Warranties
for Faster Strong Consistency

Jed Liu Tom Magrino Owen Arden
Michael D. George Andrew C. Myers

Cornell University
Department of Computer Science

11th USENIX Symposium on Networked Systems Design and Implementation
4 April 2014

Consistency vs. scalability

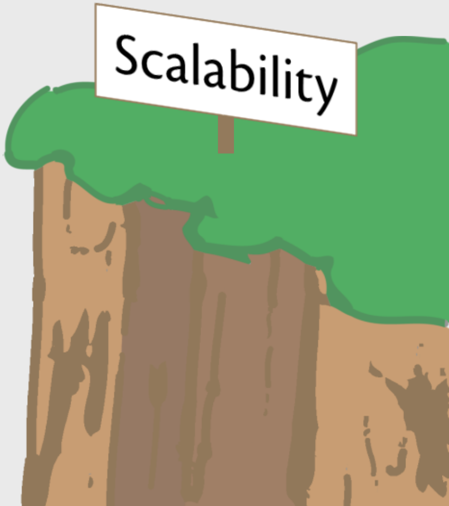
Traditional RDBMSes

- Strong consistency
  - ACID guarantees
- Simple to program
- Don't scale well

Today's "web-scale" systems

- Weak (eventual) consistency
- Offer better scalability
- Difficult to program
  - Consistency failures affect higher software layers unpredictably

Warranties help bridge the gap



Consistency: how strong?

- **Strict serializability** [Papadimitriou 1979]
  - Behaviour = sequential ordering (serializability)
  - Order of non-overlapping transactions preserved
  - Ensures transactions always see most recent state
- **External consistency** [Gifford 1981]
  - Serialization consistent w/ wall-clock time of commits

Warranties

Warranty – a time-limited assertion about system state

- **State warranty** – state of an object

  `acct == {name: "Bob", bal: 42}` until 2:00:02 p.m. (2 s)

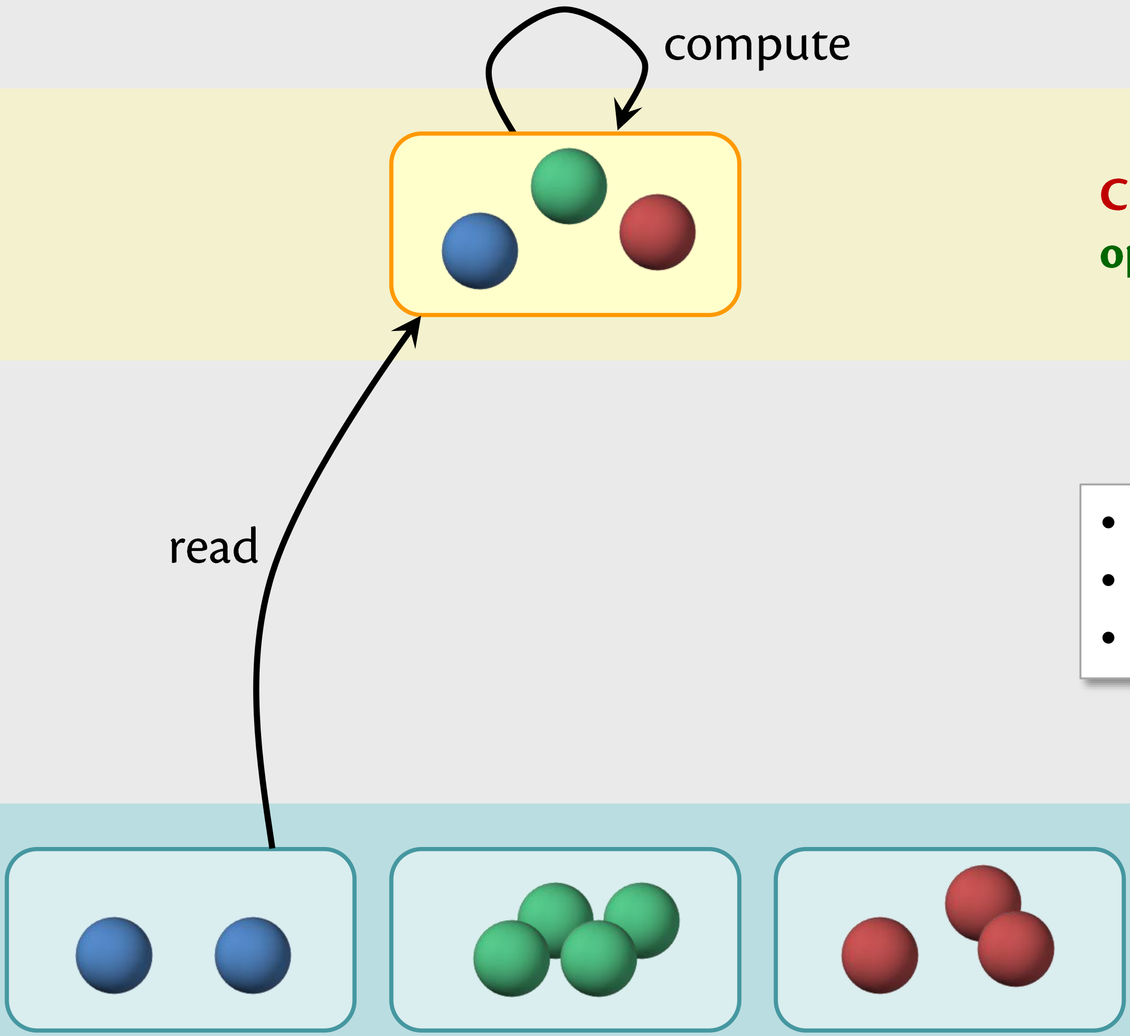
- **Computation warranty** – result of a computation

  `flight.seatsAvail(AISLE) >= 6` until 2:00:05 p.m. (5 s)

- Duration can be set automatically, adaptively
  - Each warranty **defended** to ensure assertion remains true
- Assume loosely synchronized clocks (e.g., NTP)

Warranties allow commits to **avoid communication** while guaranteeing **strict serializability** and **external consistency**

Distributed OCC refresher

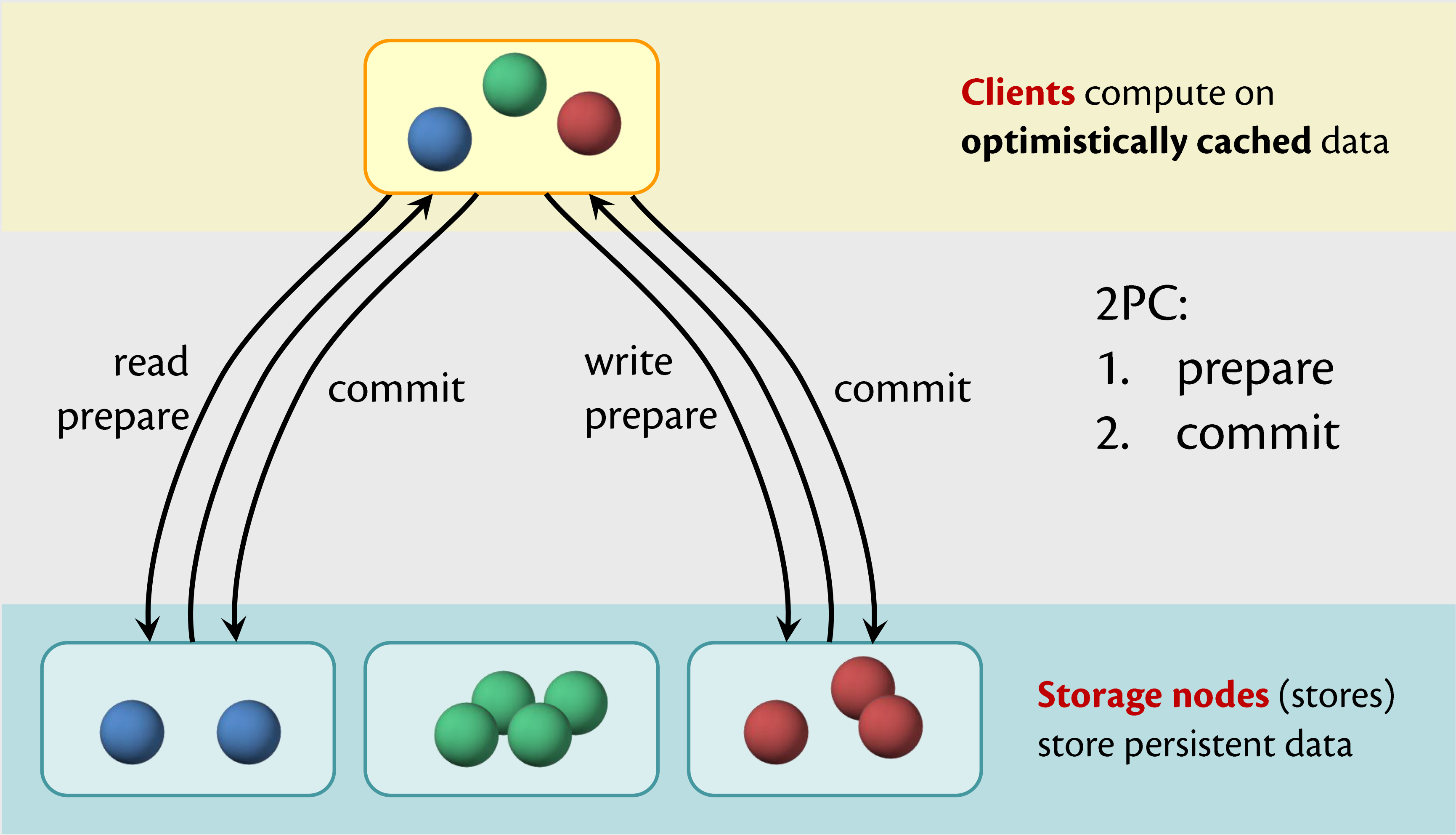


Clients compute on **optimistically cached** data

- Scalable
- Strongly consistent
- Distributed transactions

Storage nodes (stores) store persistent data

Distributed OCC refresher



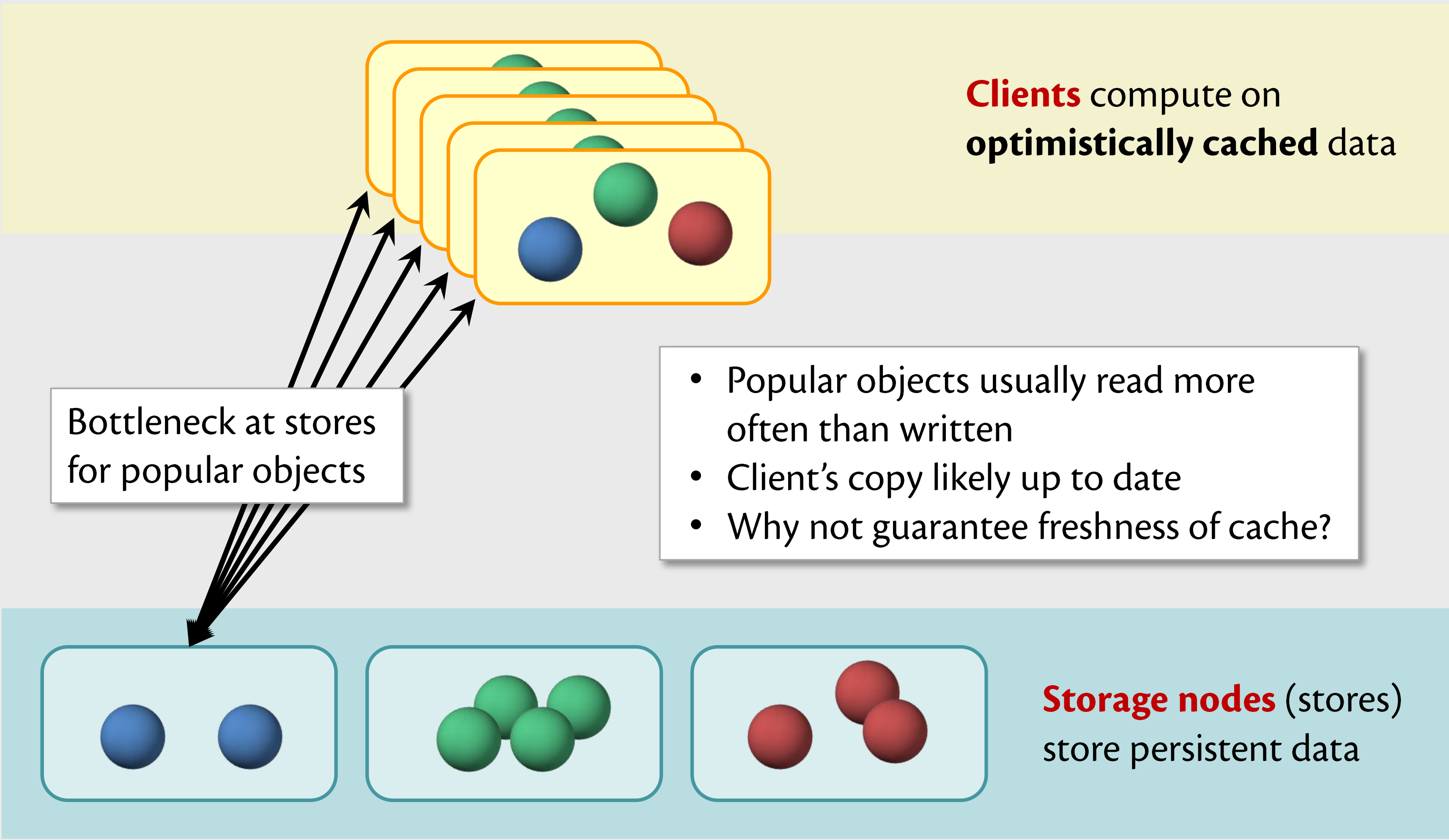
Clients compute on **optimistically cached** data

2PC:

1. prepare
2. commit

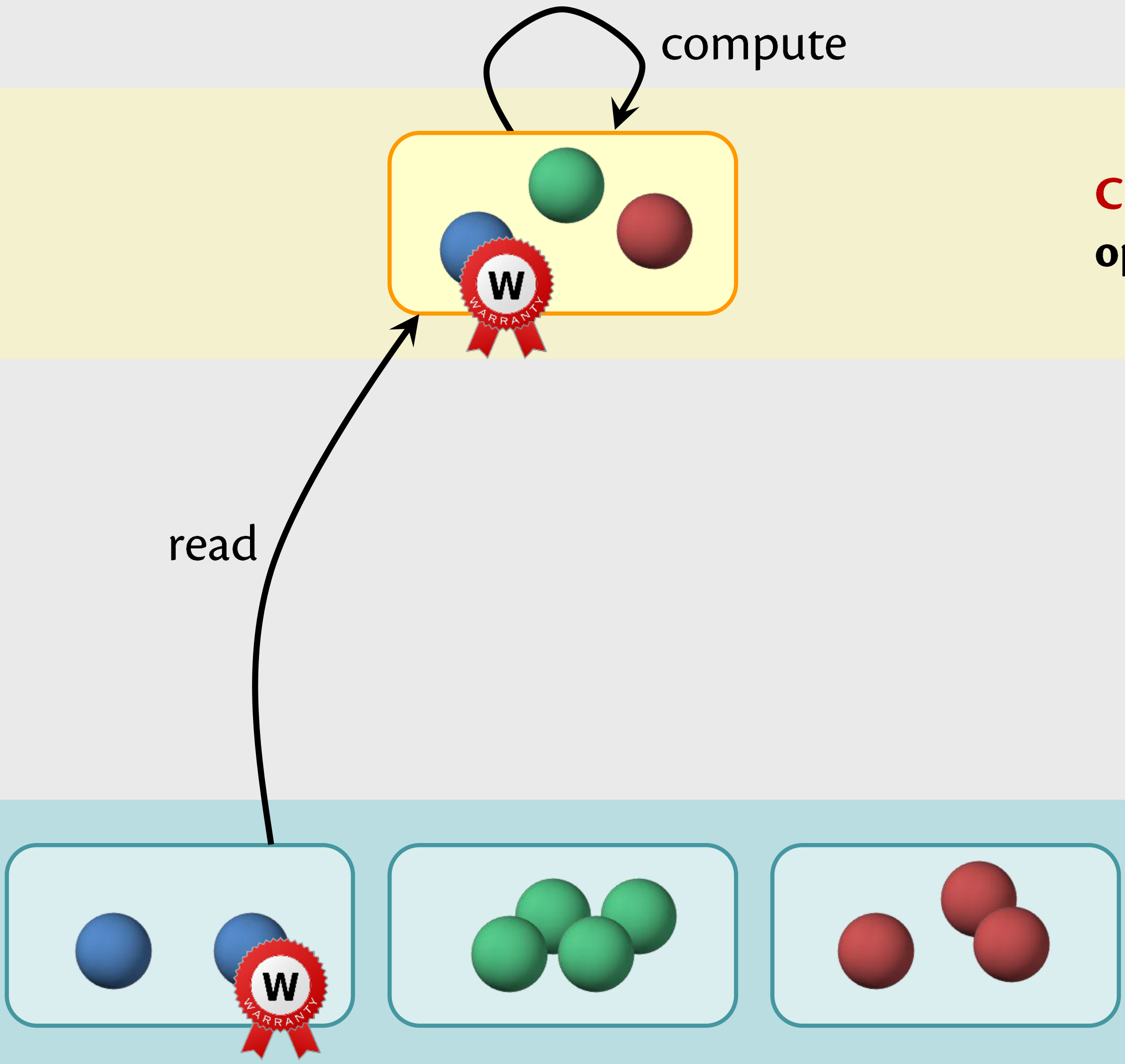
Storage nodes (stores) store persistent data

Distributed OCC refresher



- Popular objects usually read more often than written
- Client's copy likely up to date
- Why not guarantee freshness of cache?

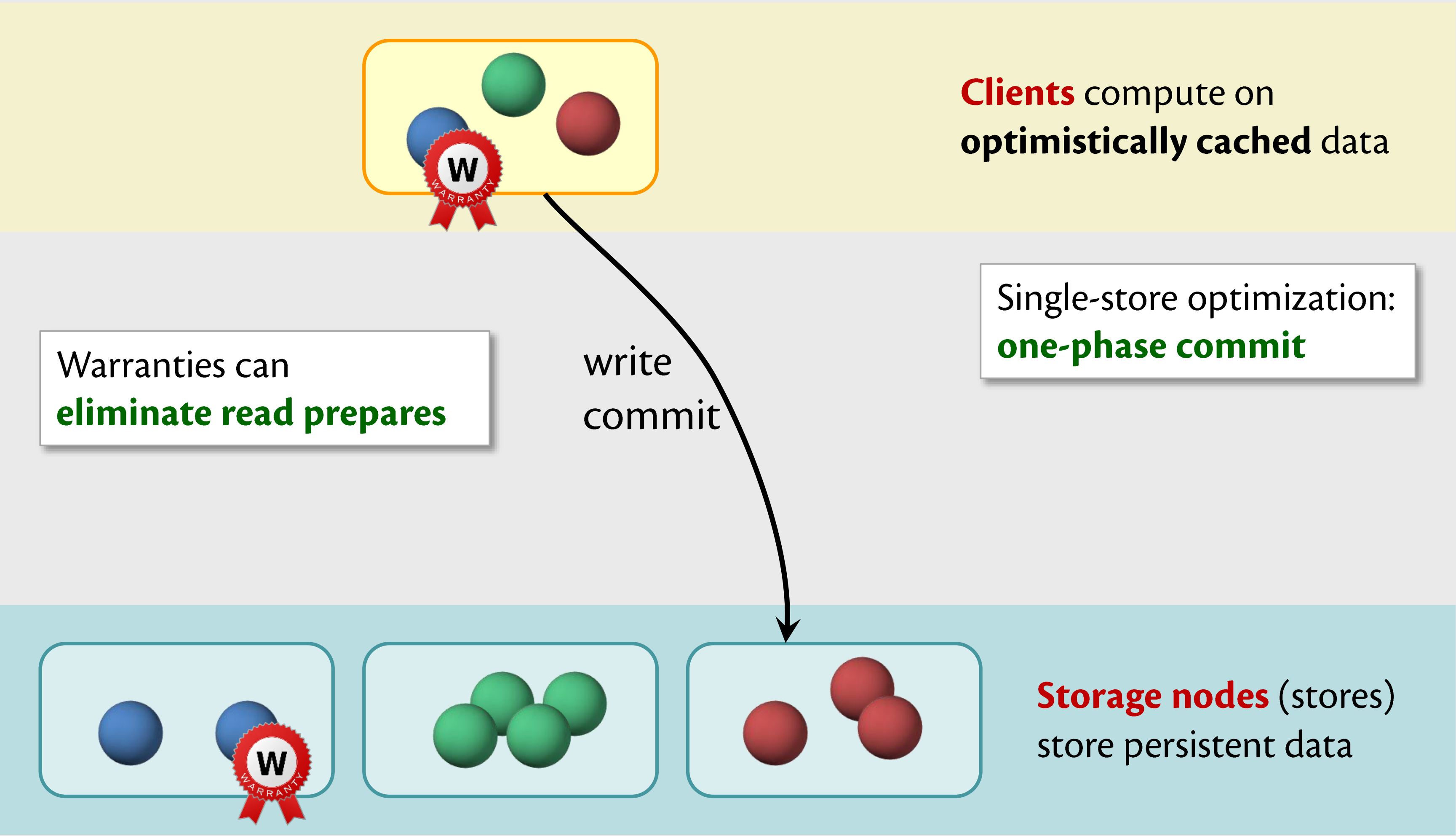
Warranties avoid communication



Clients compute on **optimistically cached** data

Storage nodes (stores) store persistent data

Warranties avoid communication



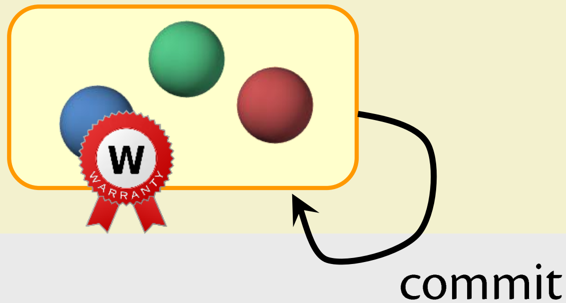
Clients compute on **optimistically cached** data

Single-store optimization: **one-phase commit**

Warranties can **eliminate read prepares**

Storage nodes (stores) store persistent data

Warranties avoid communication

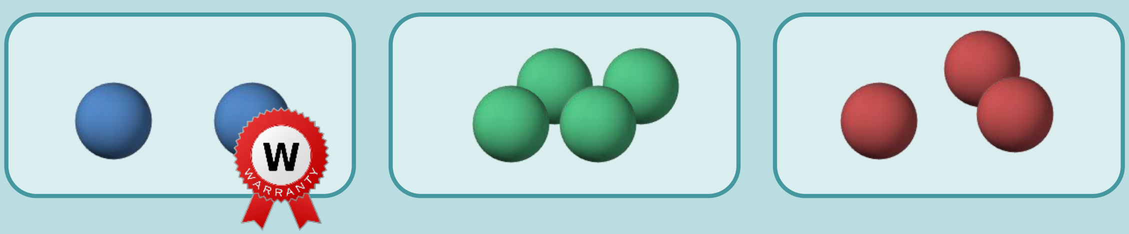


Clients compute on **optimistically cached** data

Single-store optimization: **one-phase commit**

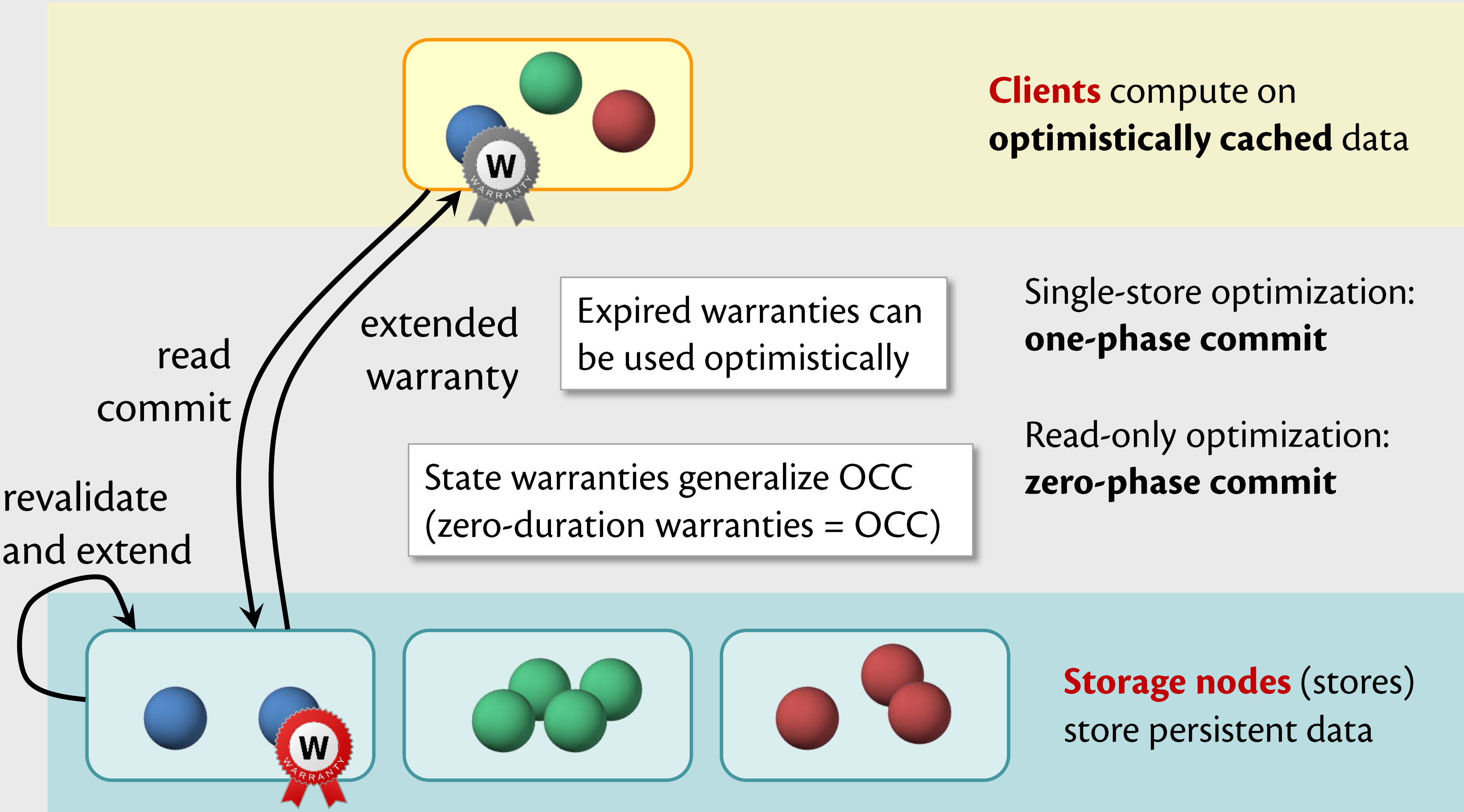
Warranties can **eliminate read prepares**

Read-only optimization: **zero-phase commit**



Storage nodes (stores) store persistent data

Using expired warranties



Warranties are related to read leases

- Leases [GC89] give time-limited **rights** to resources
  - e.g., use IP address, read object, write object
  - Must have lease to perform corresponding action
    - Can relinquish lease when no longer needed
  - Allow outsourcing of consistency to clients
- Warranties: a shift in perspective
  - Time-limited **assertions**: "What's true in the system?"
  - Some overlap: state warranties similar to read leases
  - Naturally generalize to computation warranties

Memoized methods

One lightweight way to present computation warranties in language

– e.g., extend Java:

memoized = issue warranties on method result

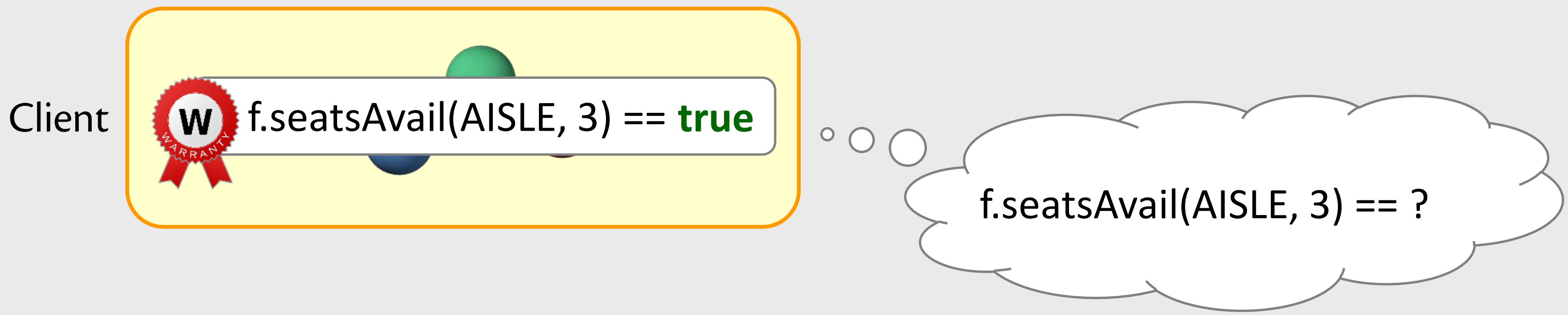
Memoized method declaration

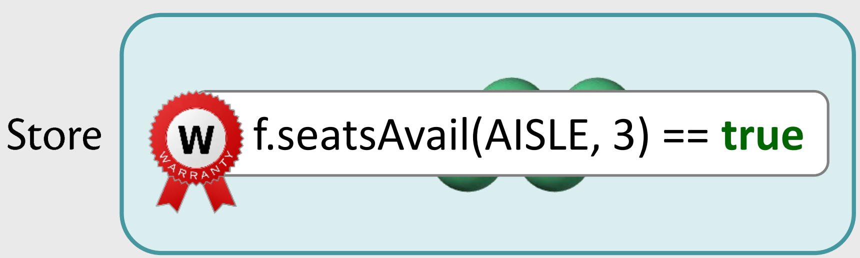
```
memoized boolean seatsAvail(SeatType t, int n) {
  return seatsAvail(t) >= n;
}
```

Client code (ordinary Java)

```
for (Flight f : flights)
  if (f.seatsAvail(AISLE, 3))
    displayFlights.add(f);
```

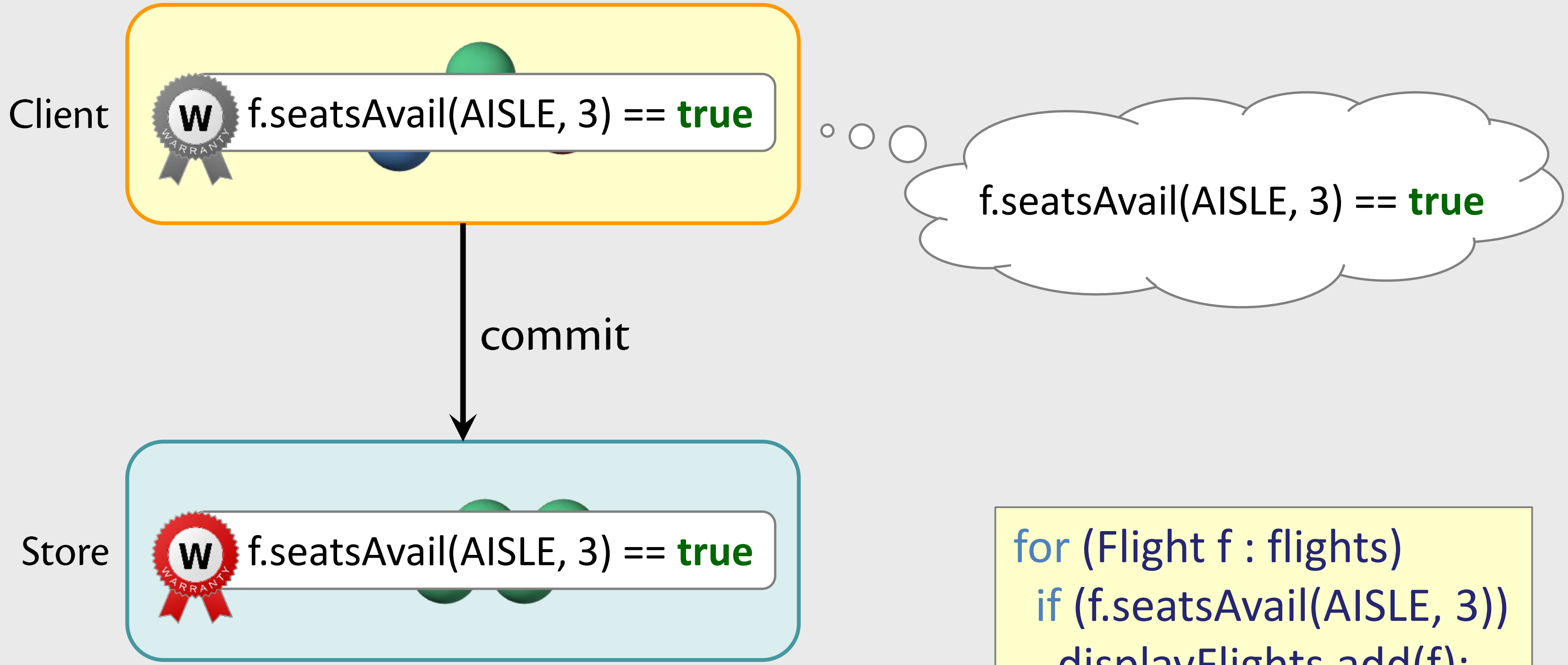
Using computation warranties






```
for (Flight f : flights)
  if (f.seatsAvail(AISLE, 3))
    displayFlights.add(f);
```

Proposing computation warranties




```
for (Flight f : flights)
  if (f.seatsAvail(AISLE, 3))
    displayFlights.add(f);
```

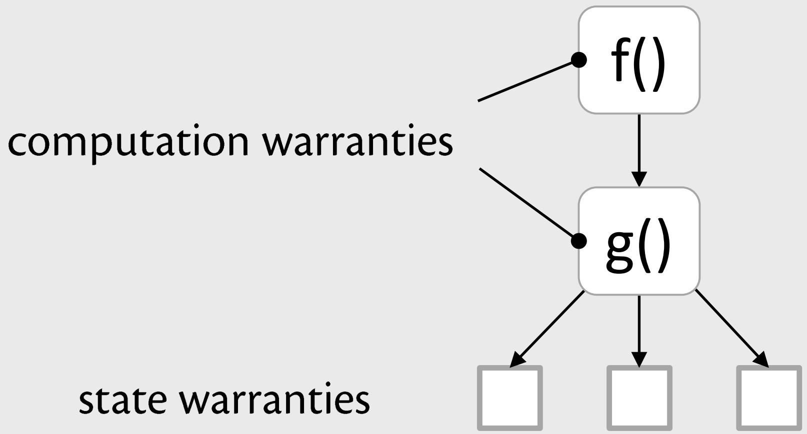
Warranty dependencies

- Computation warranties can depend on other warranties

```
memoized int f() {
  return g() + 1;
}

memoized int g() { … }
```

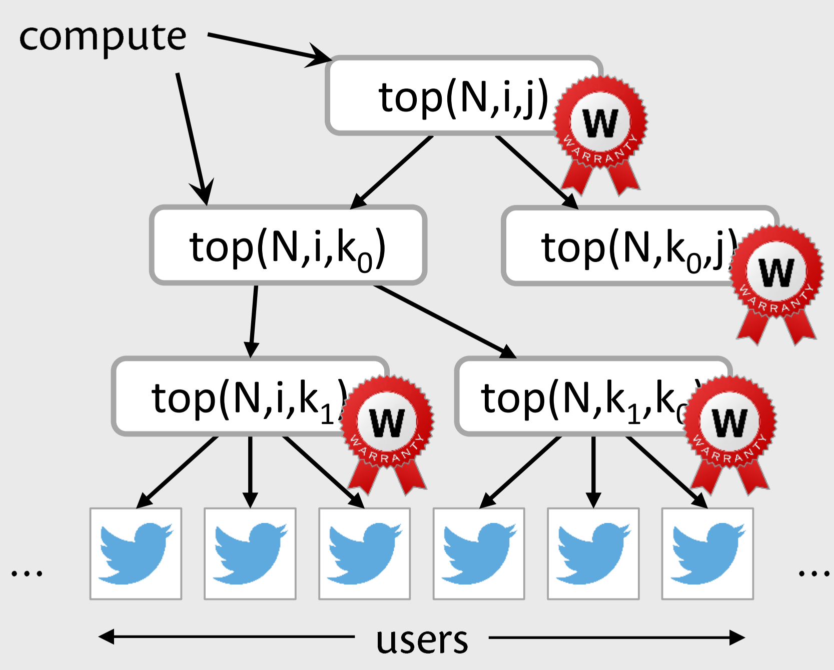
Warranty dependency tree



Twitter analytics example

- Who are the top N most-followed Twitter users?
  - Unlikely to change often, though followers change frequently
- Divide & conquer implementation
  - Allows incremental computation of new warranties

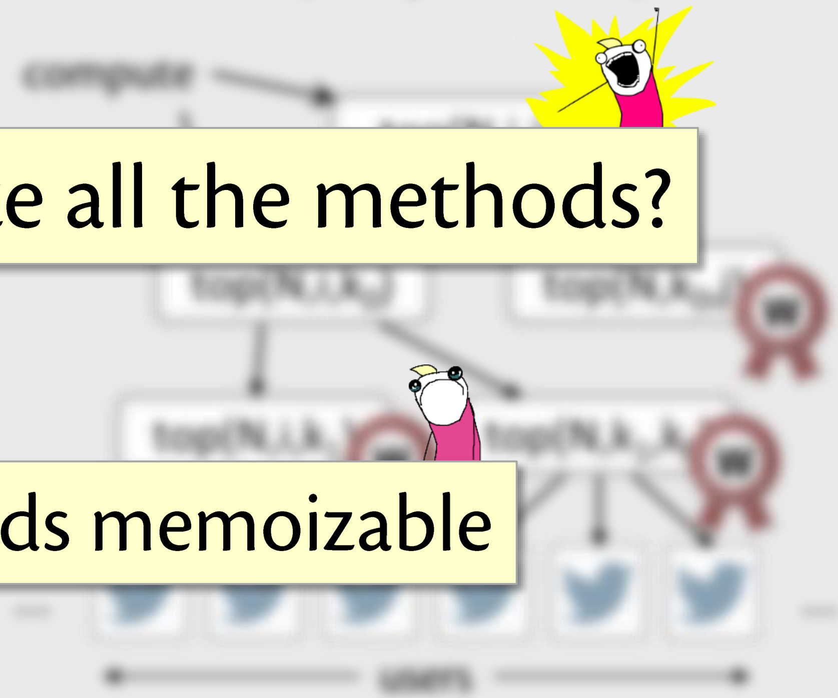
Warranty dependency tree



Twitter analytics example

- Who are the top N most-followed Twitter users?
  - frequently
- implementation
  - Allows incremental computation of new warranties

Warranty dependency tree



Why not memoize all the methods?

Not all methods memoizable

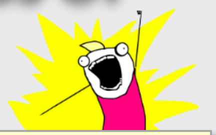
Not all methods memoizable

- Behaviour should be identical regardless of whether warranty is used

- Memoized computations must:
  - Be deterministic
  - Have no observable side effects
    - i.e., cannot modify pre-existing objects

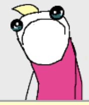
Not all methods memoizable

- Behaviour should be identical regardless of whether warranty is used



Let's memoize all the other methods!

- Memoized computations must:
  - Be deterministic
  - Ha

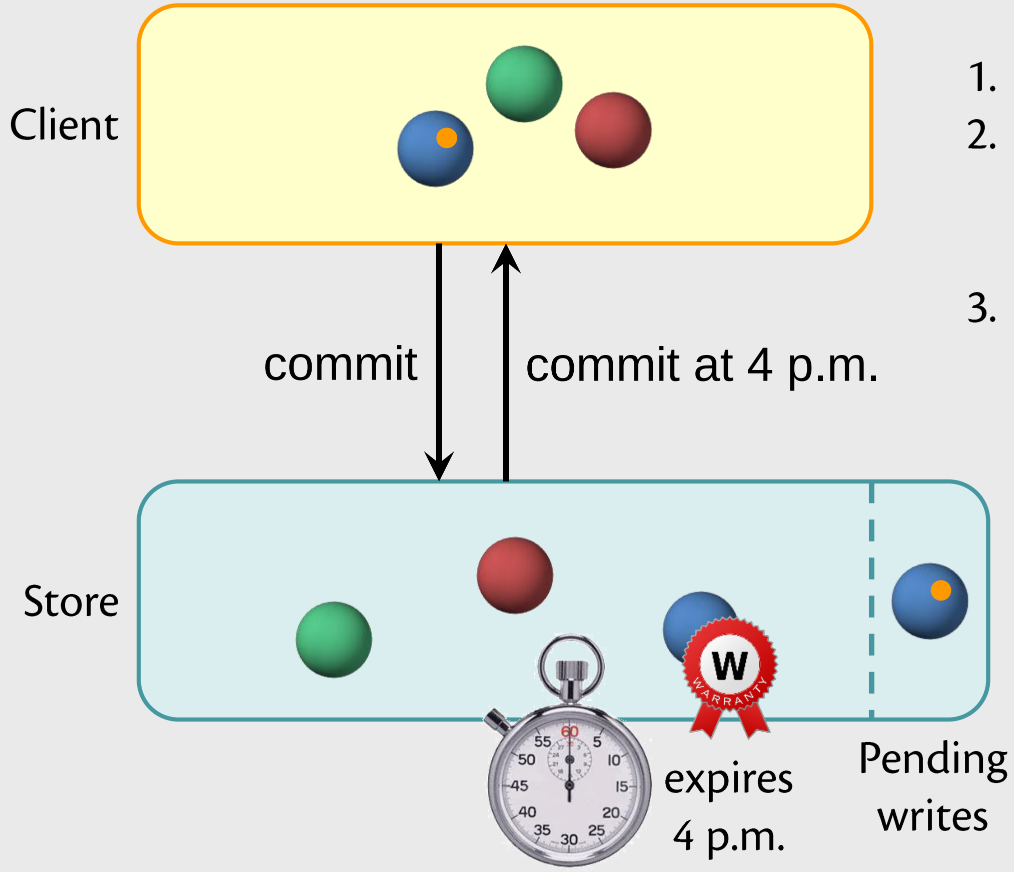


Warranties aren't free:

- Creation & bookkeeping have cost
- Need to be **defended** against writes that invalidate them

Defending state warranties

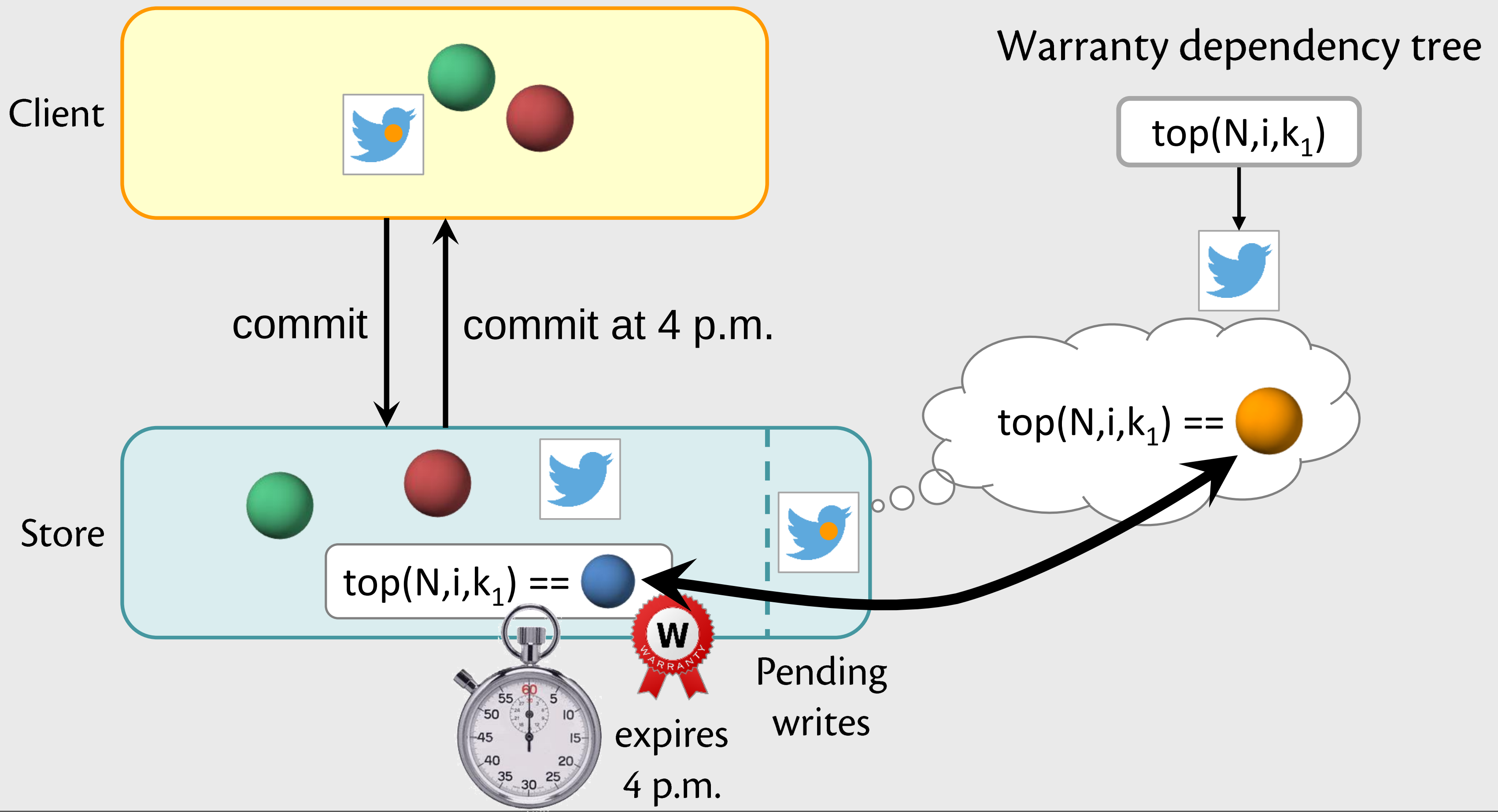
- Writes delayed until conflicting warranties expire



1. Client sends update to store
2. Store notices conflicting warranty
   - Write is delayed
   - Client notified of delayed commit
3. Update commits when warranty expires

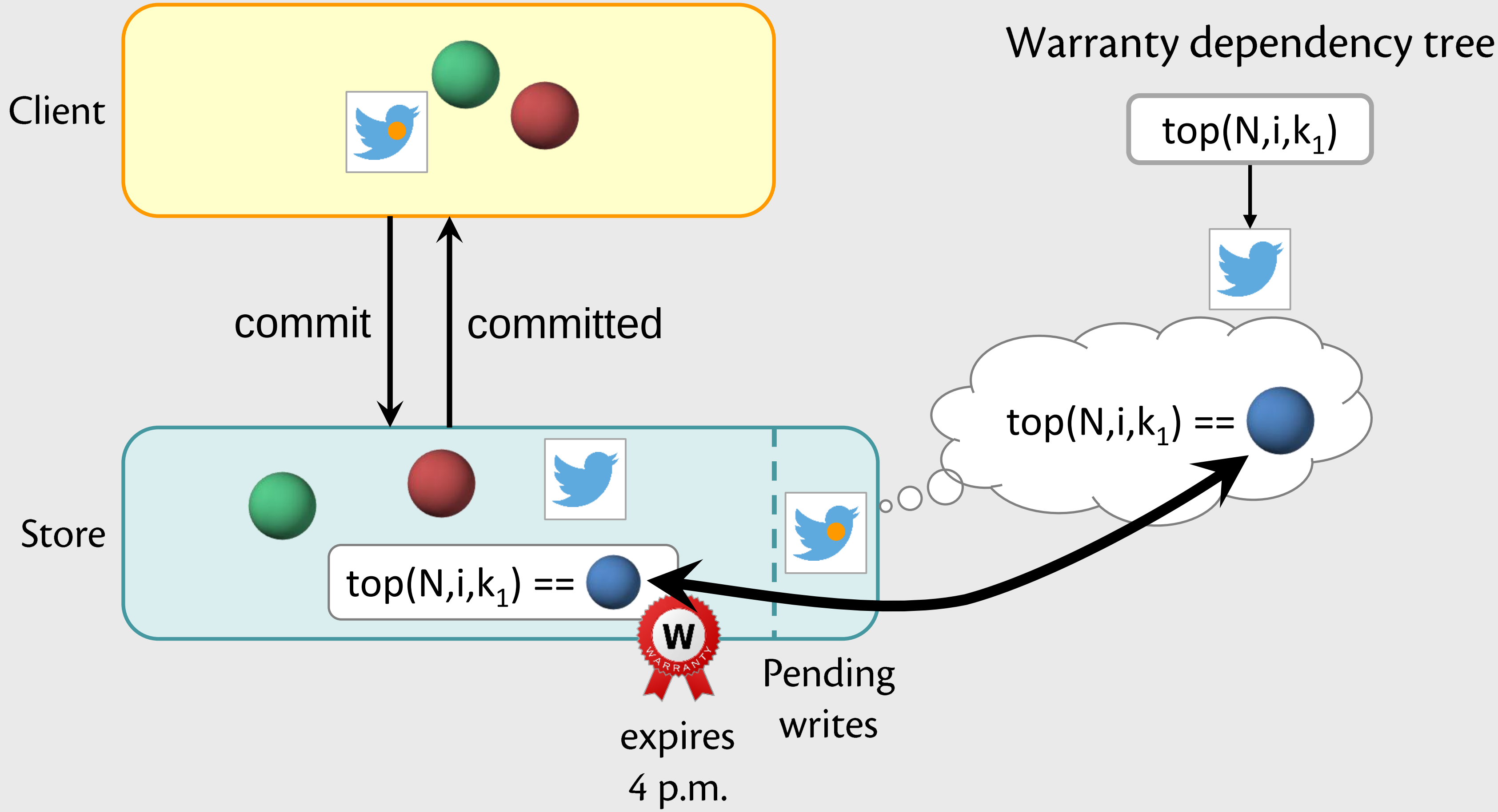
Defending computation warranties

- Writes delayed until conflicting warranties expire



Defending computation warranties

- Writes delayed until conflicting warranties expire



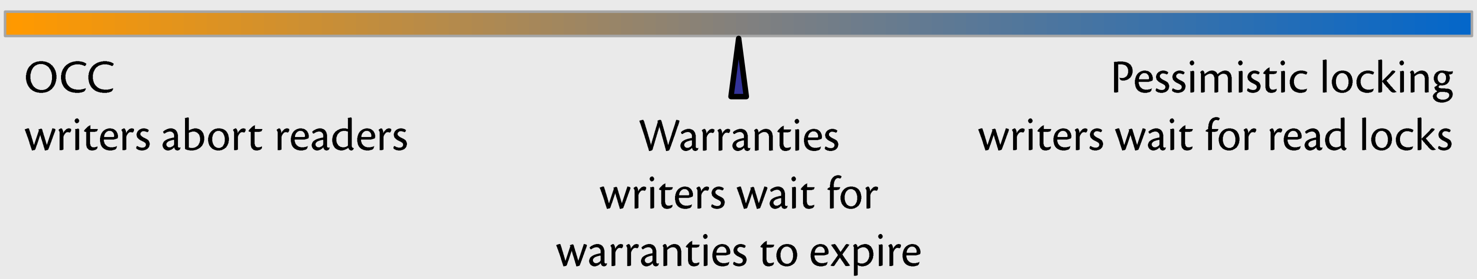
Warranty durations

- Warranties can delay writes
- Key to performance: **warranty durations**
  - Long enough to be useful
  - Short enough to keep writers from blocking
  - **Automatic, adaptive, online mechanism**
    - Analytical model driven by run-time measurements

Frequently used & seldom changed	➡	long warranties
Frequently changes or seldom used	➡	short warranties (if any at all)

Trade-offs

- Unavoidable trade-off between readers & writers
  - Read performance improved, but writes delayed



Implementation

- Extended Fabric [SOSP 2009]
  - Secure distributed object system
  - High-level programming model
    - Presents persistent data as ordinary language-level objects

- Support for both state & computation warranties
  - Fabric language extended with memoized methods

Fabric 0.2.1	44 kLOC
Warranties extension	7 kLOC added or modified

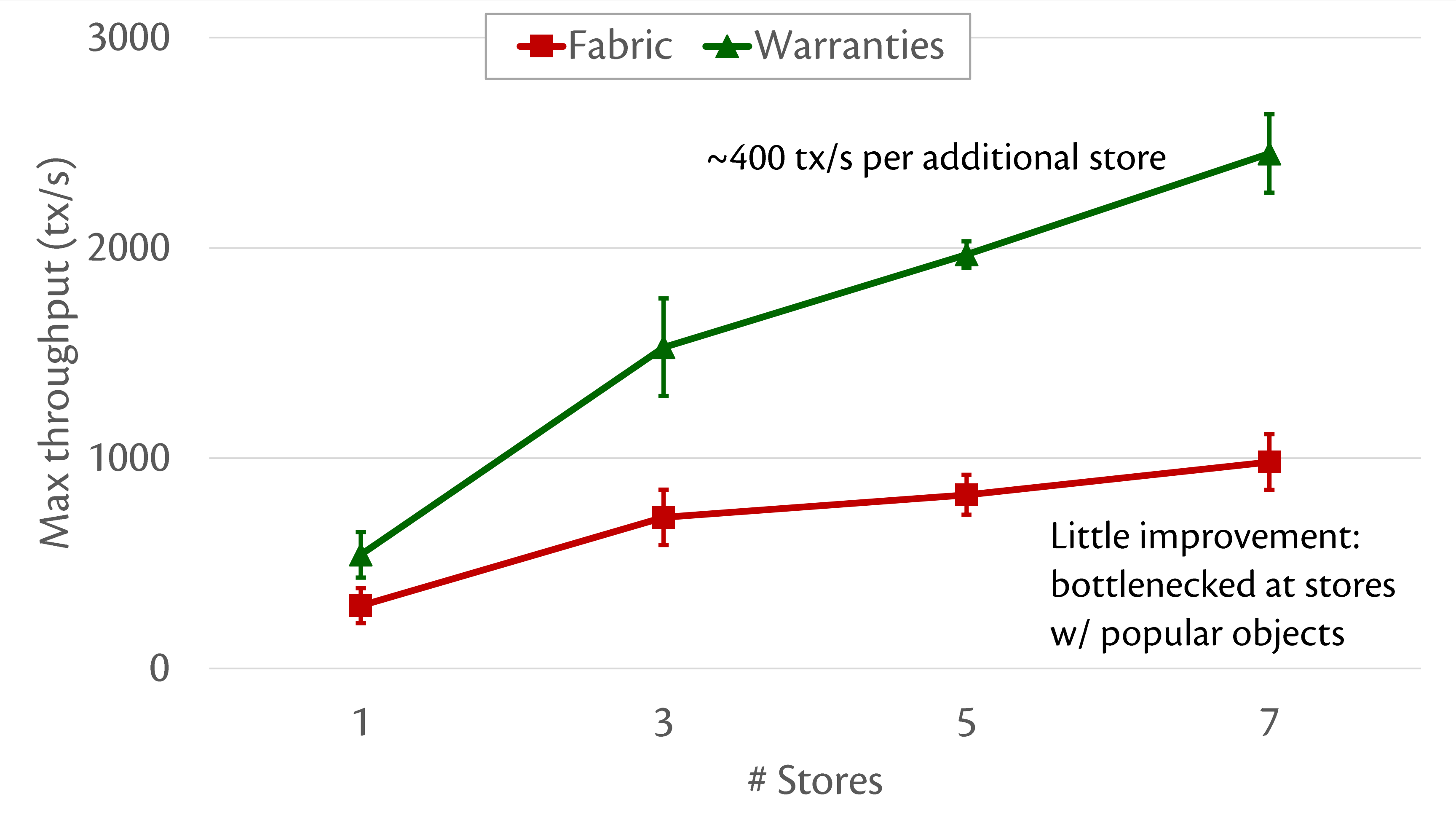
Evaluation: state warranties

- Multiuser OO7 benchmark
  - Models OODBMS applications
  - Heavyweight transactions (~460 objects involved)
- Changed to model popularity of reads (power law)
  - Increases read/write contention (harder to scale)

- Ran on Eucalyptus cluster
  - Stores: 2 cores, 8 GB memory
  - Clients: 4 cores, 16 GB memory

Scalability

- 2% writes
- 36 clients

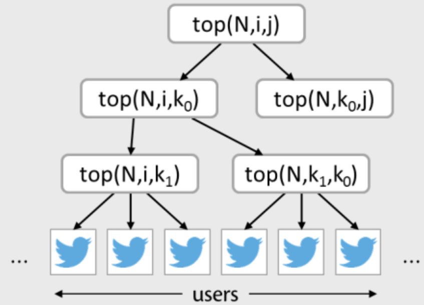


Effect of read/write ratios

- 3 stores
- 24 clients

Evaluation: computation warranties

- Twitter benchmark
  - 1,000 users
  - 98% reads (compute top-5 users)
  - 2% writes (follow/unfollow random user)



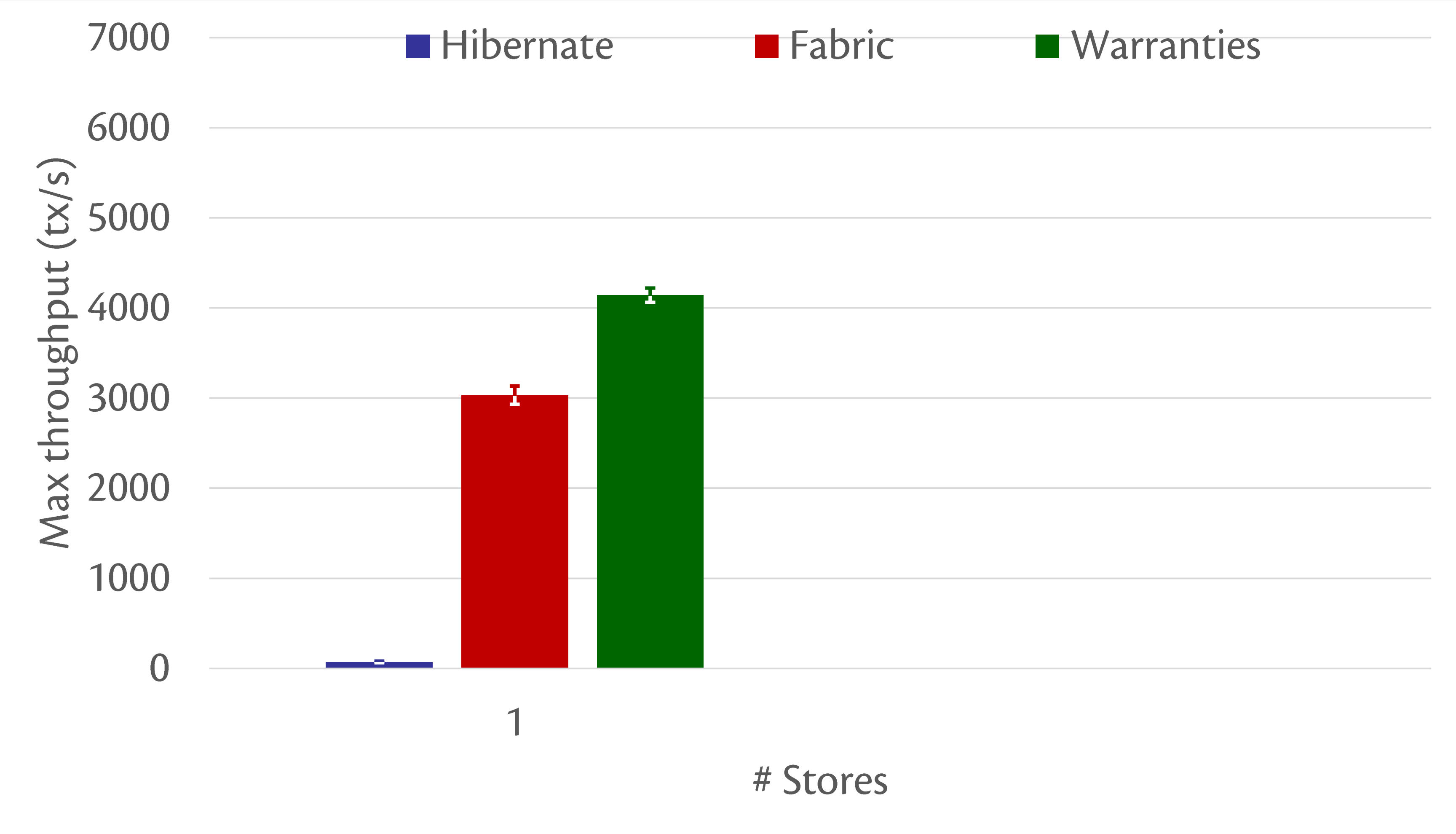
	Throughput (tx/s)	Median latency (ms)	95th percentile write delay (ms)
Fabric	17 ± 4	568 ± 354	—
State warranties	26 ± 5	1239 ± 455	623 ± 274
Comp. warranties	**343 ± 10**	**12 ± 2**	**16 ± 4**

Speedup by giving application-specific consistency

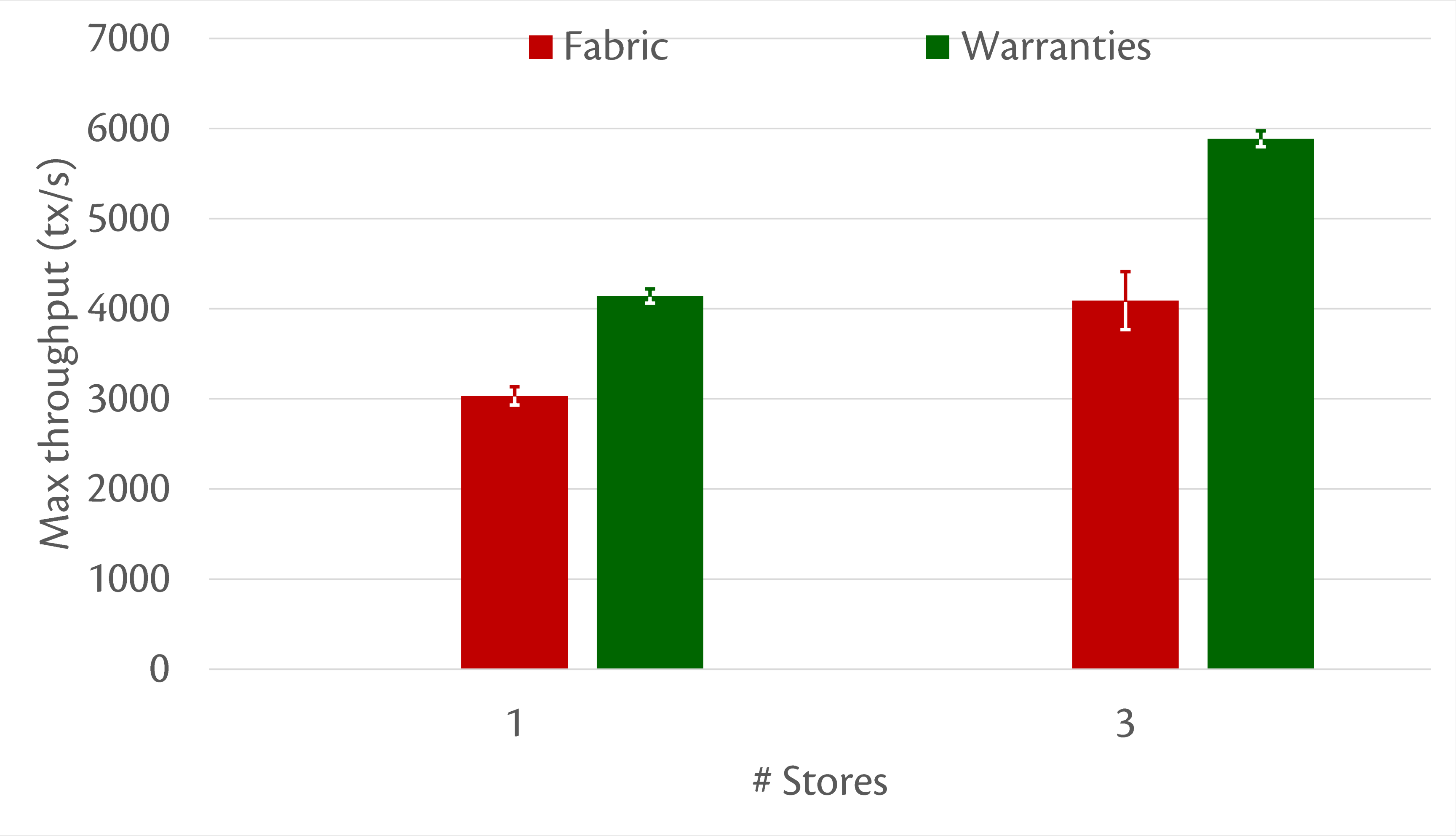
Evaluation: Cornell CS CMS

- Web app for managing assignments & grading
- Ported to Hibernate (JPA implementation)
  - Hibernate: popular ORM library for building web apps
  - Ran in "optimistic locking" mode
    - Emerging best practice
- Also ported to Fabric
- Workload based on 3-week trace from production CMS in 2013

CMS throughput



CMS scalability



Related work

- Promises [JFG 2007] generalize leases
  - Specify resource requirements w/ logical formulas
  - Given time-limited guarantees about resource availability
- Spanner [CDE+ 2012] – distributed transaction system w/ strict serializability
  - Lower level programming model, no computation caching
- TxCache [PCZML 2010] – application cache w/ transactional consistency
  - Weaker consistency model
- Escrow transactions [O'Neil 1986]
  - Transactions can commit when predicate on state is satisfied
  - Focused on allowing updates to commit more frequently

Warranties is the first to provide strong consistency by defending client caches

Warranties

for Faster Strong Consistency

Jed Liu Tom Magrino Owen Arden Michael D. George Andrew C. Myers

FABRIC

Cornell University
Department of Computer Science

Warranties help bridge the gap between consistency and scalability

- Defend client caches
- Commits avoid communication
- Strict serializability
- External consistency