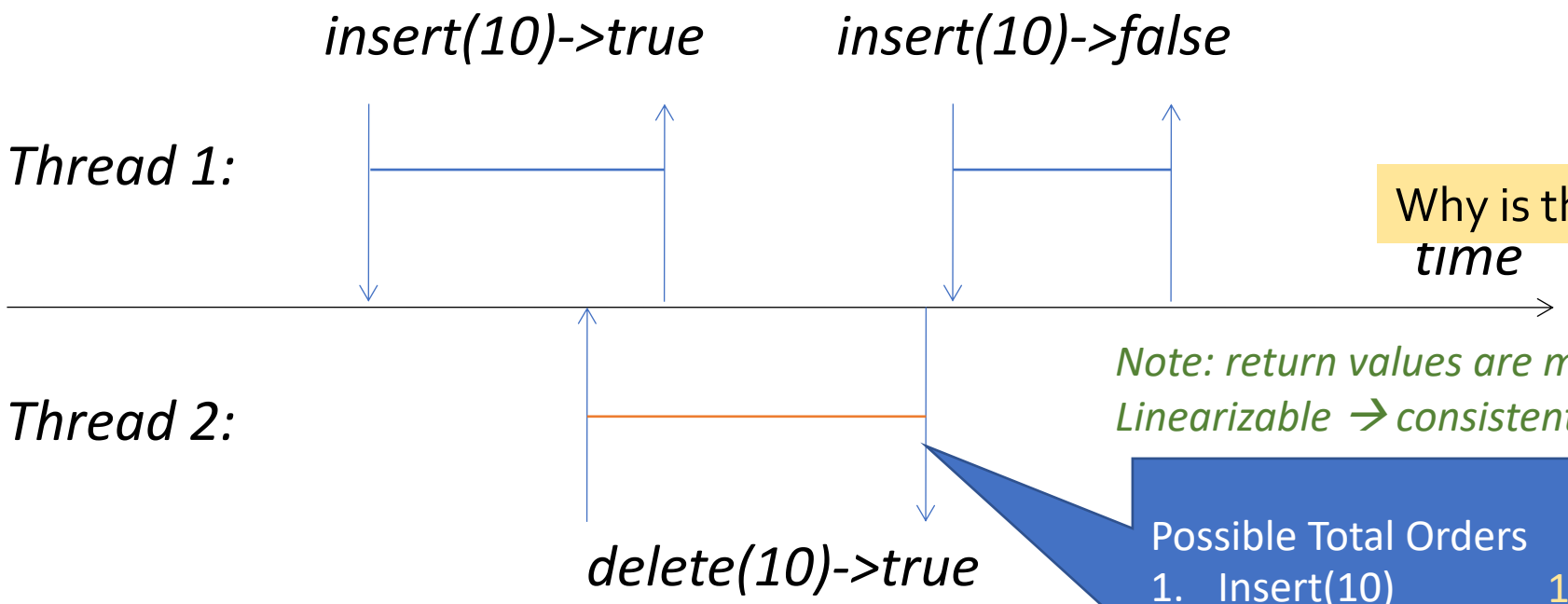
1. Insert(10)
2. Find(20)
3. Insert(20)

- Is consistent with real-time order
- 2, 3 overlap, but return order OK

Example: linearizable



Example: not linearizable



Why is this one NOT OK?

Note: return values are meaningful!
Linearizable → consistent with return values

Possible Total Orders

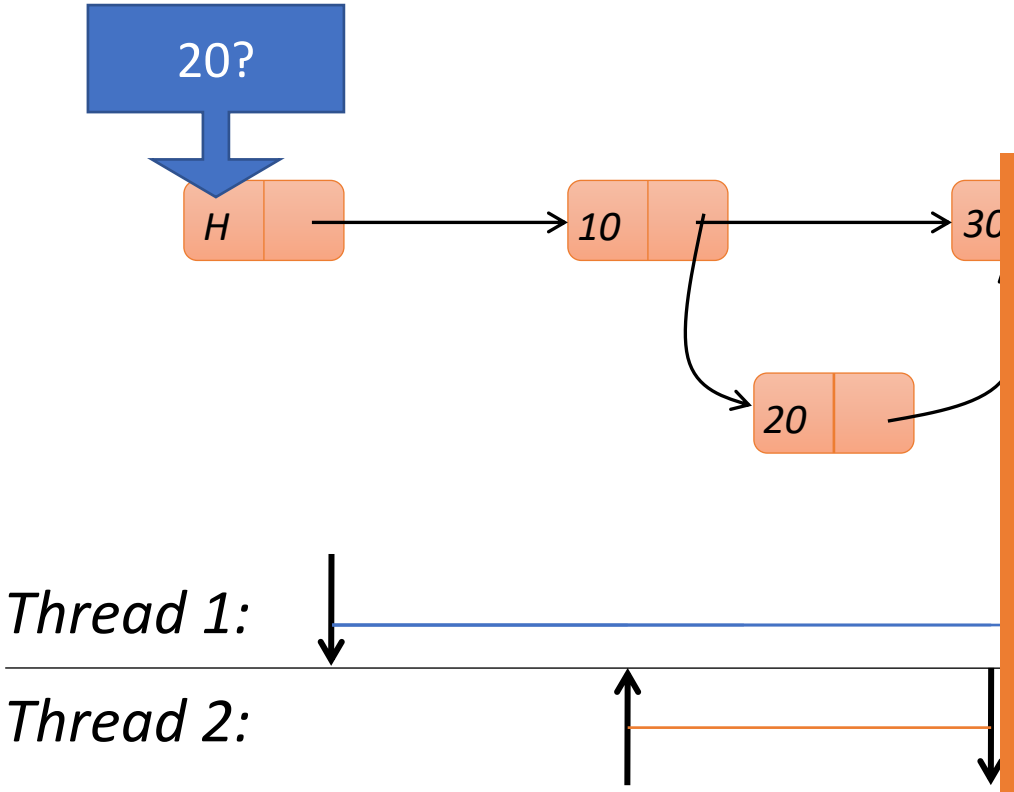
1. Insert(10)	1. Delete(10)
2. Delete(10)	2. Insert(10)
3. Insert(10)	3. Insert(10)

- Both consistent with real-time order
- 1, 2 overlap, but 3 doesn't

How can things like this happen?

Example Revisited

- find(20) -> false
- insert(20) -> true



Recurring Techniques:

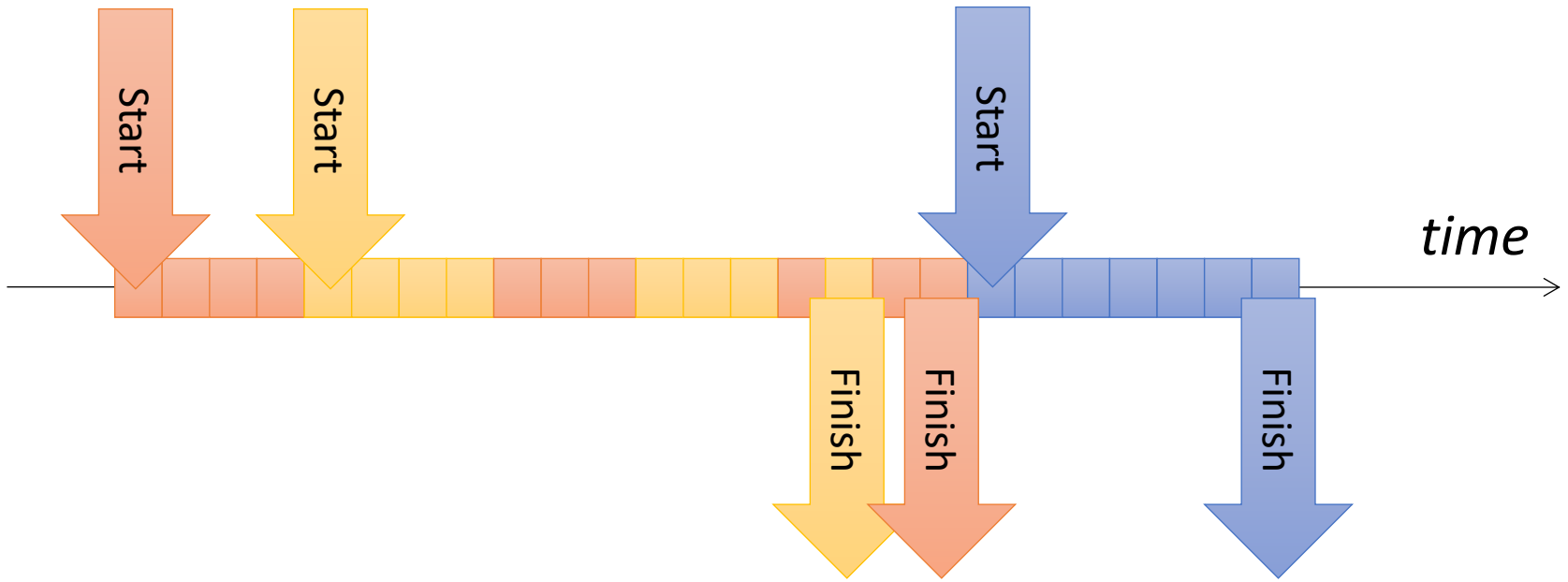
- For updates
  - Perform an essential step of an operation by a single atomic instruction
  - E.g. CAS to insert an item into a list
  - This forms a “linearization point”
- For reads
  - Identify a point during the operation’s execution when the result is valid
  - Not always a specific instruction

Formal Properties

- Wait-free
  - A thread finishes its own operation if it continues executing steps
  - Strong: everyone eventually finishes
- Lock-free
  - Some thread finishes its operation if threads continue taking steps
  - Weaker: some forward progress guaranateed, but admits unfairness, live-lock, etc.
- Obstruction-free
  - A thread finishes its own operation if it runs in isolation
  - Very weak. Means if you remove contention, someone finishes

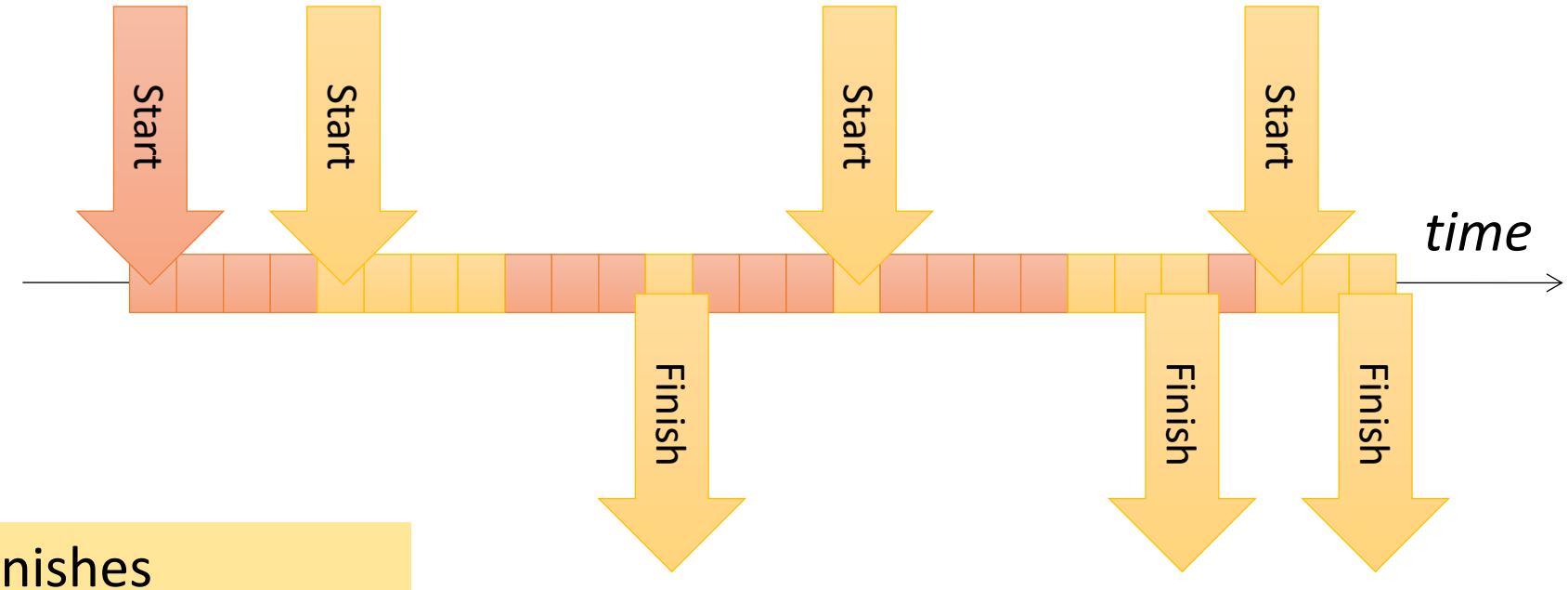
Wait-free

- A thread finishes its own operation if it continues executing steps



Lock-free

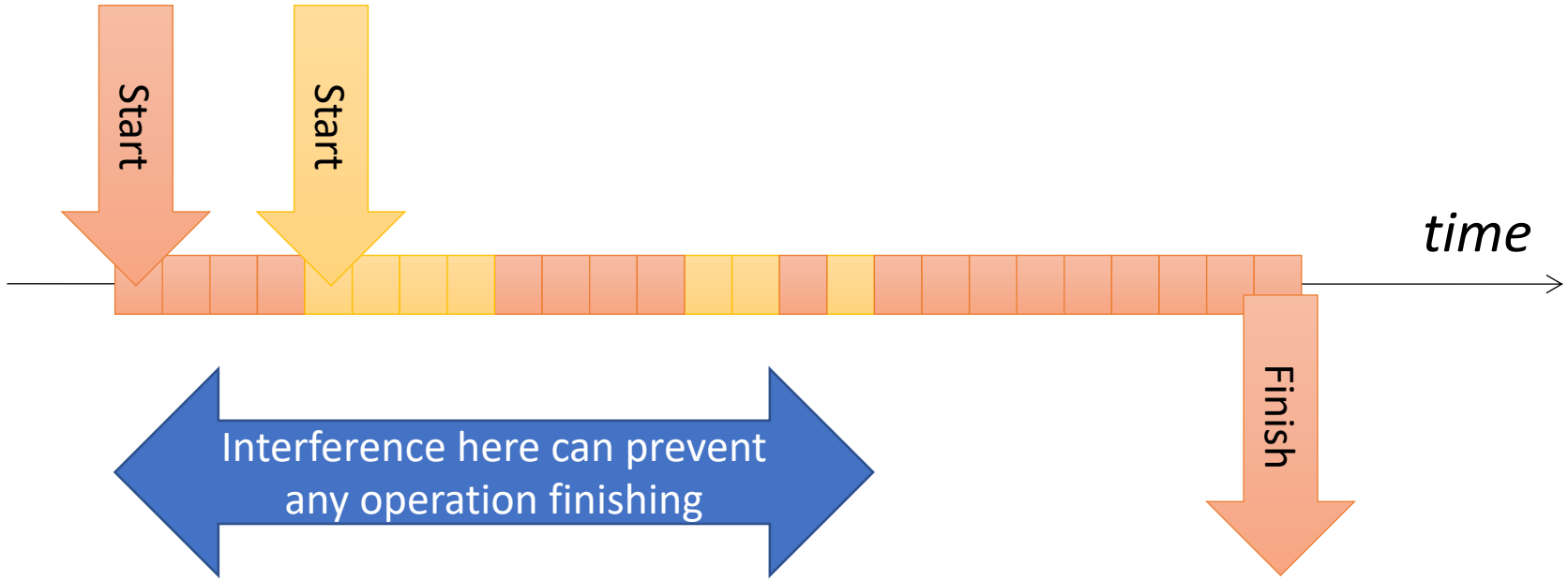
- Some thread finishes its operation if threads continue taking steps



- Red never finishes
- Orange does
- Still lock-free

Obstruction-free

- A thread finishes its own operation if it runs in isolation
- *Meaning, if you de-schedule contenders*



Formal Properties

- Wait-free
  - A thread finishes its own operation if it continue
  - Strong: everyone eventually finishes
- Lock-free
  - Some thread finishes its operation if threads continue taking steps
  - Weaker: some forward progress guaranateed, but admits unfairness, li
- Obstruction-free
  - A thread finishes its own operation if it runs in isolation
  - Very weak. Means if you remove contention, someone finishes

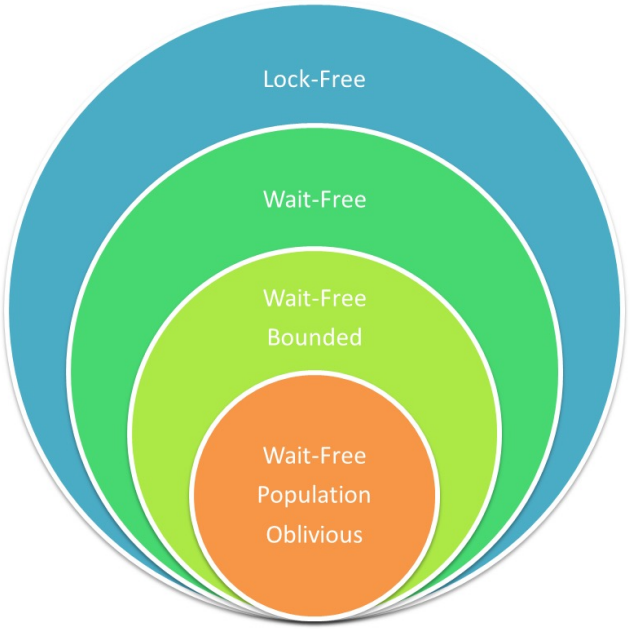
Blocking
1. Blocking
2. Starvation-Free

Obstruction-Free
3. Obstruction-Free

Lock-Free
4. Lock-Free (LF)

Wait-Free
5. Wait-Free (WF)
6. Wait-Free Bounded (WFB)
7. Wait-Free Population Oblivious (WFPO)

stronger



Linearizability Properties

- non-blocking
  - one method is never forced to wait to sync with another.
- local property:
  - a system is linearizable iff each individual object is linearizable.
  - gives us **composability**.
- Why is it important?
  - Serializability is not composable.

Huh? Composable?

Composability

Thread-safe?

```
T * list::remove(Obj key){
  LOCK(this);
  tmp = __do_remove(key);
  UNLOCK(this);
  return tmp;
}

void list::insert(Obj key, T * val){
  LOCK(this);
  __do_insert(key, val);
  UNLOCK(this);
}
```

```
void move(list s, list d, Obj key){
  tmp = s.remove(key);
  d.insert(key, tmp);
}
```

```
void move(list s, list d, Obj key){
  LOCK(s);
  LOCK(d);
  tmp = s.remove(key);
  d.insert(key, tmp);
  UNLOCK(d);
  UNLOCK(s);
}
```

- Lock-based code doesn’t compose
- If list were a linearizable concurrent data structure, composition OK

Linearizability Properties

- non-blocking
  - one method is never forced to wait to sync with another.
- local property:
  - a system is linearizable iff each individual object is linearizable.
  - gives us **composability**.
- Why is it important?
  - Serializability is not composable.
  - Core hypotheses:
    - structuring all as concurrent objects buys composability
    - structuring all as concurrent objects is tractable/possible

Practical difficulties:

- Key-value ma
- Population co
- Iteration
- Resizing the h

Options to consider when implementing a "difficult" operation:

- Relax the semantics
  (e.g., non-exact count, or non-linearizable count)
- Fall back to a simple implementation if permitted
  (e.g., lock the whole table for resize)
- Design a clever implementation
  (e.g., split-ordered lists)
- Use a different data structure
  (e.g., skip lists)