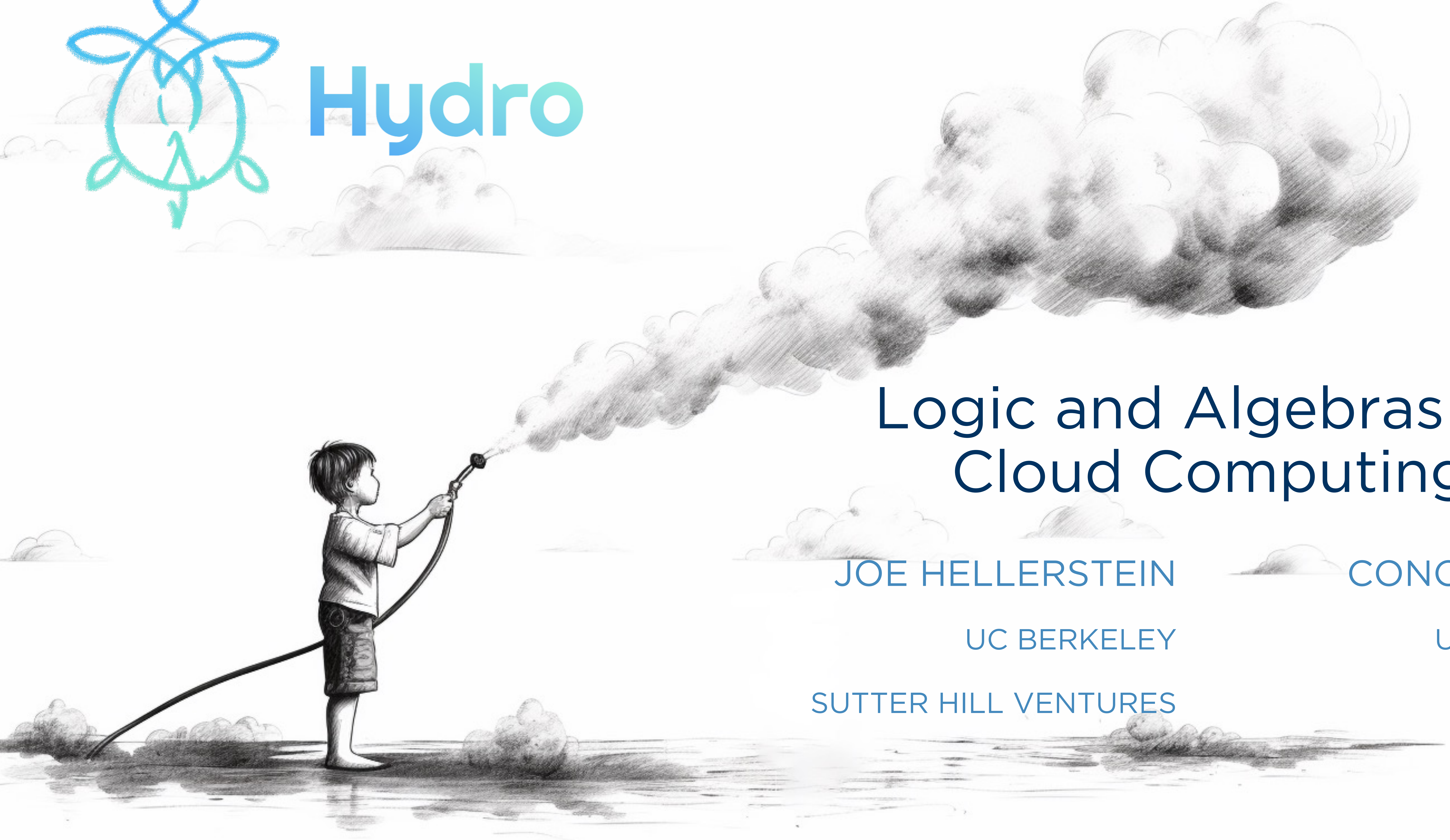


Hydro

Logic and Algebras for Cloud Computing

JOE HELLERSTEIN

UC BERKELEY

SUTTER HILL VENTURES

CONOR POWER

UC BERKELEY

sky

Sea Changes in Computing

Minicomputers

PDP-11, 1970

Supercomputers

Cray-1, 1976

Personal Computers

Macintosh, 1984

Smart Phones

iPhone, 2007

New Platform + New Language = Innovation

THE C PROGRAMMING LANGUAGE

Minicomputers

PDP-11, 1970

CRAY CHAPEL

MPI

Supercomputers

Cray-1, 1976

Personal Computers

Macintosh, 1984

HyperCard

NATIONAL INSTRUMENTS LabVIEW

Smart Phones

iPhone, 2007

Swift

The Big Question

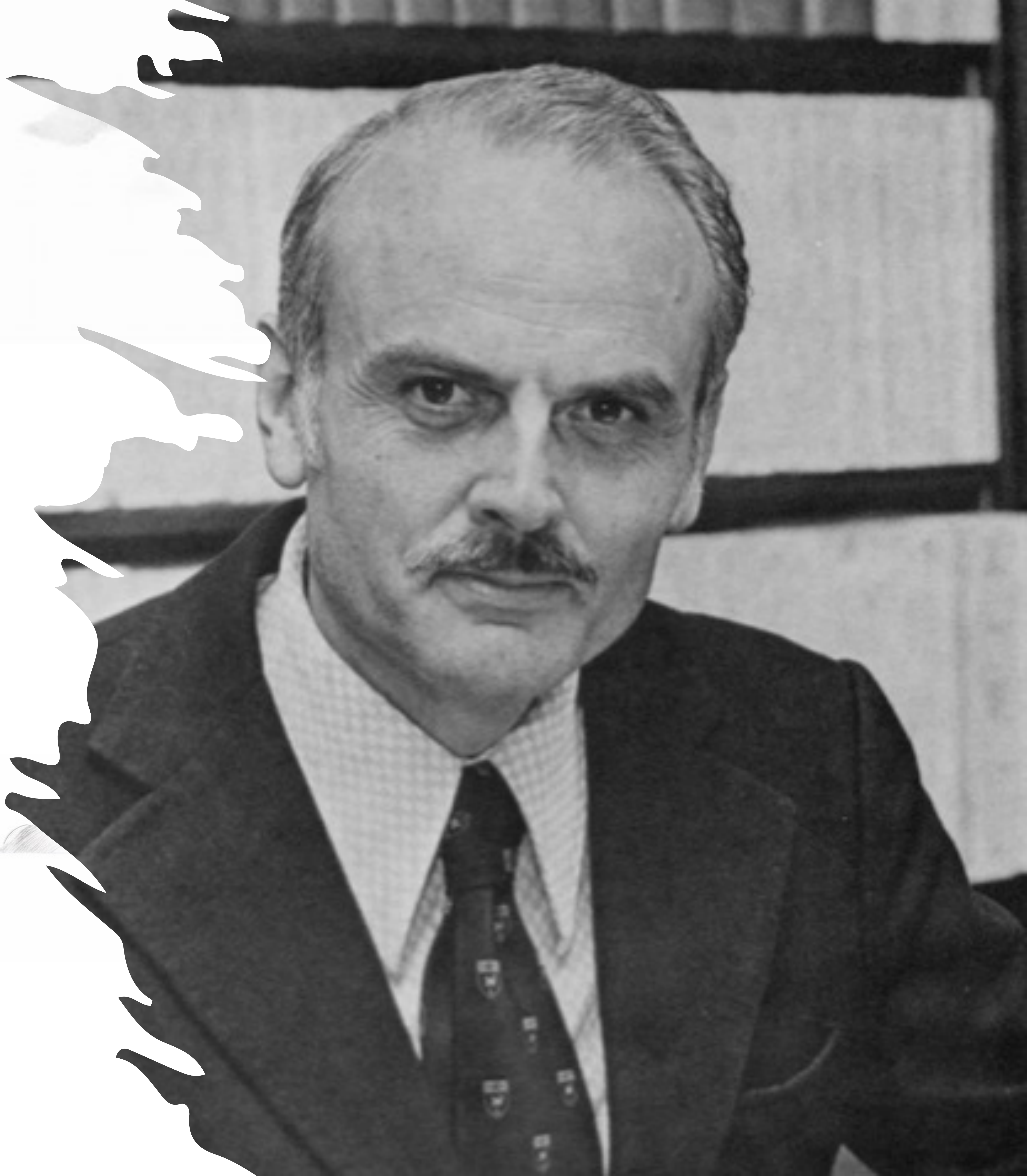
- How will folks program the cloud?
  - In a way that fosters unexpected innovation
  - Distributed programming is hard!
    - Parallelism, consistency, partial failure, …
  - Autoscaling makes it harder!

Programming the Cloud: A Grand Challenge for Computing

- Today’s compilers don’t address distributed concerns

Ted Codd

Turing Award 1981

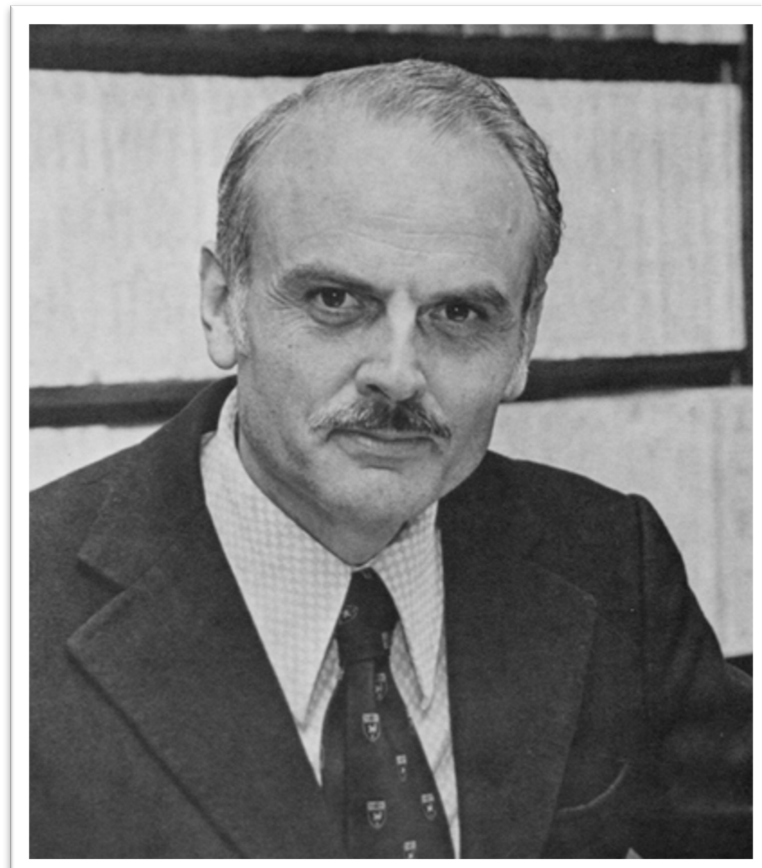


Formalize specification;

automate implementation.

Long-Running Agendas, Recent Trends

- Declarative Networking
- Relational Machine Learning
- Compiler Analysis
- …



Long-Running Agendas, Recent Trends

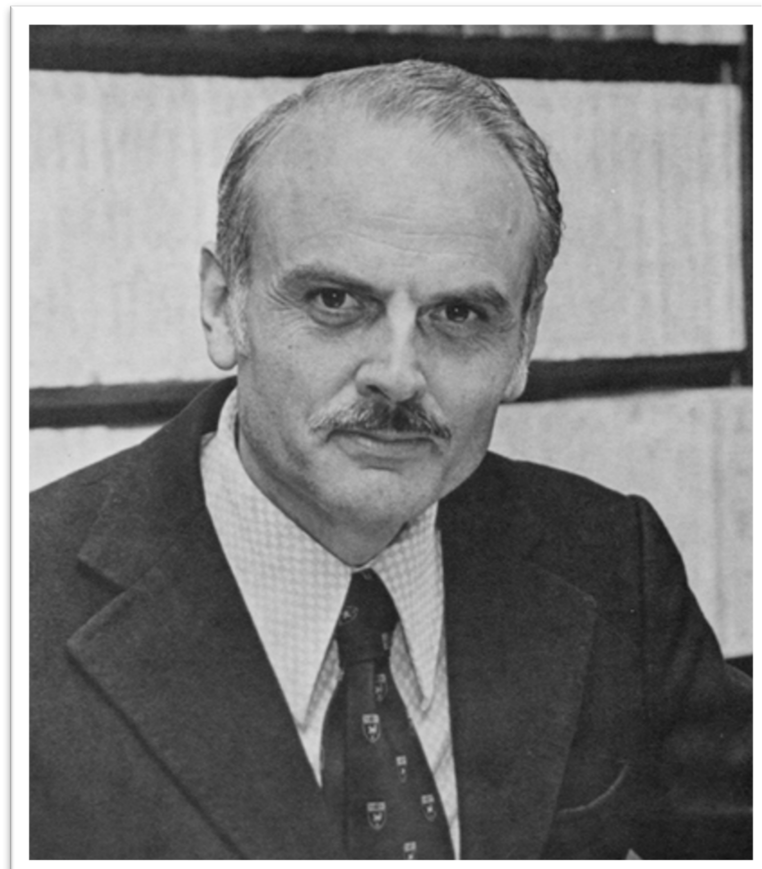
- Declarative Networking
- Relational Machine Learning
- Compiler Analysis
- ...

Trend: Logic → Algebra

Semi-Lattices

Semi-Rings

Abelian Groups



Declarative Programming for the Cloud

The cloud was invented

to hide how computing resources are laid out

and how computations are executed.

Relational databases were invented

to hide how data is laid out

and how queries are executed.


Hydro

Goals

LLVM for the Cloud?

- A language/compiler/debugger that addresses distributed concerns!
  - Is my program consistent or will different machines disagree?
  - How can I partition state safely?
  - What failures can this tolerate and how many?
  - What data is going where and who can see it?
  - Tunable objective functions. Please optimize for:
    - $$, not latency.
    - 99'th percentile, not 95th
    - Etc.

LLVM for the Cloud?

- A language/compiler/debugger that addresses distributed concerns!
  - Is my program consistent or will different machines disagree?
  - How can I partition state safely?
  - What failures can this tolerate and how many?
  - What data is going where and who can see it?
  - Tunable objective functions. Please optimize for:
    - $$, not latency.
    - 99'th percentile, not 95th
    - Etc.

Hydro

- A language/compiler/debugger that addresses distributed concerns!
  - Is my program consistent or will different machines disagree?
  - How can I partition state safely?
  - What failures can this tolerate and how many?
  - What data is going where and who can see it?
  - Tunable objective functions. Please optimize fo
    - $$, not latency.
    - 99'th percentile, not 95th
    - Etc.

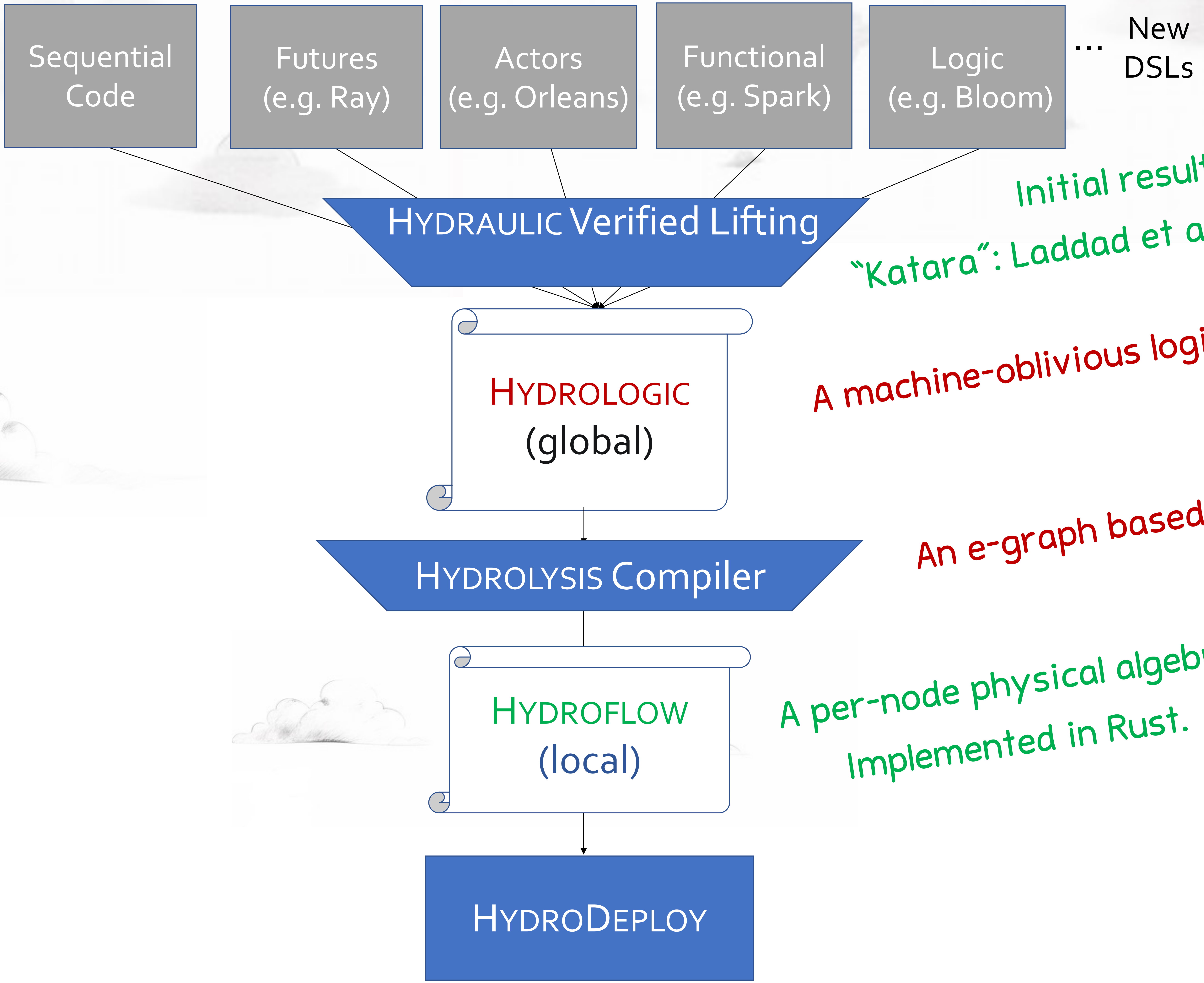
New Directions in Cloud Programming

Alvin Cheung   Natacha Crooks   Joseph M. Hellerstein   Matthew Milano

CIDR 21

Hydro

HYDRO Stack



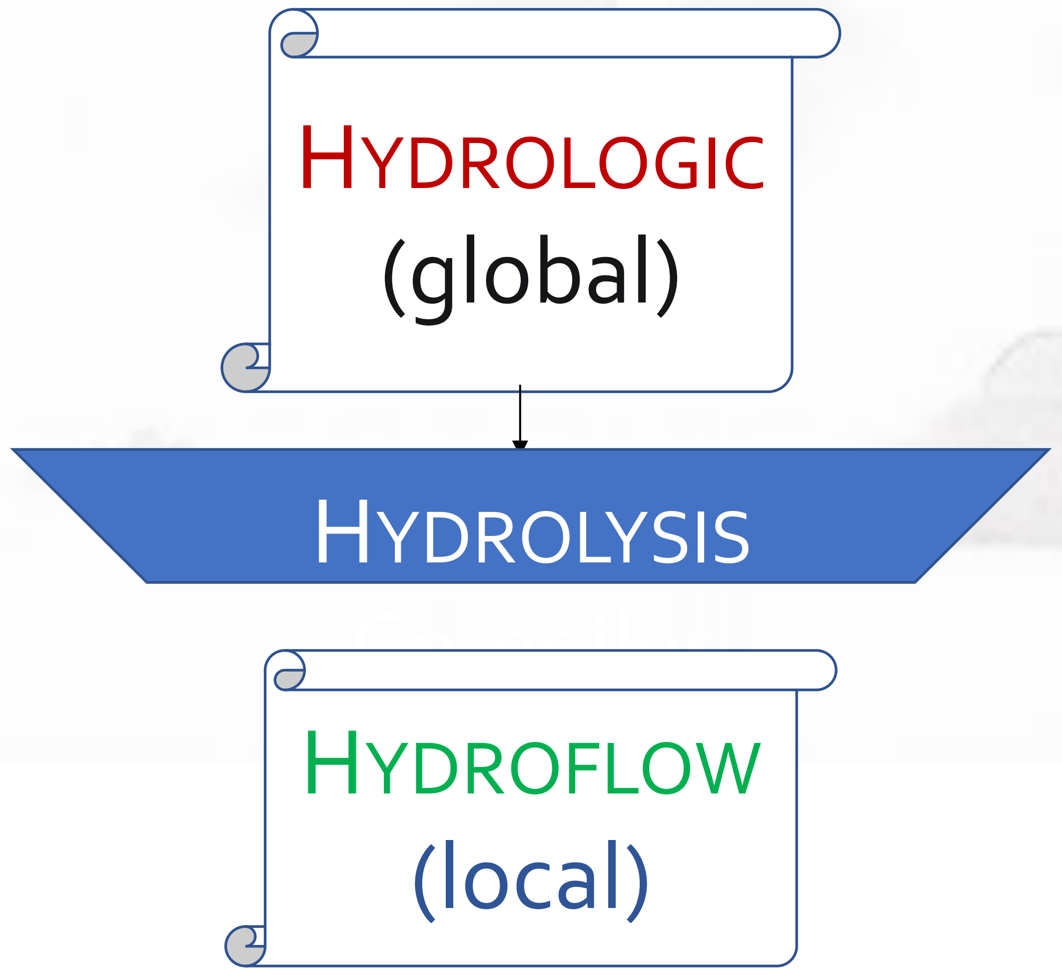
Initial results
"Katara": Laddad et al. OOPSLA 22

A machine-oblivious logic (or algebra)?

An e-graph based "optimizer"

A per-node physical algebra
Implemented in Rust.

Topics for Today (and WIP)



- Automatic Replication (of code and data)
  - Esp. “free” replication — consistency sans coordination (CALM)

    (algebraic CALM Theorem)
- Termination detection
  - Esp. “free” termination — detection sans coordination

    *(threshold morphisms and **equivalences**)*
- Automatic partitioning (of code and data)
  - Esp. “free” partitioning — parallel execution sans coordination

    *(functional dependencies **integrated into algebraic types**)*

And excellent performance!
(vs hand-written C++)


Hydro


Hydro

Language/Theory Work: 2010-15

- Formalism: **Dedalus**

DEDALUS: Datalog in Time and Space

Peter Alvaro[1], William R. Marczak[1], Neil Conway[1], Joseph M. Hellerstein[1], David Maier[2], and Russell Sears[3]

Datalog Reloaded 2010

A Declarative Semantics for Dedalus

Peter Alvaro
Tom J. Ameloot
Joseph M. Hellerstein
William Marczak
Jan Van den Bussche

TR 2011

- **CALM Theorem**: coordination in its place
  - Consistency ⇔ Monotonicity

JMH PODS Keynote, 2010
Ameloot, et al PODS 2011
Ameloot et al. TODS 2016
Alvaro/Hellerstein CACM 2020

DOI:10.1145/3369736

In distributed systems theory, CALM presents a result that delineates the frontier of the possible.

BY JOSEPH M. HELLERSTEIN AND PETER ALVARO

Keeping CALM: When Distributed Consistency Is Easy

DISTRIBUTED SYSTEMS ARE tricky. Multiple unreliable

across many machines. Most scientific computing and machine learning systems work in parallel across multiple processors. Even legacy desktop operating systems and applications like spreadsheets and word processors are tightly integrated with distributed backend services.

The challenge of building correct distributed systems is increasingly urgent, but it is not new. One traditional answer has been to reduce this complexity with *memory consistency* guarantees—assurances that accesses to memory (heap variables, database keys, and so on) occur in a controlled fashion. However, the mechanisms used to enforce these guarantees—*coordination protocols*—are often criticized as barriers to high performance, scale, and availability of distributed systems.

The high cost of coordination. Coordination protocols enable autonomous, loosely coupled machines to jointly decide how to control basic behaviors, including the order of access to shared memory. These protocols are among the most clever and widely cited ideas in distributed computing. Some well-known techniques include the Paxos[33] and Two-Phase Commit (2PC)[25,34] protocols, and global barriers underly-

» key insights

- **Coordination is often a limiting factor in system performance. While sometimes necessary for consistent outcomes,**

- <~ bloom: Logic + Lattices

  w/stratified neg/agg, morphisms, semi-naïve eval on lattices, etc

SOCC 2012

Logic and Lattices for Distributed Programming

Neil Conway, UC Berkeley, nrc@cs.berkeley.edu
William R. Marczak, UC Berkeley, wrm@cs.berkeley.edu
Peter Alvaro, UC Berkeley, palvaro@cs.berkeley.edu
Joseph M. Hellerstein, UC Berkeley, hellerstein@cs.berkeley.edu
David Maier, Portland State University, maier@cs.pdx.edu

Systems Work: 2015-2021

- Cloudburst: Stateful FaaS
- Compartmentalized Paxos
- Lineage Driven Fault Injection
- Why-Across-Time Provenance

VLDB 2020

Cloudburst: Stateful Functions-as-a-Service

Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, Alexey Tumanov†
U.C. Berkeley, †Georgia Tech

VLDB 2021

Scaling Replicated State Machines with Compartmentalization

Michael Whittaker, UC Berkeley, mjwhittaker@berkeley.edu
Ailidani Ailijiang, Microsoft, aiailiji@microsoft.com
Aleksey Charapko, University of New Hampshire, aleksey.charapko@unh.edu
Murat Demirbas, University at Buffalo, demirbas@buffalo.edu
Neil Giridharan, UC Berkeley, giridhn@berkeley.edu
Joseph M. Hellerstein, UC Berkeley, hellerstein@berkeley.edu
Heidi Howard, University of Cambridge, hh360@cst.cam.ac.uk
Ion Stoica, UC Berkeley, istoica@berkeley.edu
Adriana Szekeres, VMWare, aszekeres@vmware.com

SIGMOD 2015

Lineage-driven Fault Injection

Peter Alvaro, UC Berkeley, palvaro@cs.berkeley.edu
Joshua Rosen, UC Berkeley, rosenville@gmail.com
Joseph M. Hellerstein, UC Berkeley, hellerstein@cs.berkeley.edu

SOCC 2018

Debugging Distributed Systems with Why-Across-Time Provenance

Michael Whittaker, UC Berkeley, mjwhittaker@berkeley.edu
Cristina Teodoropol, UC Berkeley, ct@berkeley.edu
Peter Alvaro, UC Santa Cruz, palvaro@ucsc.edu
Joseph M. Hellerstein, UC Berkeley, hellerstein@berkeley.edu

Systems Highlight: Anna Key-Value Store

- KVS: Petri dish of distributed systems!
- "CALM" Semi-lattice Design
  - Monotonic ⇒ Freely Replicable (w/o coordination)
  - Update anywhere, gossip lazily
  - **Zero** concurrency control (locks, atomics, protocols)

Anna: A KVS For Any Scale

Chenggang Wu [#1], Jose M. Faleiro [*2], Yihan Lin [**3], Joseph M. Hellerstein [#4]

ICDE 18

Autoscaling Tiered Cloud Storage in Anna

Chenggang Wu, Vikram Sreekanti, Joseph M. Hellerstein

VLDB 19

Systems Highlight: Anna Key-Value Store

- KVS: Petri dish of distributed systems!
- “CALM” Semi-lattice Design
  - Monotonic ⇒ Freely Replicable (w/o coordination)
  - Update anywhere, gossip lazily
  - **Zero** concurrency control (locks, atomics, protocols)

2022 SIGMOD Jim Gray Doctoral Dissertation Award

ACM SIGMOD is pleased to present the 2022 SIGMOD Jim Gray Doctoral Dissertation Award to Chenggang Wu.

Chenggang Wu is Co-founder and CTO at Aqueduct, a SaaS startup building machine learning prediction infrastructure. He received his Ph.D. in 2020 from UC Berkeley, advised by Joseph M. Hellerstein. He is the recipient of best-of-conference citations for research appearing in both VLDB 2019 and ICDE 2018. He frequently serves as a PC member and a reviewer for conferences and journals such as SIGMOD, ICDE, VLDBJ, and TKDE. Chenggang’s Ph.D. dissertation develops design principles for building serverless infrastructure that can achieve excellent performance, seamless scalability, and rich consistency guarantees. The dissertation proposes two key ideas that are fundamental to achieving the combination of these goals: lattice-based coordination-free consistency, and LDPC (logical disaggregation with physical colocation). These ideas

Anna: A KVS For Any Scale

Chenggang Wu [#1], Jose M. Faleiro [*2], Yihan Lin [**3], Joseph M. Hellerstein [#4]

ICDE 18

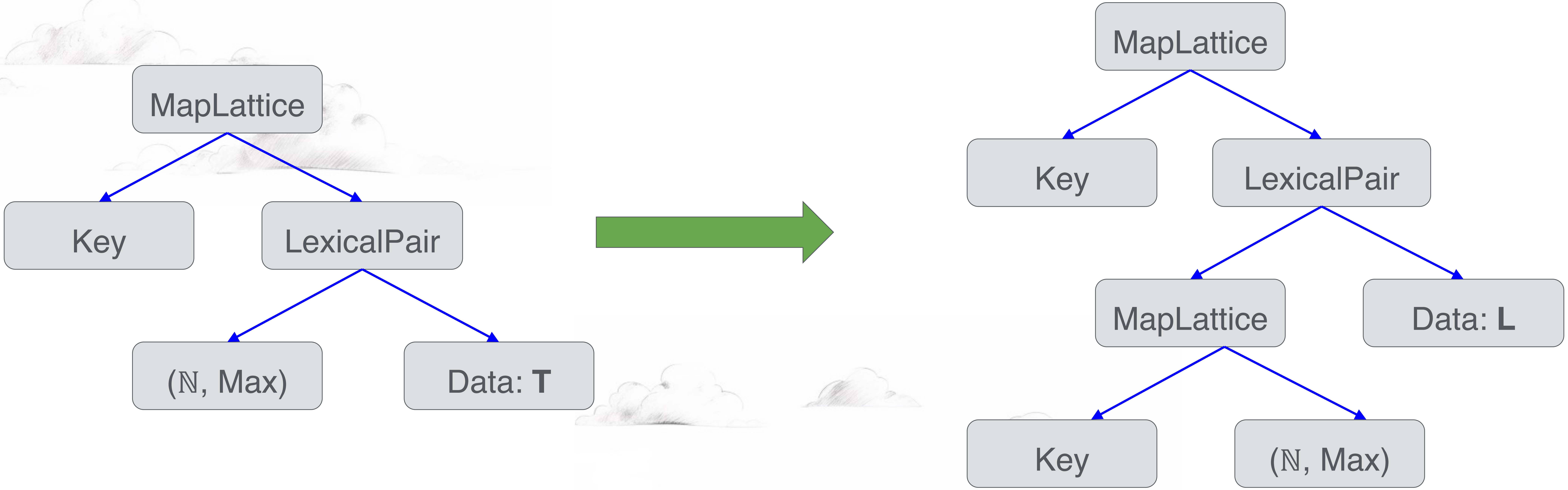
Autoscaling Tiered Cloud Storage in Anna

Chenggang Wu, Vikram Sreekanti, Joseph M. Hellerstein

VLDB 19

Examples of lattice composition

- Metadata “wrappers” for various replica consistency mechanisms

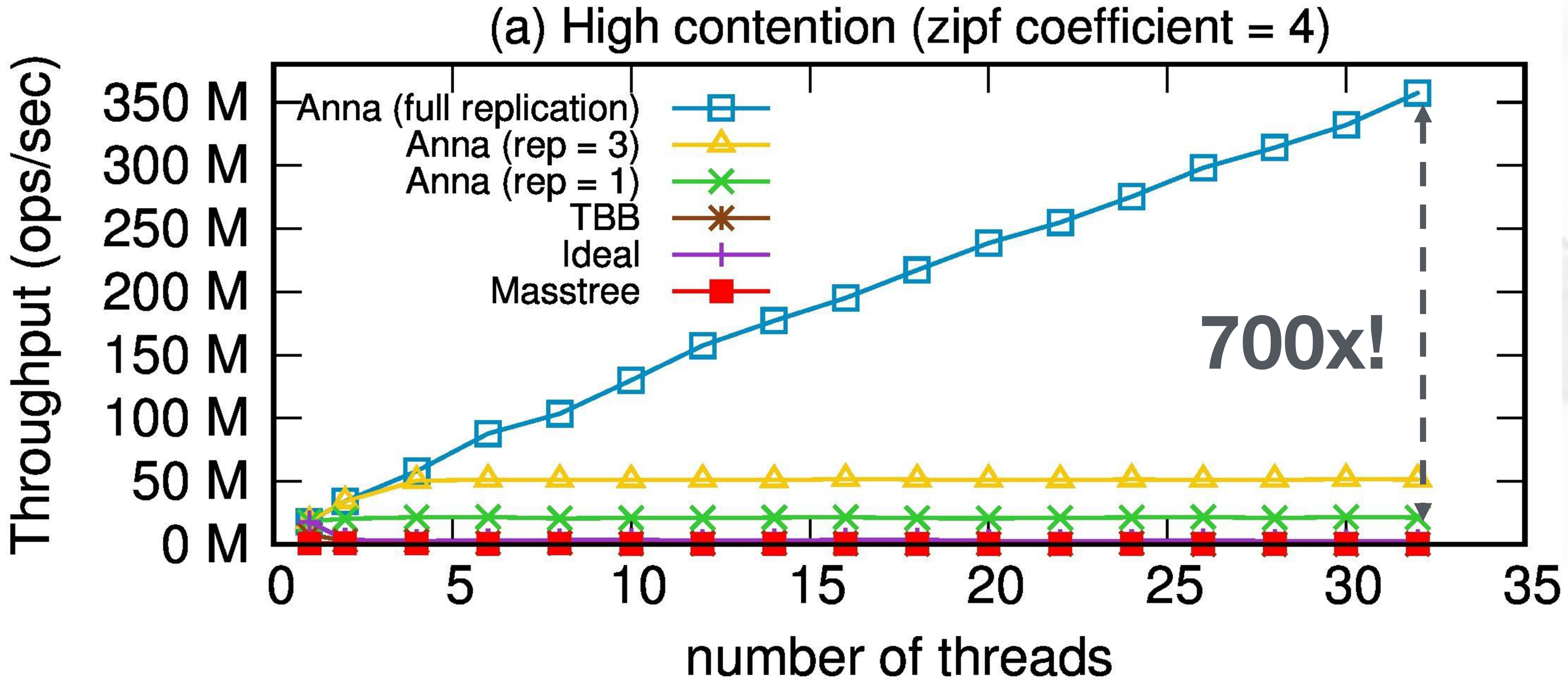


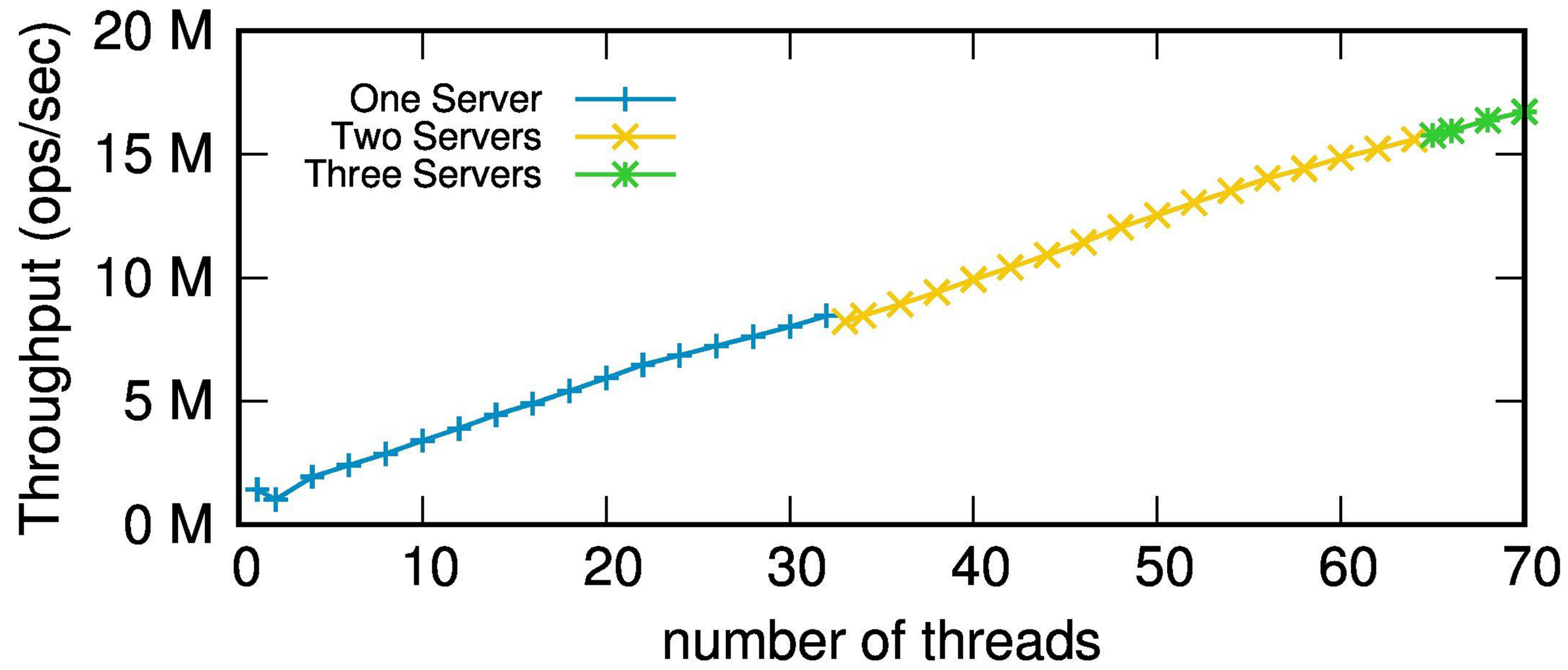
Last Writer Wins

Causal Consistency

Anna KVS
Performance + Consistency

- Fast, especially under contention
  - Up to **700x faster than Masstree and Intel TBB** on multicore
  - Up to **10x faster than Cassandra** in a geo-deployment
  - **350x the performance of DynamoDB** for the same price





KVS	Request Handling	Atomic Instruction
Anna (full)	90%	0%
Anna (rep=3)	91%	0%
Anna (rep=1)	94%	0%
Ideal	97%	0%
TBB	4%	95%
Masstree	7%	92%

Hand-written in C++ for a Ph.D. dissertation.

Implementation correct by assertion.

Can we formalize and maintain speed?

Formalisms for Distributed Correctness

- Desired: type system or compiler guarantee
- Starting from a "trusted base"
  - **Basic semi-lattices**, e.g.
    - Sets: $(\mathcal{P}(T), \cup)$
    - Counters: $(\mathbb{N}, \max)$
  - **Composite semi-lattices**, e.g.
    - KeyValueMap,
    - Product, LexicalProduct (when possible)
  - **"Physical Algebra" of operators**, e.g.
    - $\sqcup$, $\times$, filter, map, fold
    - scan, "network", mux, demux, etc.

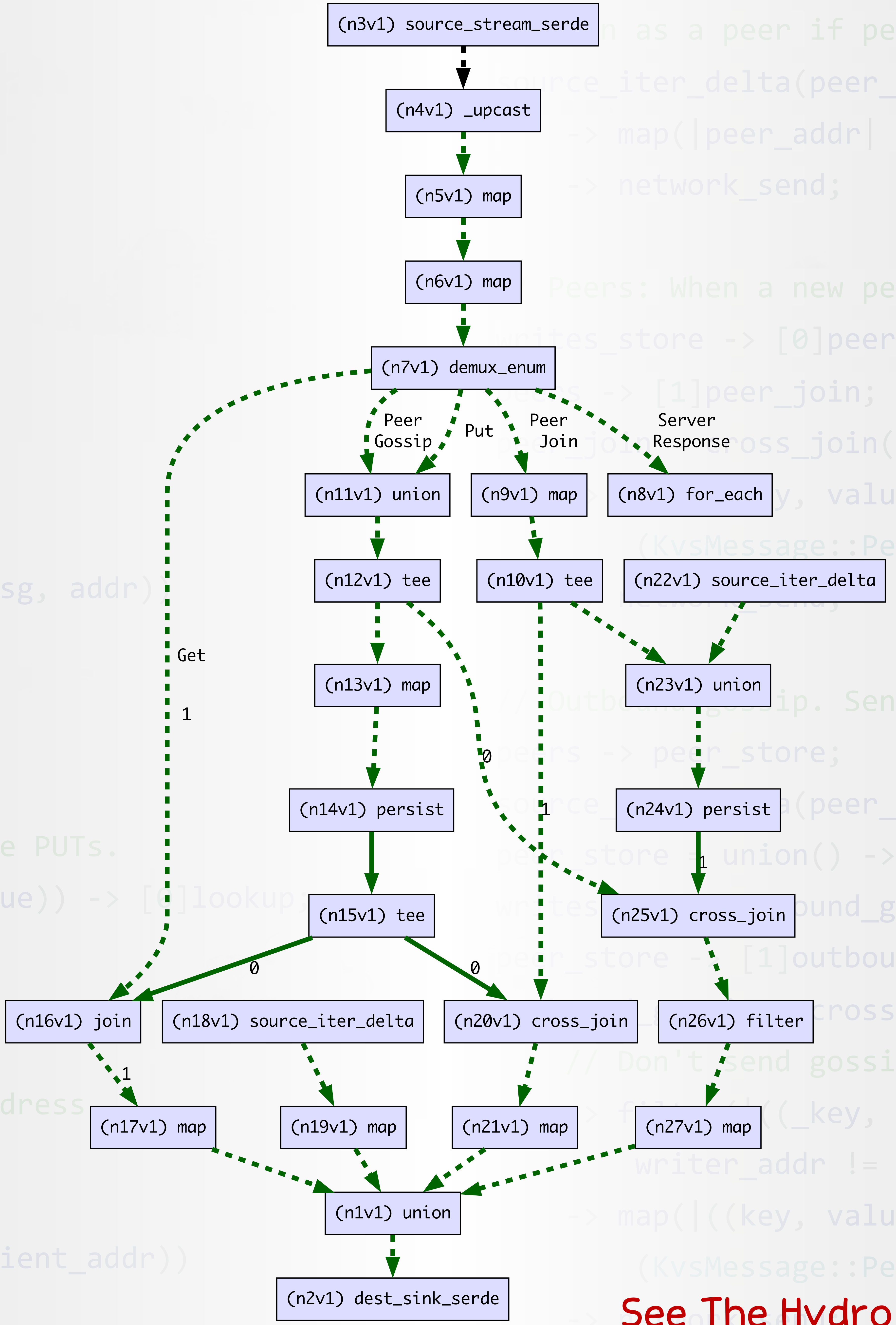
Anna in Hydroflow, a semi-lattice -inspired dataflow lang (semi-lattice “query plans”)

```
// Demux network inputs
network_recv = source_stream_serde(inbound)
    -> _upcast(Some(Delta))
    -> map(Result::unwrap)
    -> map(|(msg, addr)|
        KvsMessageWithAddr::from_message(msg, addr))
    -> demux_enum::<KvsMessageWithAddr>();
puts = network_recv[Put];
gets = network_recv[Get];

// Join PUTs and GETs by key, persisting the PUTs.
puts -> map(|(key, value, _addr)| (key, value)) -> [0]lookup;
gets -> [1]lookup;
lookup = join::<'static, 'tick>();

// Send GET responses back to the client address.
lookup
    -> map(|(key, (value, client_addr))|
            (KvsResponse { key, value }, client_addr))
    -> dest_sink_serde(outbound);
```

```
// Join as a peer if peer_server is set.
source_iter_delta(peer_server)
    -> map(|peer_addr| (KvsMessage::PeerJoin, peer_addr))
    -> network_send;

// Peers: When a new peer joins, send them all data.
writes_store -> [0]peer_join;
peers -> [1]peer_join;
peer_join = cross_join()
    -> map(|((key, value), peer_addr)|
        (KvsMessage::PeerGossip { key, value }, peer_addr))
    -> network_send;

// Outbound gossip. Send updates to peers.
peers -> peer_store;
source_iter_delta(peer_server) -> peer_store;
peer_store = union() -> persist();
writes -> [0]outbound_gossip;
peer_store -> [1]outbound_gossip;
outbound_gossip = cross_join()
    // Don't send gossip back to same sender.
    -> filter(|((_key, _value, writer_addr), peer_addr)|
        writer_addr != peer_addr)
    -> map(|((key, value, _writer_addr), peer_addr)|
        (KvsMessage::PeerGossip { key, value }, peer_addr))
    -> network_send;
```

In Hydroflow

```
// Demux network inputs
network_recv = source_stream_serde(inbound)
    -> _upcast(Some(Delta))
    -> map(Result::unwrap)
    -> map(|(msg, addr)|
        KvsMessageWithAddr::from_message(msg, addr))
    -> demux_enum::<KvsMessageWithAddr>();
puts = network_recv[Put];
gets = network_recv[Get];

// Join PUTs and GETs by key, persisting the PUTs.
puts -> map(|(key, value, _addr)| (key, value)) -> [0]lookup;
gets -> [1]lookup;
lookup = join::<'static, 'tick>();

// Send GET responses back to the client address.
lookup
    -> map(|(key, (value, client_addr))|
        (KvsResponse { key, value }, client_addr))
    -> dest_sink_serde(outbound);
```

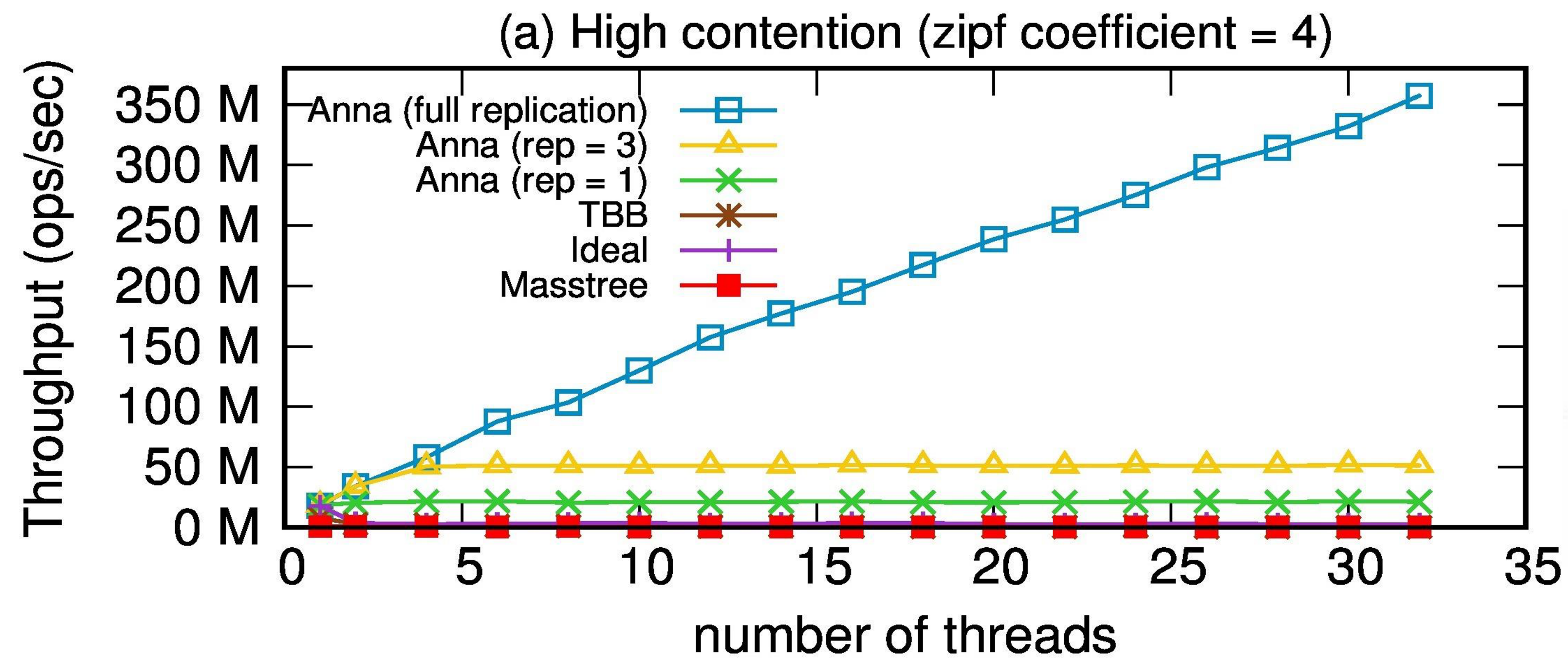


See The Hydro Book: https://hydro.run/docs/hydroflow/

Fast?

Original Anna KVS. C++

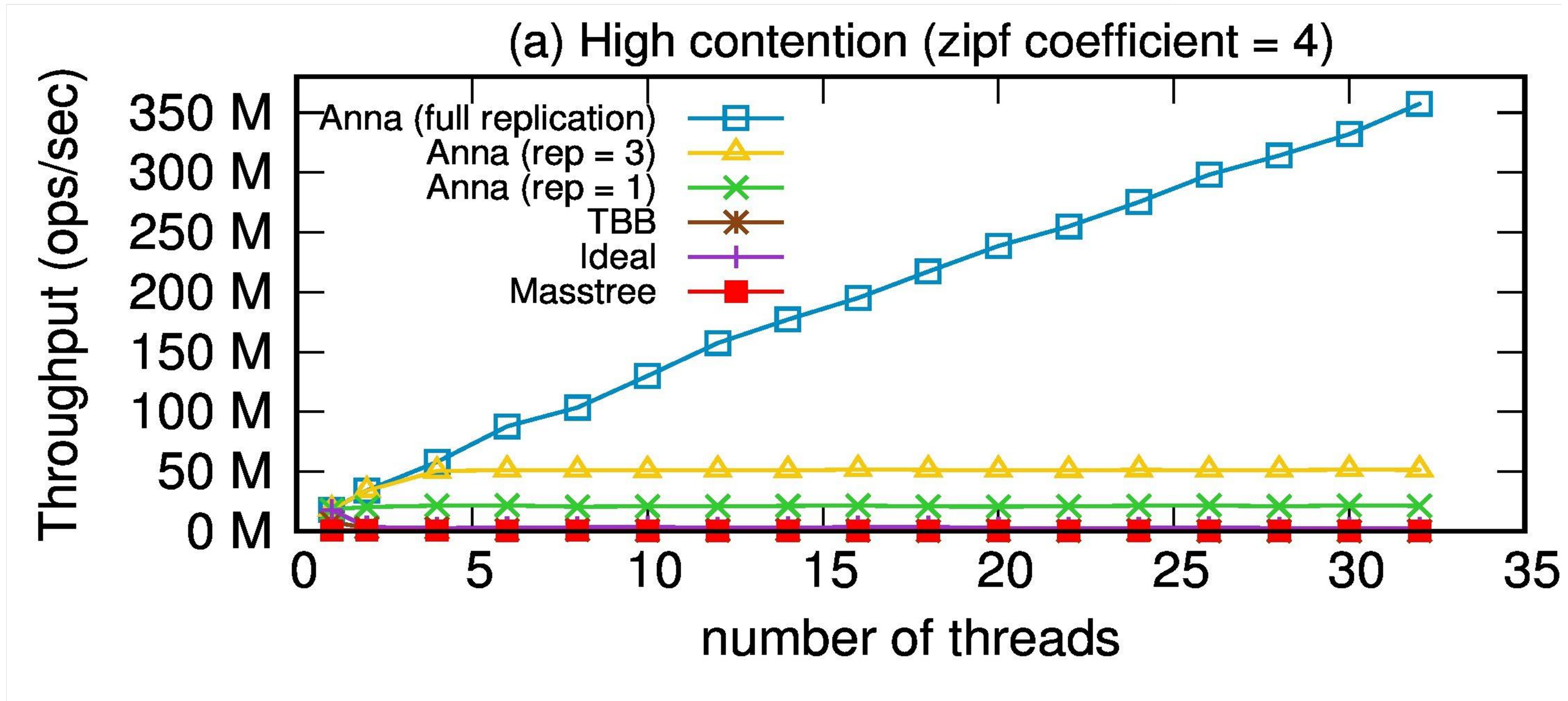
2018 Amazon m4.16xlarge instances
(64 vCPU, 256GB RAM,)



Fast? ✅

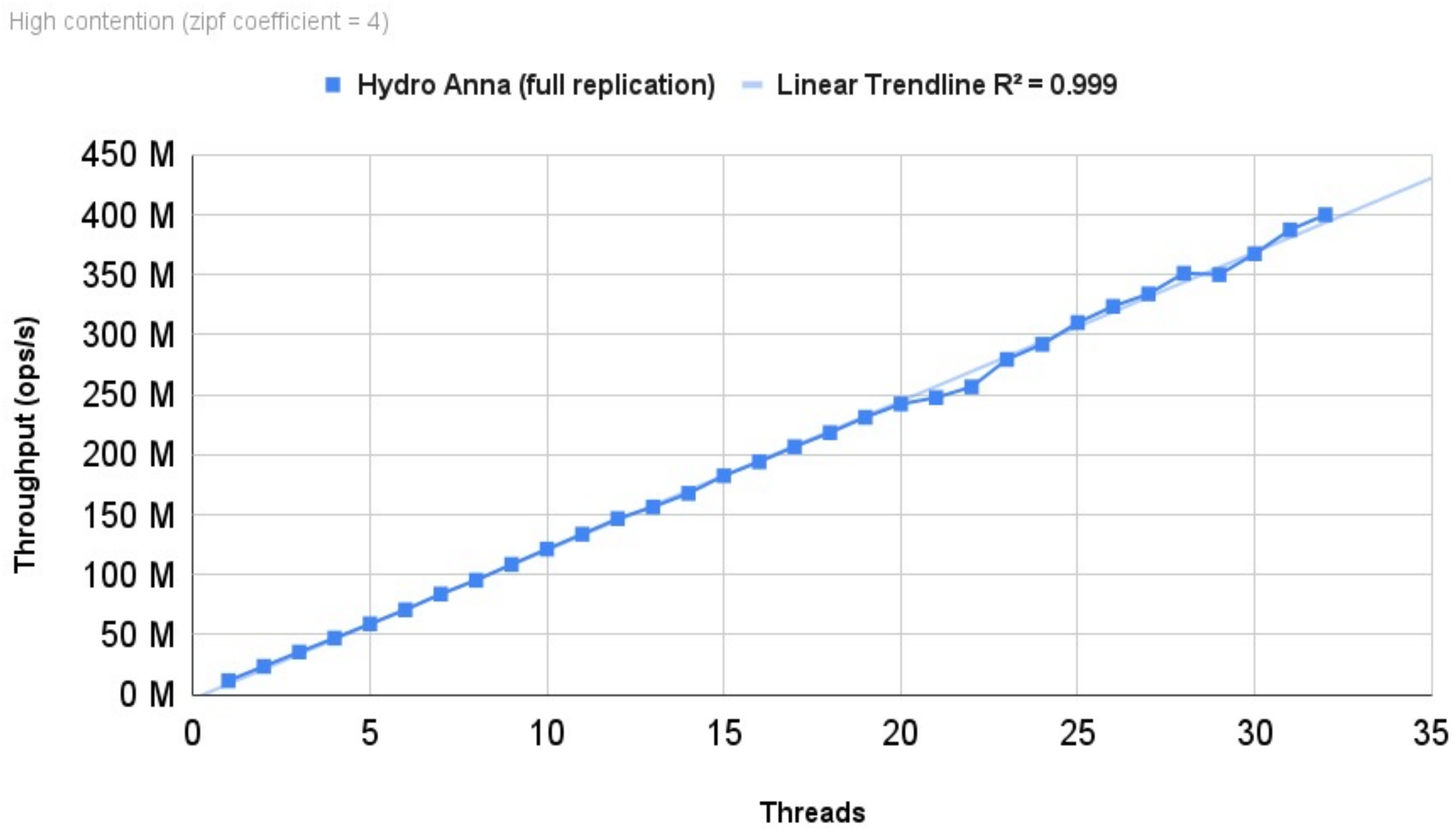
Original Anna KVS. C++

2018 Amazon m4.16xlarge instances
(64 vCPU, 256GB RAM,)

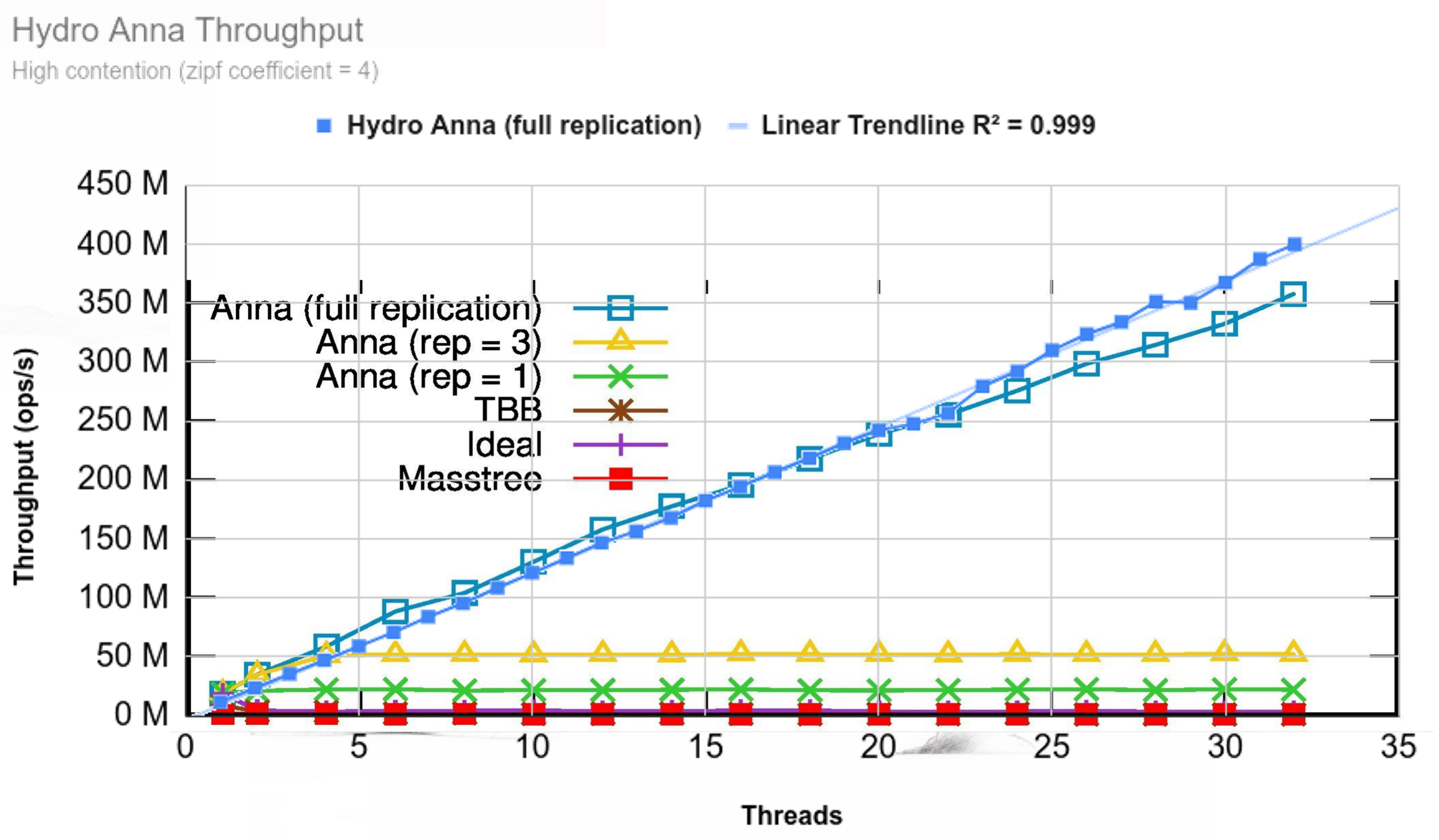


Anna KVS. Hydro

2023 GCP n2-standard-64 instances
(64 vCPU, 256GB RAM)



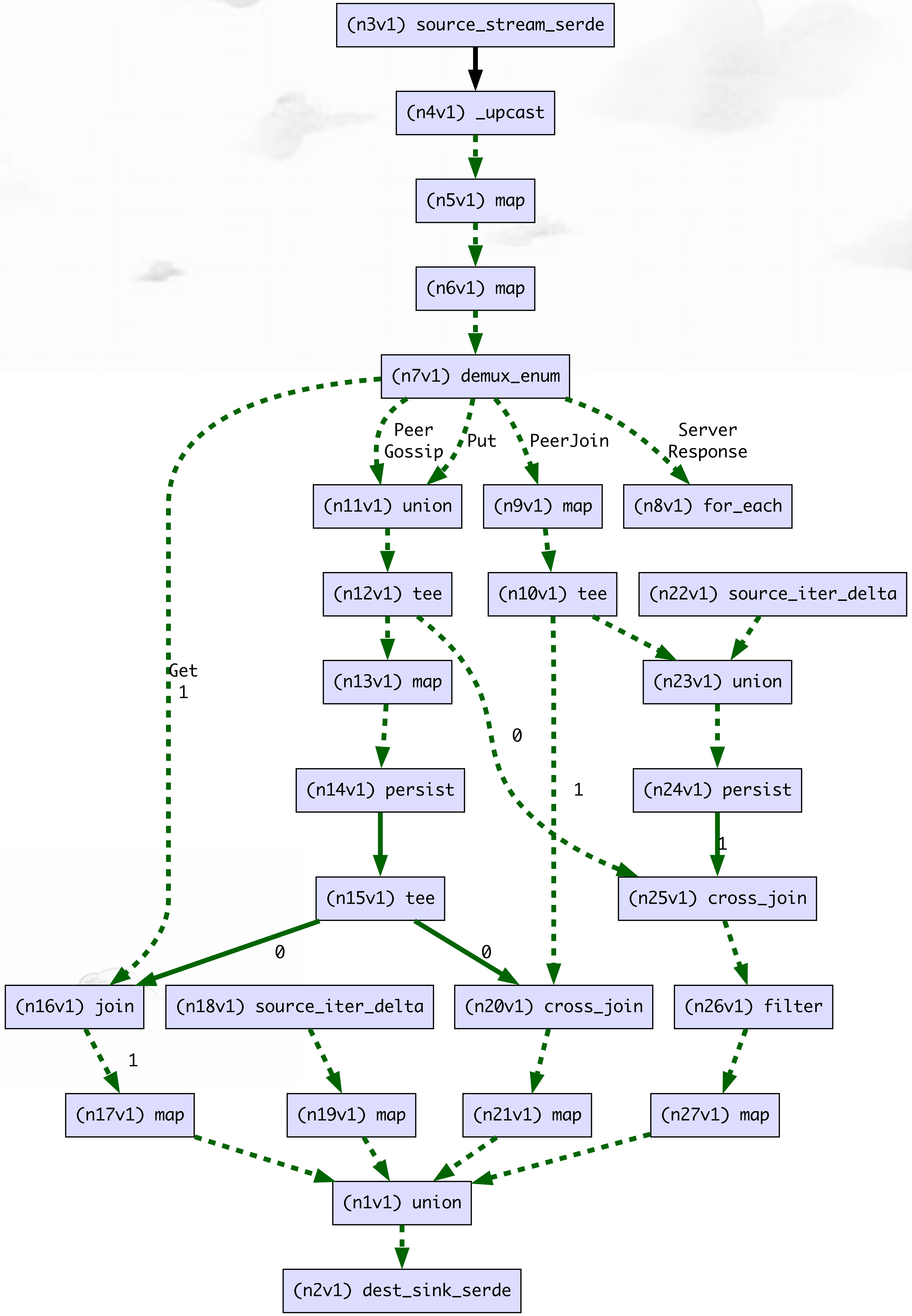
Fast? ✅



Hydro Anna Throughput
High contention (zipf coefficient = 4)
Hydro Anna (full replication)
Linear Trendline R^2 = 0.999
450 M
400 M
350 M
300 M
250 M
200 M
150 M
100 M
50 M
0 M
Throughput (ops/s)
0
5
10
15
20
25
30
35
Threads
Anna (full replication)
Anna (rep = 3)
Anna (rep = 1)
TBB
Ideal
Masstree

Consistently Replicable ✅

- At a glance!
- Sort of



A DBMS Lens on Cloud Programming

A Classical DBMS Lens

Decreasing declarativity, increasing implementation detail

Relational Calculus

=> Relational Algebra (SPJU…)

=> Physical Algebra (Scan, BtreeScan, Hashjoin, Sort, MergeJoin, etc.)

Good News / Bad News on the state of affairs

- Good news: Dedalus is a "Relational calculus" for distributed programming
  - Bad news: programmers don't like it. Can we leave the walled garden of logic?
  - Functional/algebraic expressions are more palatable (ie. they're in Python)
- Good news: An Algebra for distributed updates: Semi-Lattices/CRDTs
  - Bad news: they define updates on state, but no queries/functions
- Also, we can't ignore the shifting "physical" properties of data in motion
  - (Randomized) ordering, batching, duplication

Ideally

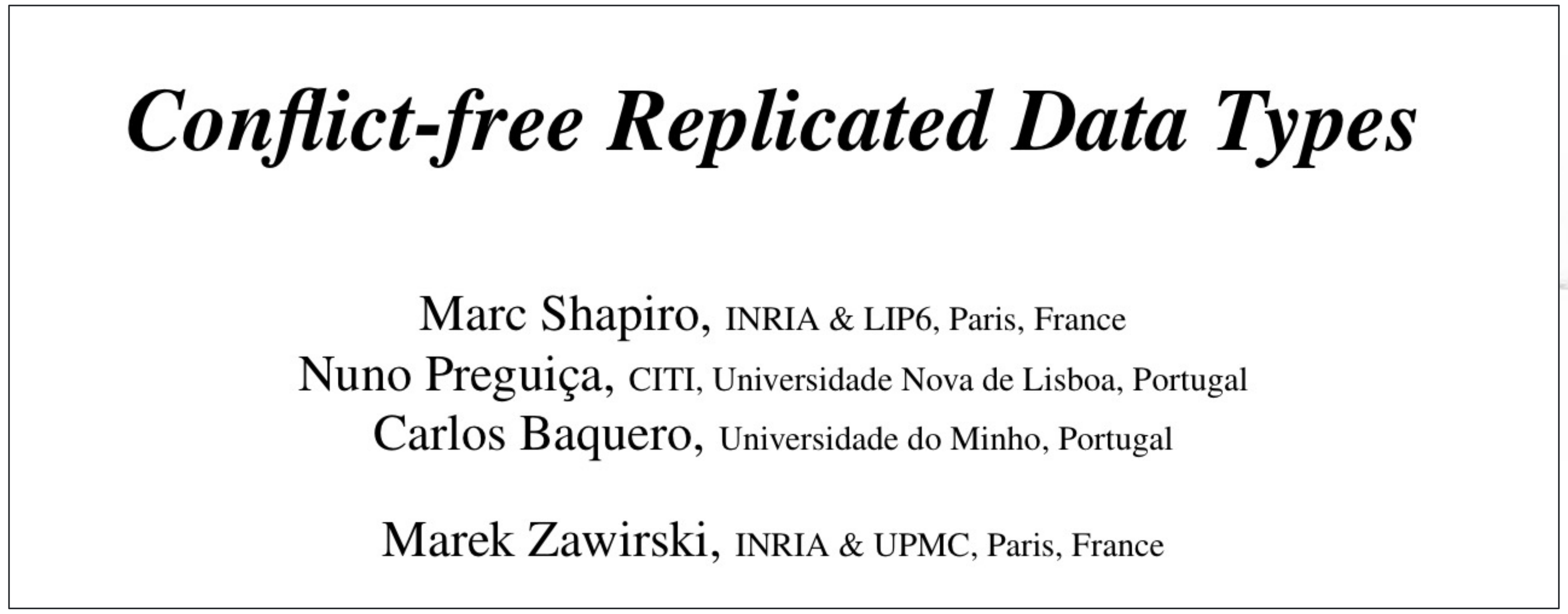
- Unify formalisms across Logic / Algebra / “Physical” Algebra
- Physical layer correctness proofs under network non-determinism
  - Physical algebra rich enough to capture “typical reality”
  - Correctness under replication, partitioning, batching, incrementalization
  - Analysis of termination

Algebras of Dataflow

Semi-Lattices / CRDTs

$(S, \sqcup)$

- Batching of Messages = Associativity
- Reordering of Messages = Commutativity
- Duplication of Messages = Idempotence

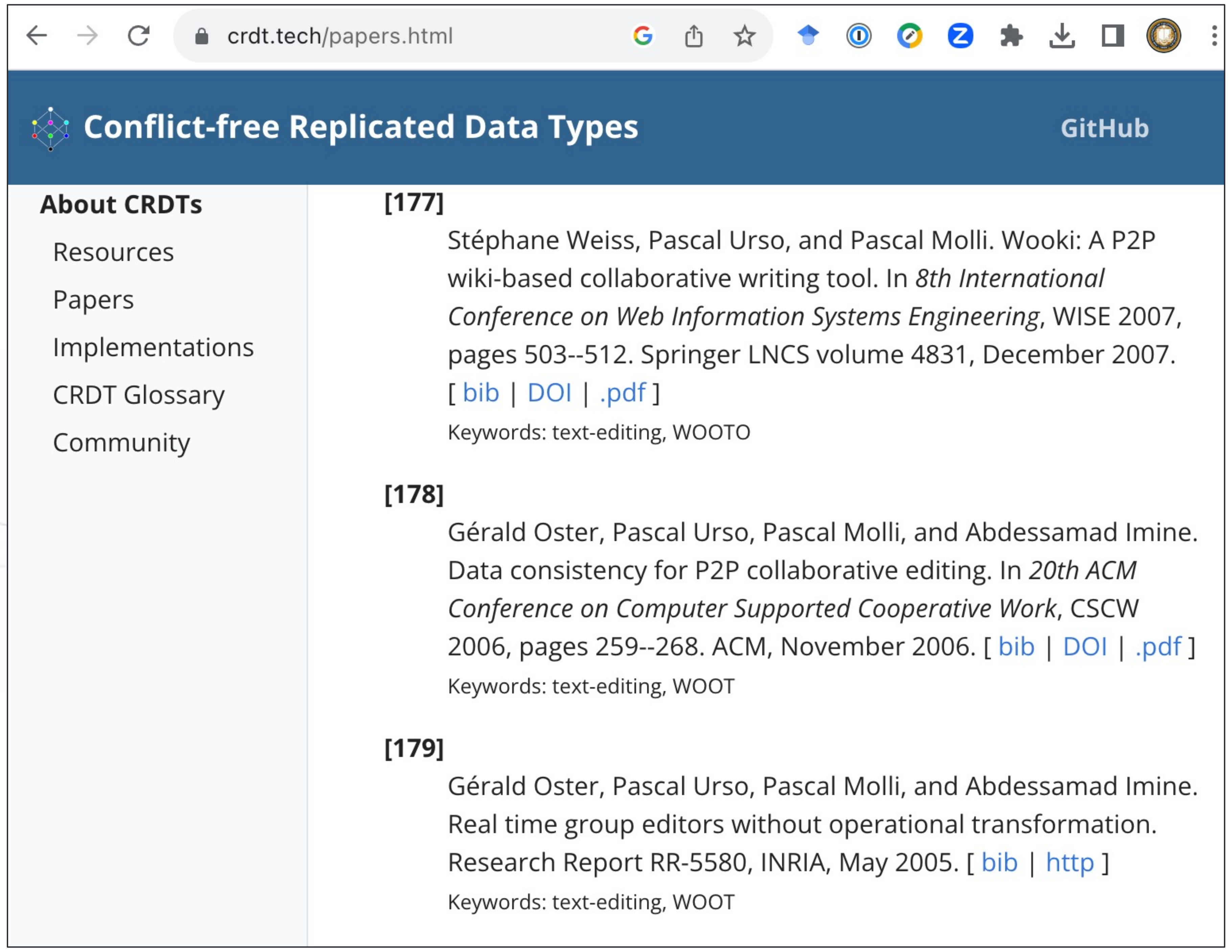


Conflict-free Replicated Data Types

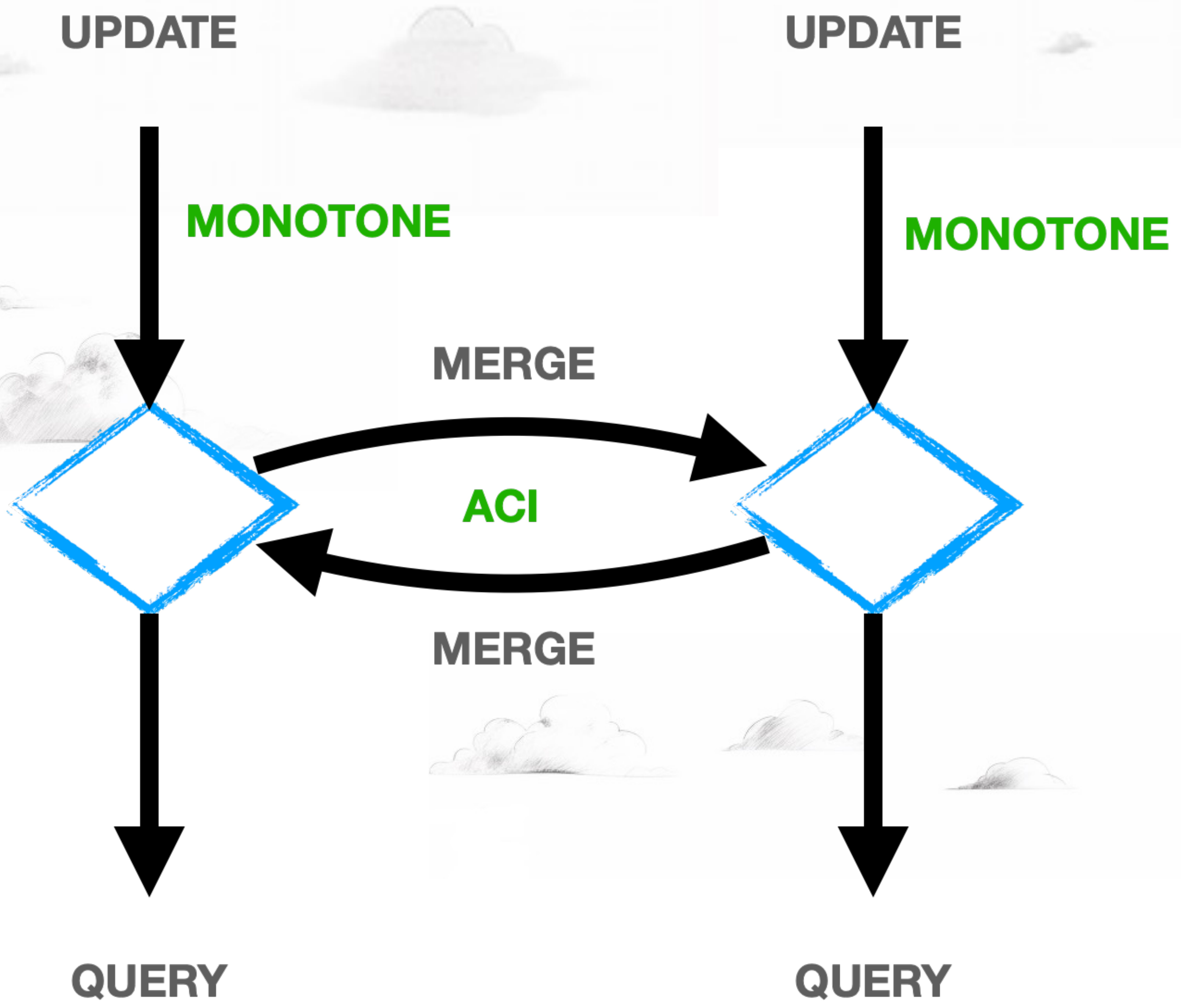
Marc Shapiro, INRIA & LIP6, Paris, France
Nuno Preguiça, CITI, Universidade Nova de Lisboa, Portugal
Carlos Baquero, Universidade do Minho, Portugal

Marek Zawirski, INRIA & UPMC, Paris, France

TR 2011

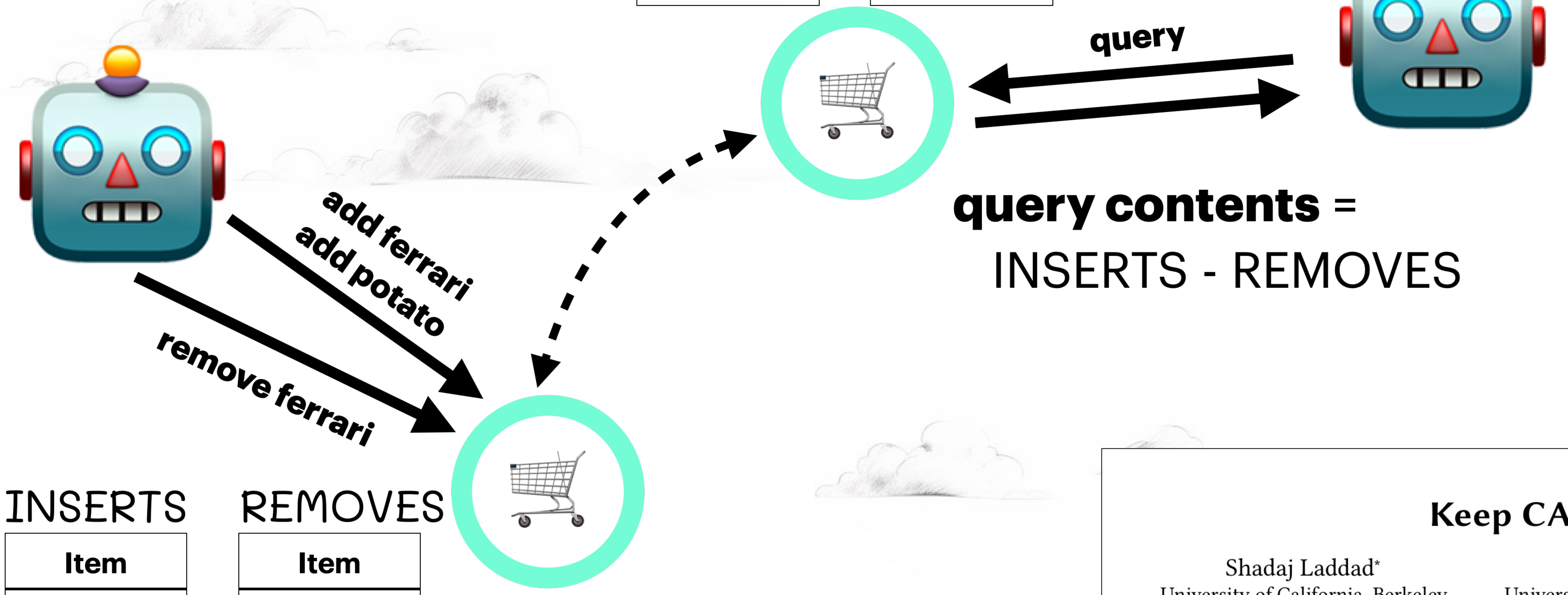


Conflict-Free Replicated Data Types (CRDTs)



CRDT Example: Shopping Cart

SemiLattice: $(\mathcal{P}(I), \cup) \times (\mathcal{P}(I), \cup)$



INSERTS

Item
Potato
Ferrari

REMOVES

Item
Ferrari

INSERTS

Item
Potato
Ferrari

REMOVES

Item
Ferrari

query contents =
INSERTS - REMOVES

VLDB 2023

Keep CALM and CRDT On

Shadaj Laddad*
University of California, Berkeley
shadaj@cs.berkeley.edu

Conor Power*
University of California, Berkeley
conorpower@cs.berkeley.edu

Mae Milano
University of California, Berkeley
mpmilano@cs.berkeley.edu

Alvin Cheung
University of California, Berkeley
akcheung@cs.berkeley.edu

Natacha Crooks
University of California, Berkeley
ncrooks@cs.berkeley.edu

Joseph M. Hellerstein
University of California, Berkeley
hellerstein@cs.berkeley.edu

Incremental View Maintenance Shopping Cart

Item	Count
Potato	1
Ferrari	1

“Remove Ferrari” →

Ferrari -= 1

SemiLattice: $(\mathcal{P}(I), \cup) \times (\mathcal{P}(I), \cup)$

vs.

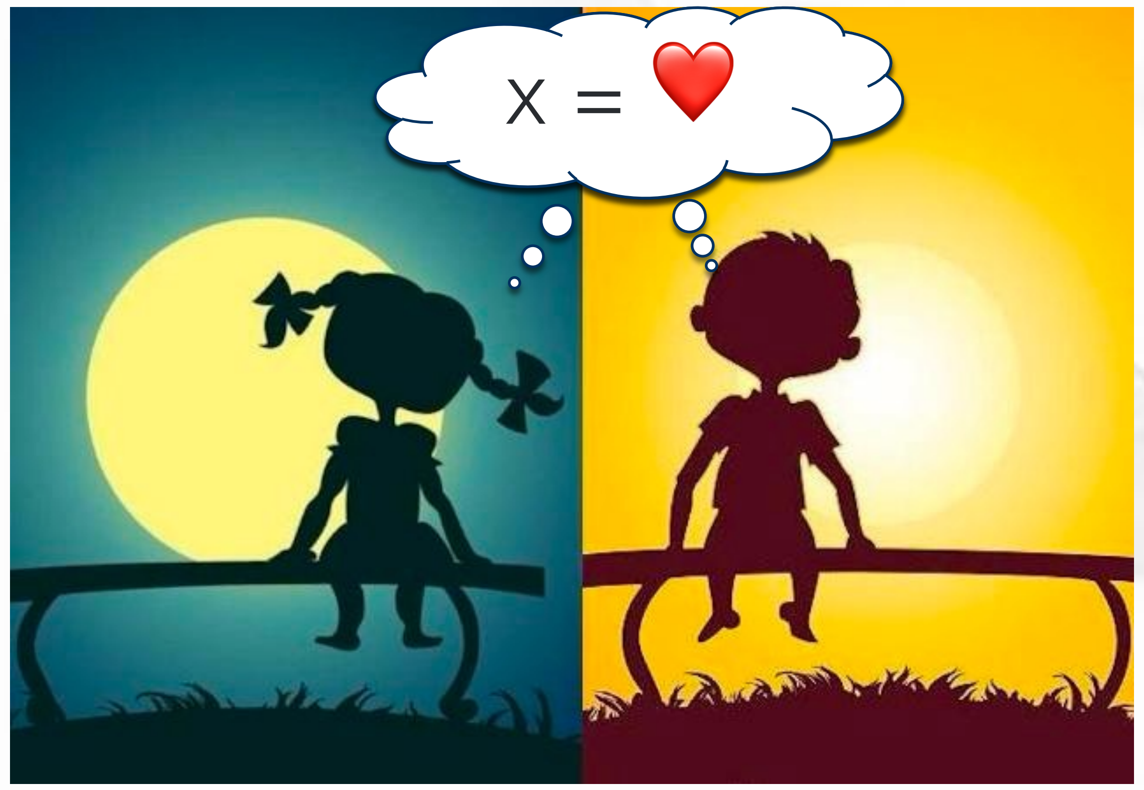
Abelian Group: $(\mathbb{Z}[I], +)$

One can put a query language “on top” of this

- Desiderata for queries over CRDT state
  - An expressive & intuitive query interface for programmers (Logic or Algebra or …)
    - Negation
    - Recursion
  - Classical query optimization e.g. operation reordering and distributivity
  - Distributed optimizations
    - Monotonicity analysis for replication
    - Functional Dependency analysis for partitioning

CALM Theorem Revisited

Challenge: Replica Consistency



- Ensure that distant agents agree (or will agree) on common knowledge.
- Classic example: data replication
  - How do we know if they agree on the value of a mutable variable x?

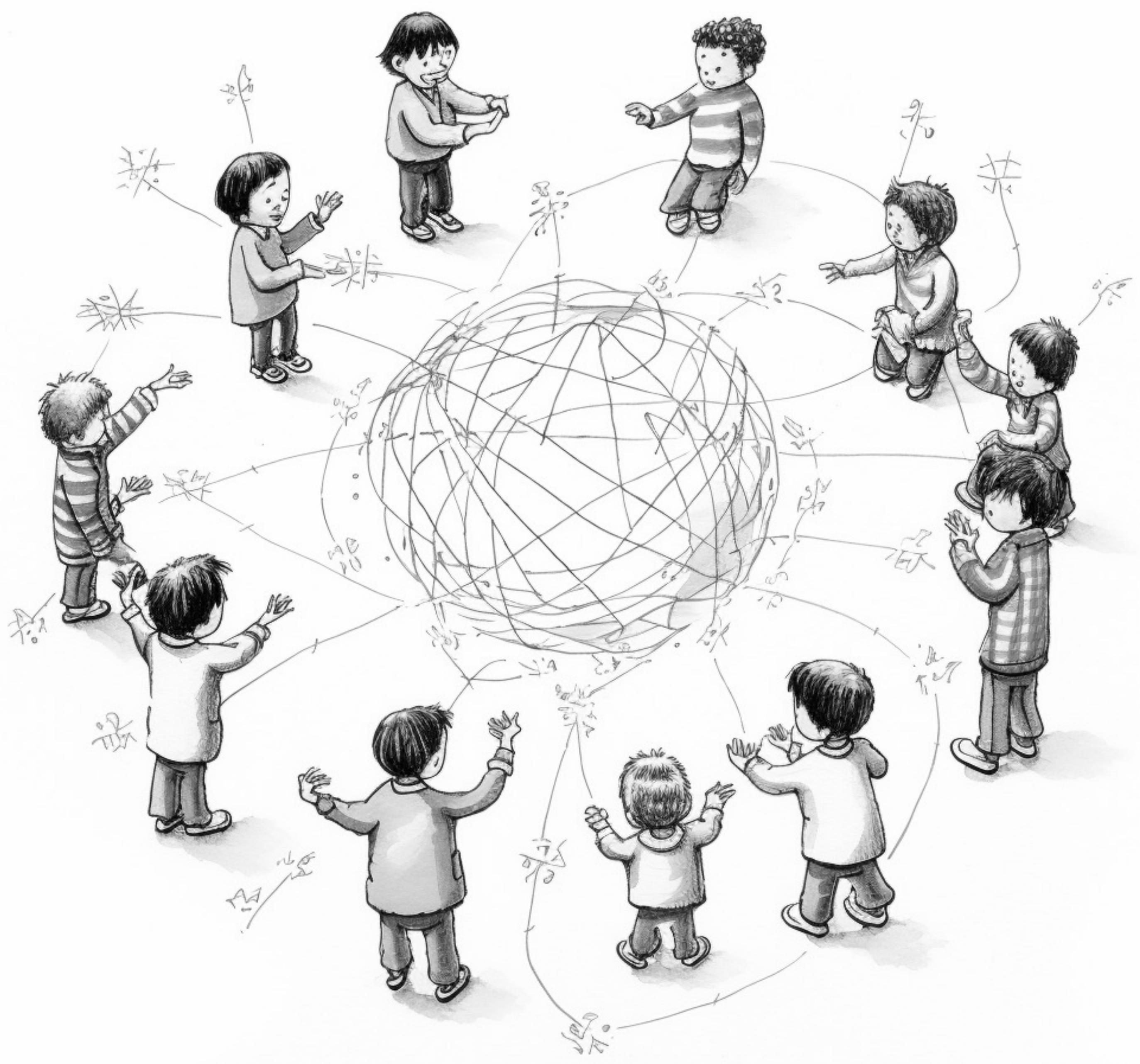
Challenge: Replica Consistency

- Ensure that distant agents agree (or will agree) on common knowledge.
- Classic example: data replication
  - How do we know if they agree on the value of a mutable variable x?
  - If they disagree now, what could happen later?
  - **Split Brain** divergence!

- We want to generalize to **program outcomes**
  - Independent of "data races" along the way

Classical Solution: Coordination

- Global total order of operations
  - via atomic instructions, locks, distributed protocols like Paxos and 2-phase commit, etc.
- Expensive at every scale
- When can we avoid?



Generational Shift to Reasoning at the App Level

20th Century	21st Century
Read/Write	Immutable State
Access/Store	Monotonicity Analysis
Linearizability	Functional Dependencies
Serializability	Data Provenance
…	…
worst-case assumptions	*app-specific assumptions*

Tired:
Reasoning about memory access

Wired:
Reasoning about App Semantics

Big Queries: When? Why?

- When do I *need* Coordination?
- Why?
- No really: Why?

When is Coordination required?

SUPPOSE YOU UNDERSTAND YOUR PROGRAM'S SEMANTICS...

- WHICH PROGRAMS HAVE A COORDINATION-FREE IMPLEMENTATION?
- WHICH PROGRAMS REQUIRE COORDINATION?

A QUESTION OF COMPUTABILITY!

Easy and Hard Questions

Is anyone over 18?	Who is the youngest?

Easy and Hard Questions

Is anyone over 18?

$$\exists x \;\; x > 18$$

Who is the youngest?

$$\exists x \forall y \, (x \leq y)$$

Easy and Hard Questions

Is anyone over 18?

$$\exists x \; x > 18$$

Monotone

Who is the person nobody is younger than?

$$\exists x \neg \exists y \, (x > y)$$

Non-Monotone

CALM: CONSISTENCY AS LOGICAL MONOTONICITY

Theorem (CALM): A distributed program has a consistent, coordination-free distributed implementation if and only if it is monotonic.

Hellerstein JM. The Declarative Imperative: Experiences and conjectures in distributed logic.
ACM PODS Keynote, June 2010
ACM SIGMOD Record, Sep 2010.

Ameloot TJ, Neven F, Van den Bussche J. Relational transducers for declarative networking.
JACM, Apr 2013.

Ameloot TJ, Ketsman B, Neven F, Zinn D. Weaker forms of monotonicity for declarative networking: a more fine-grained answer to the CALM-conjecture.
ACM TODS, Feb 2016.

Hellerstein, JM, Alvaro, P. Keeping CALM: When Distributed Consistency is Easy.
CACM, Sept, 2020.

Definitions

- Monotonic: you know

- Consistent: produces the same output regardless of data placement
  - Hence eventually consistent across replicas, runs, gossiping partitions, etc.

- Coordination:
  - “Control” messages, as opposed to “Data” messages.
  - *Coordination-free: there is* **some** *partitioning of the data s.t. the query answer is reached without communication*

More Detail

THEOREM 5.11. *Let $\mathcal{L}$ be a query language containing UCQ. For every query $\mathcal{Q}$ that is expressible in $\mathcal{L}$, the following are equivalent:*

(1) *$\mathcal{Q}$ can be distributedly computed by a coordination-free $\mathcal{L}$-transducer;*
(2) *$\mathcal{Q}$ can be distributedly computed by an oblivious $\mathcal{L}$-transducer; and,*
(3) *$\mathcal{Q}$ is monotone.*

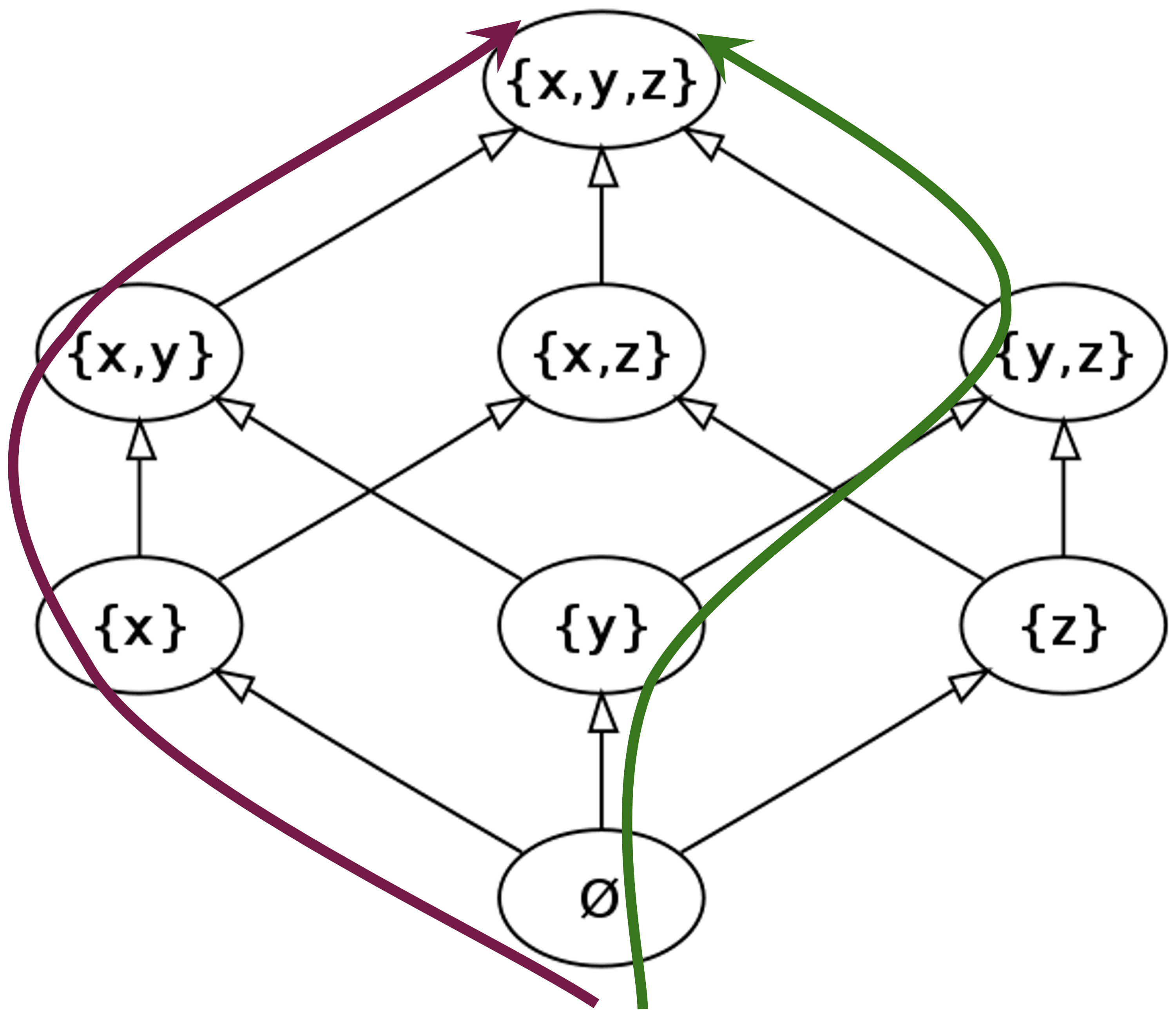
ANVdB JACM 2013

- Oblivious: does not read *Id* or *All* relations

Free Termination

Semi-Lattices: CALM Algebra

- Semi-Lattice: <S, +>
  - Associative: x + (y + z) = (x + y) + z
  - Commutative: x + y = y + x
  - Idempotent: x + x = x
- Every semi-lattice corresponds to a *partial order:*
  - x <= y ⇔ x + y = y

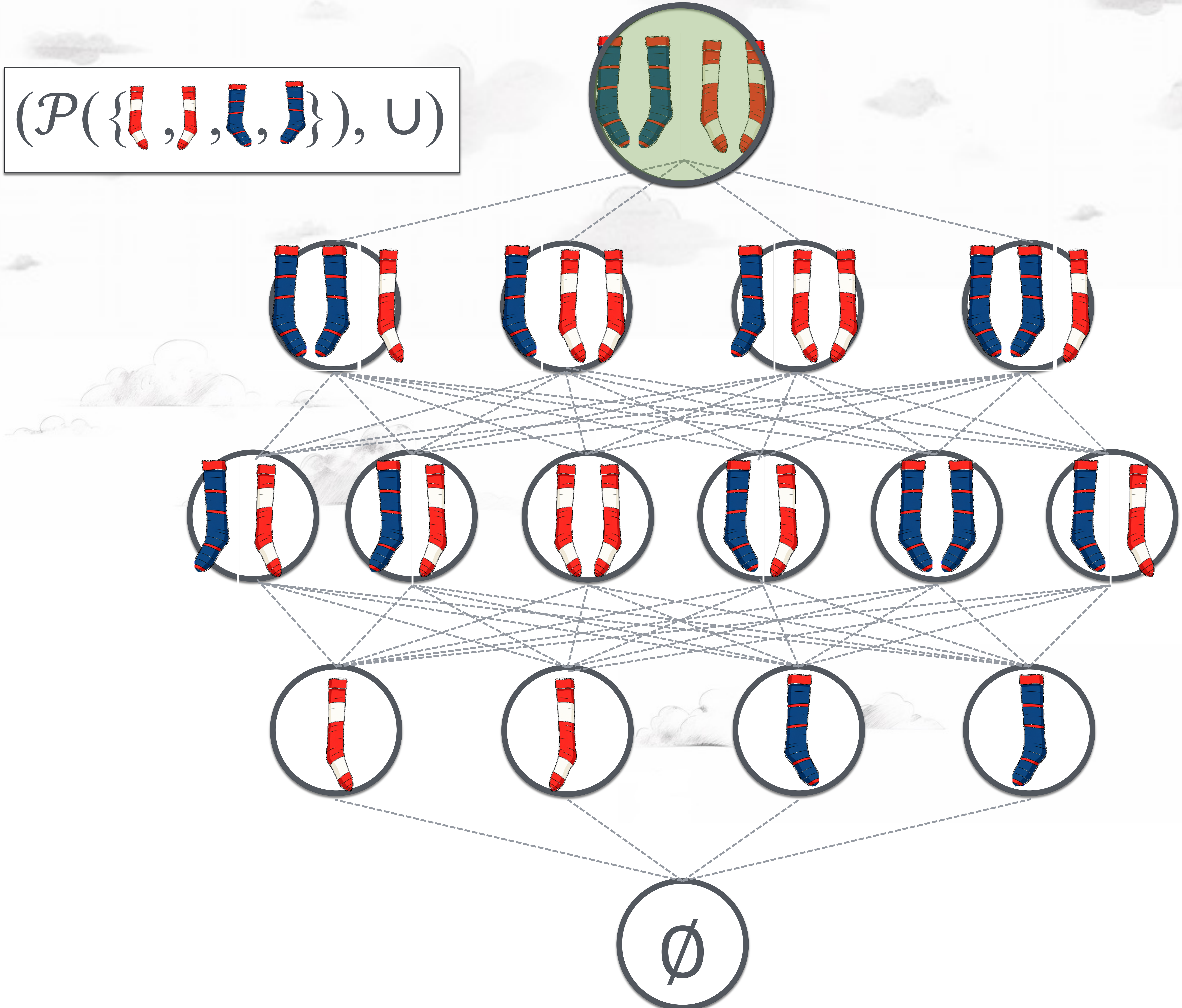


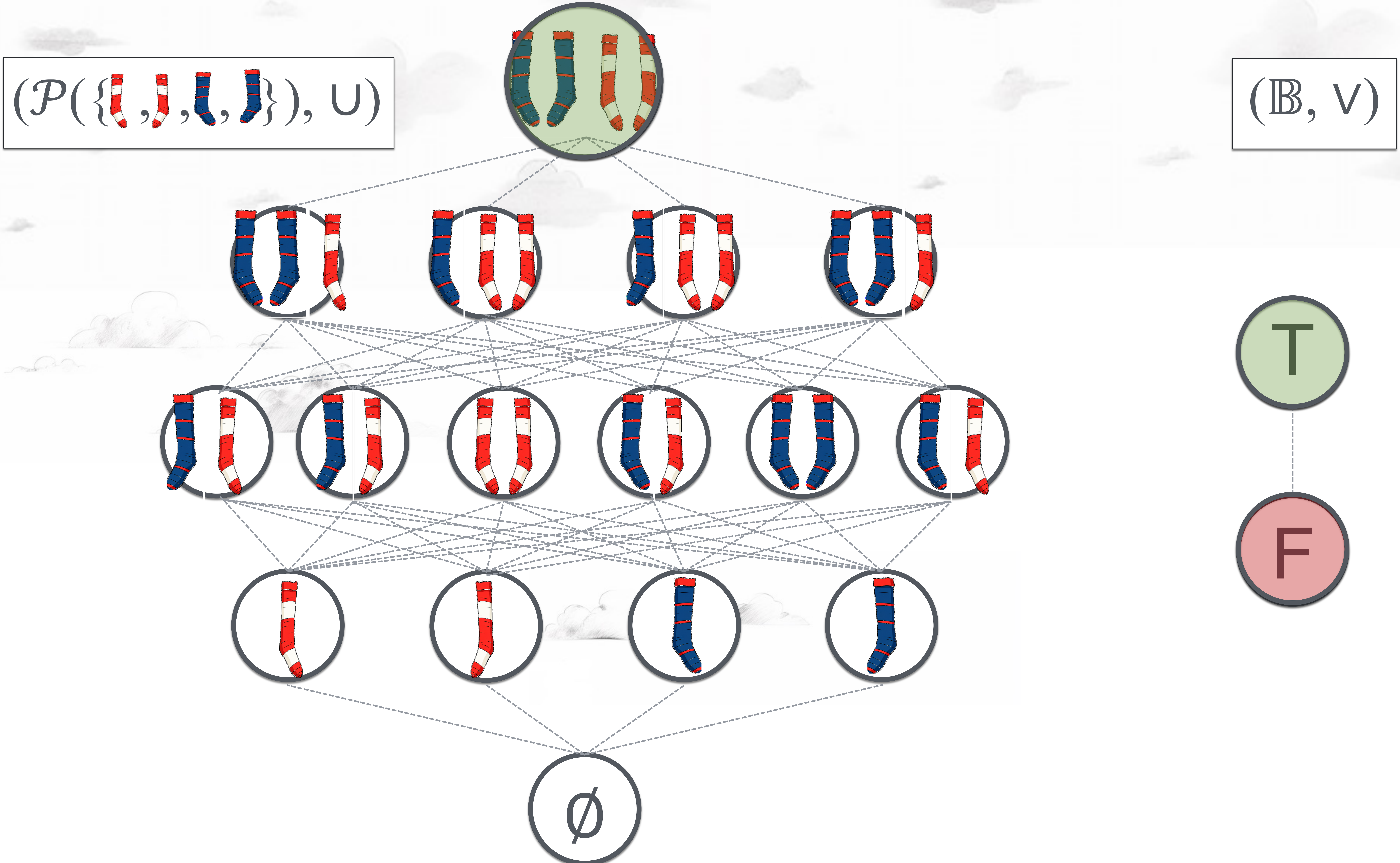
COORDINATION-FREE

CALM connection: monotonicity in the lattice's partial order

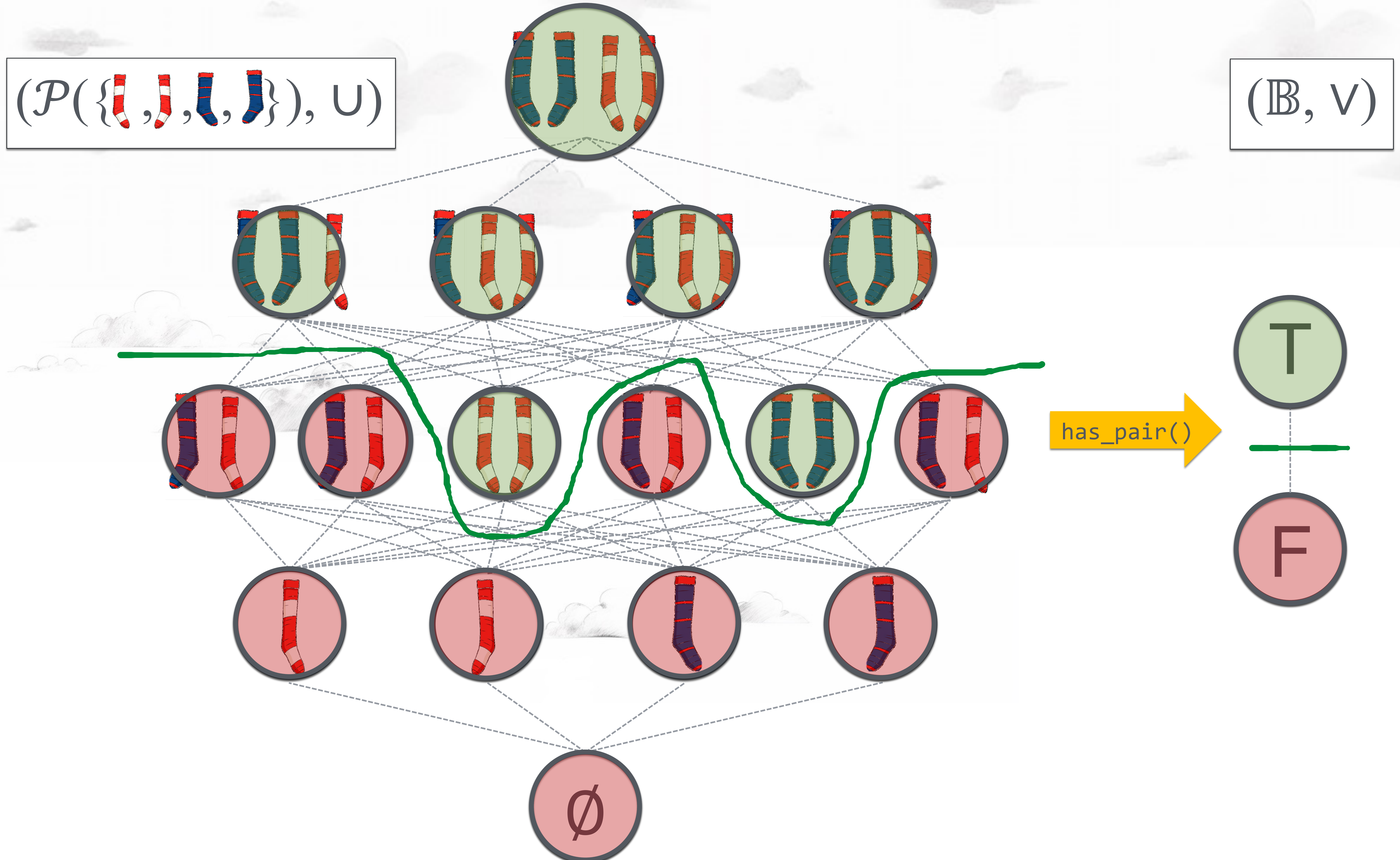
Free Termination

- Without coordination, nodes don’t know if they’ve seen the entire input
- What query results are certain regardless of future updates?
- Can we detect termination for arbitrary update and query functions?

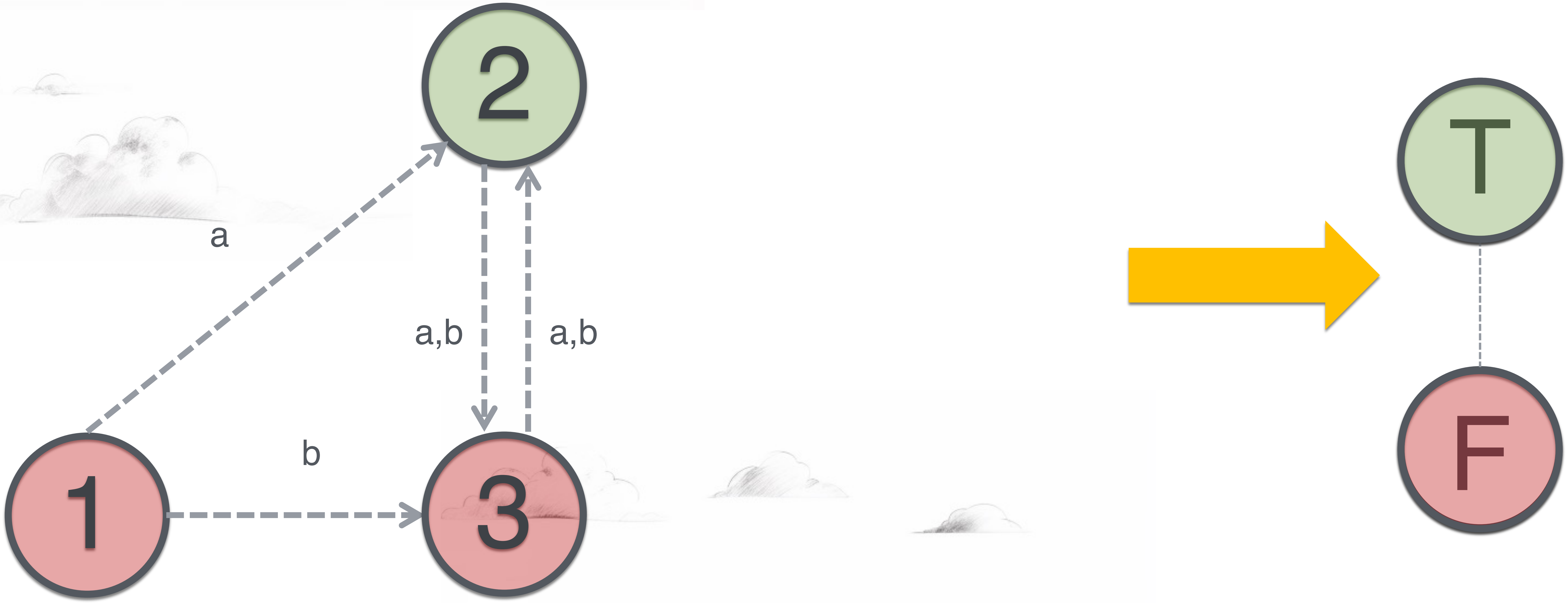

$(\mathcal{P}(\{\ ,\ ,\ ,\ \}), \cup)$
$\emptyset$



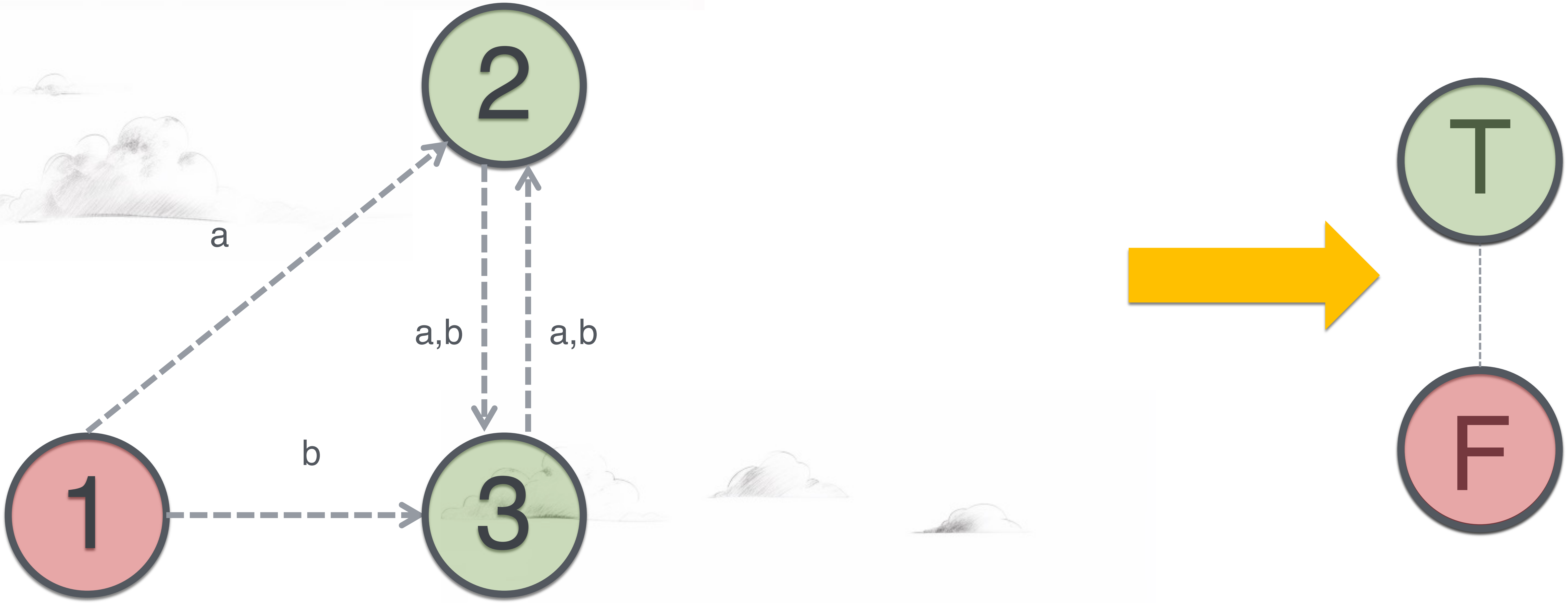
(𝒫({ , , , }), ∪)
(𝔹, ∨)
T
F
∅


(𝒫({ , , , }), ∪)
(𝔹, ∨)
has_pair()
T
F
∅

Free Termination Beyond Monotonicity

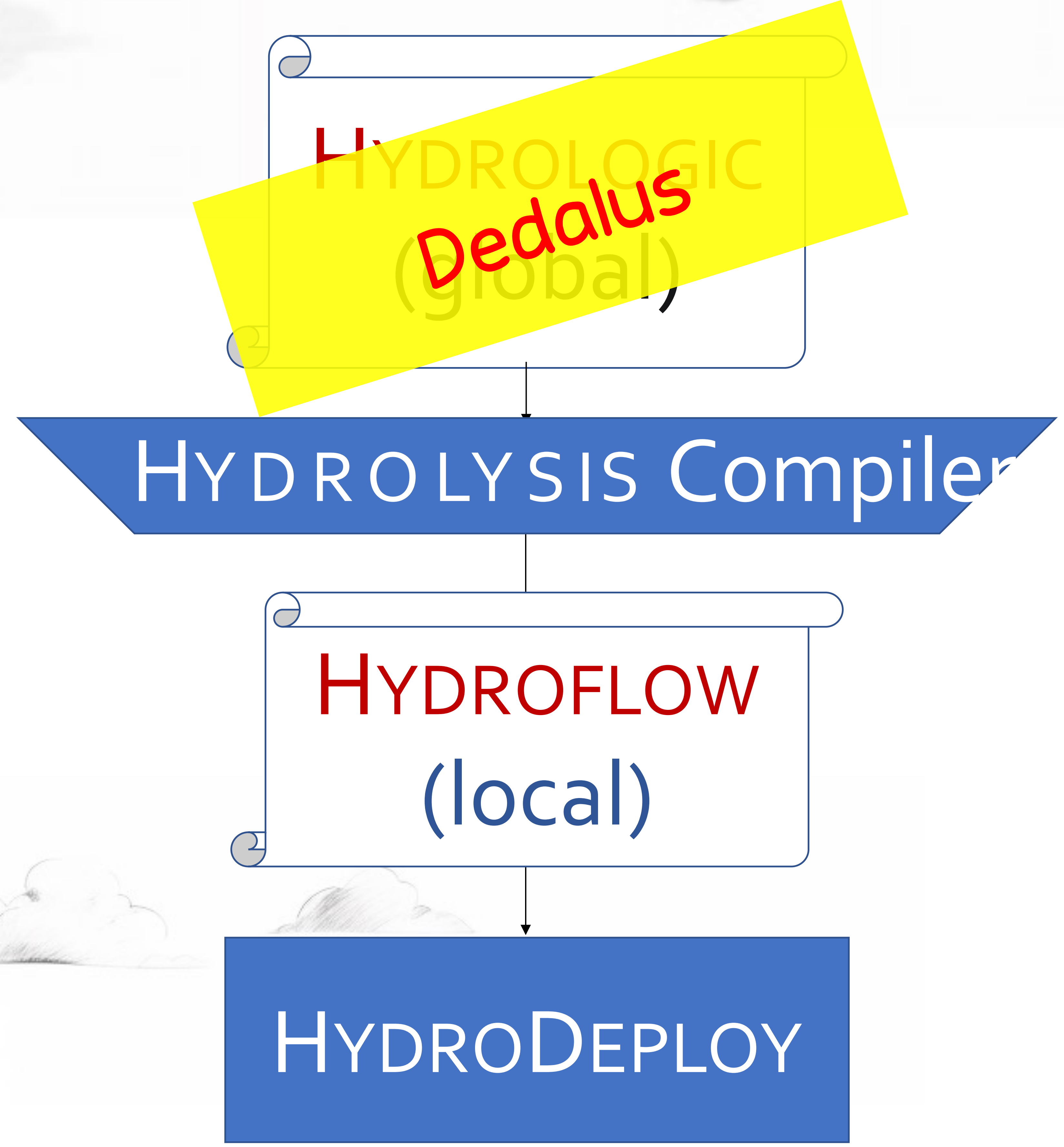


Free Termination Beyond Monotonicity



Automatic Partitioning

HYDRO Stack


HYDROLOGIC
(global)
Dedalus
HYDROLYSIS Compiler
HYDROFLOW
(local)
HYDRODEPLOY

An optimizer for protocols like Paxos? Tricky!

SOSP 2021

THE ACM SIGOPS SOSP 2021
STUDENT RESEARCH COMPETITION
GRADUATE STUDENT WINNER AWARD

is presented to

David Chu (UC Berkeley)

for their work on

Automatic Compartmentalization of Distributed Protocols

SIGOPS

Association for Computing Machinery

Optimizing Distributed Protocols with Query Rewrites

David Chu, Rithvik Panchapakesan, Shadaj Laddad, Lucky Katahanas[†], Chris Liu, Kaushik Shivakumar, Natacha Crooks, Joseph M. Hellerstein, Heidi Howard[‡]

UC Berkeley, Sutter Hill Ventures[†], Microsoft[‡]

Santiago
Chile

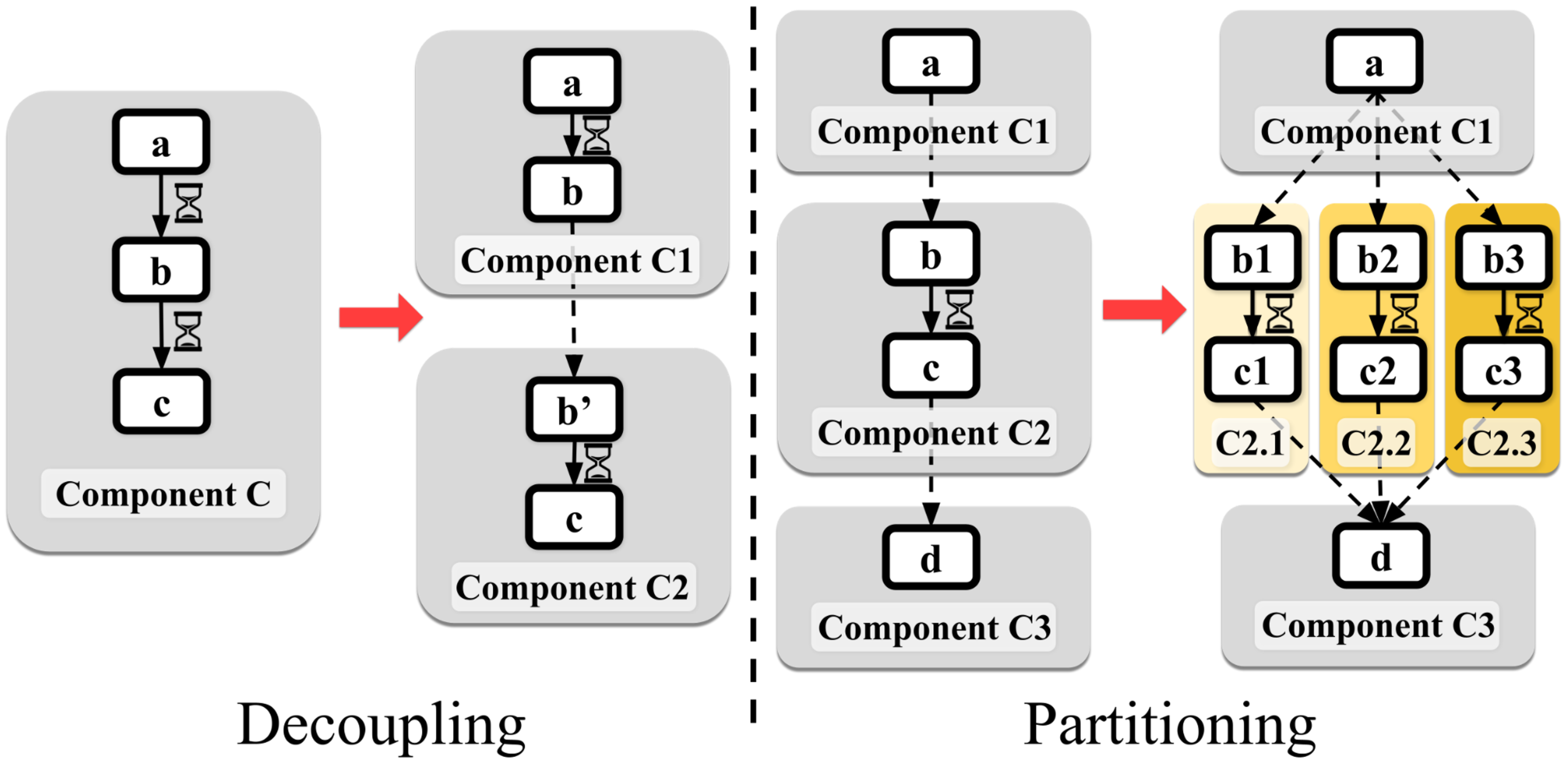
SIGMOD
PODS
2024

Challenges in Optimizing Protocols like Paxos

- Many published Paxos variants are unrecognizably equivalent
- We won't try to synthesize these human-generated variants
- Goal: using simple correct optimizations, achieve excellent performance.
  - Aim to match *performance* of the human innovations
  - In a small, provably equivalent search space of programs

Simple, Provable Equivalence

Two forms of "compartmentalization"



Scaling Replicated State Machines with Compartmentalization

Michael Whittaker
UC Berkeley
mjwhittaker@berkeley.edu

Ailidani Ailijiang
Microsoft
aiailiji@microsoft.com

Aleksey Charapko
University of New Hampshire
aleksey.charapko@unh.edu

Murat Demirbas
University at Buffalo
demirbas@buffalo.edu

Neil Giridharan
UC Berkeley
giridhn@berkeley.edu

Joseph M. Hellerstein
UC Berkeley
hellerstein@berkeley.edu

Heidi Howard
University of Cambridge
hh360@cst.cam.ac.uk

Ion Stoica
UC Berkeley
istoica@berkeley.edu

Adriana Szekeres
VMWare
aszekeres@vmware.com

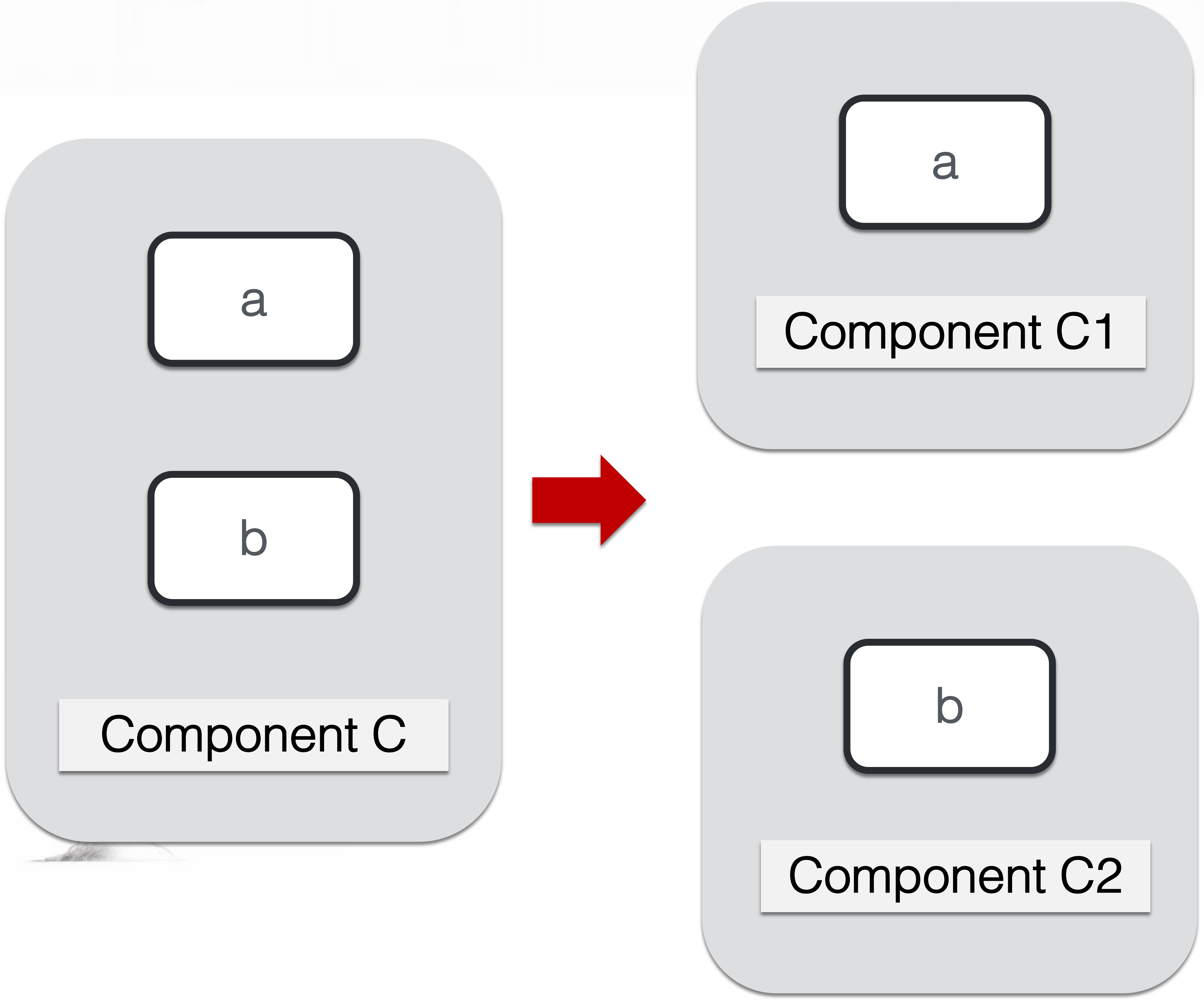
VLDB 2021

Hand-written in Scala, correct by assertion.

How much can we formalize/automate?

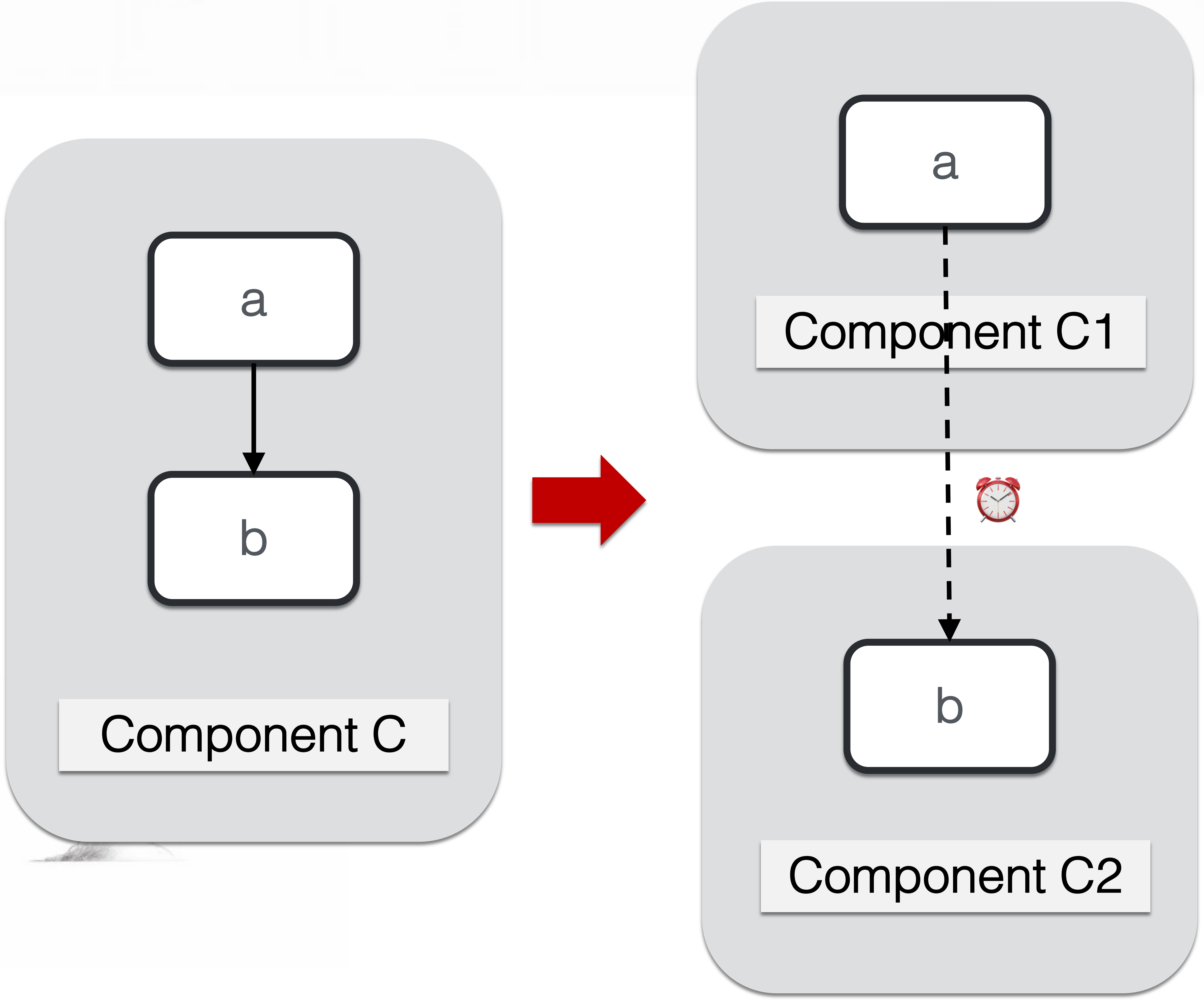
Mutually Independent Decoupling

- C1 and C2 mutually independent



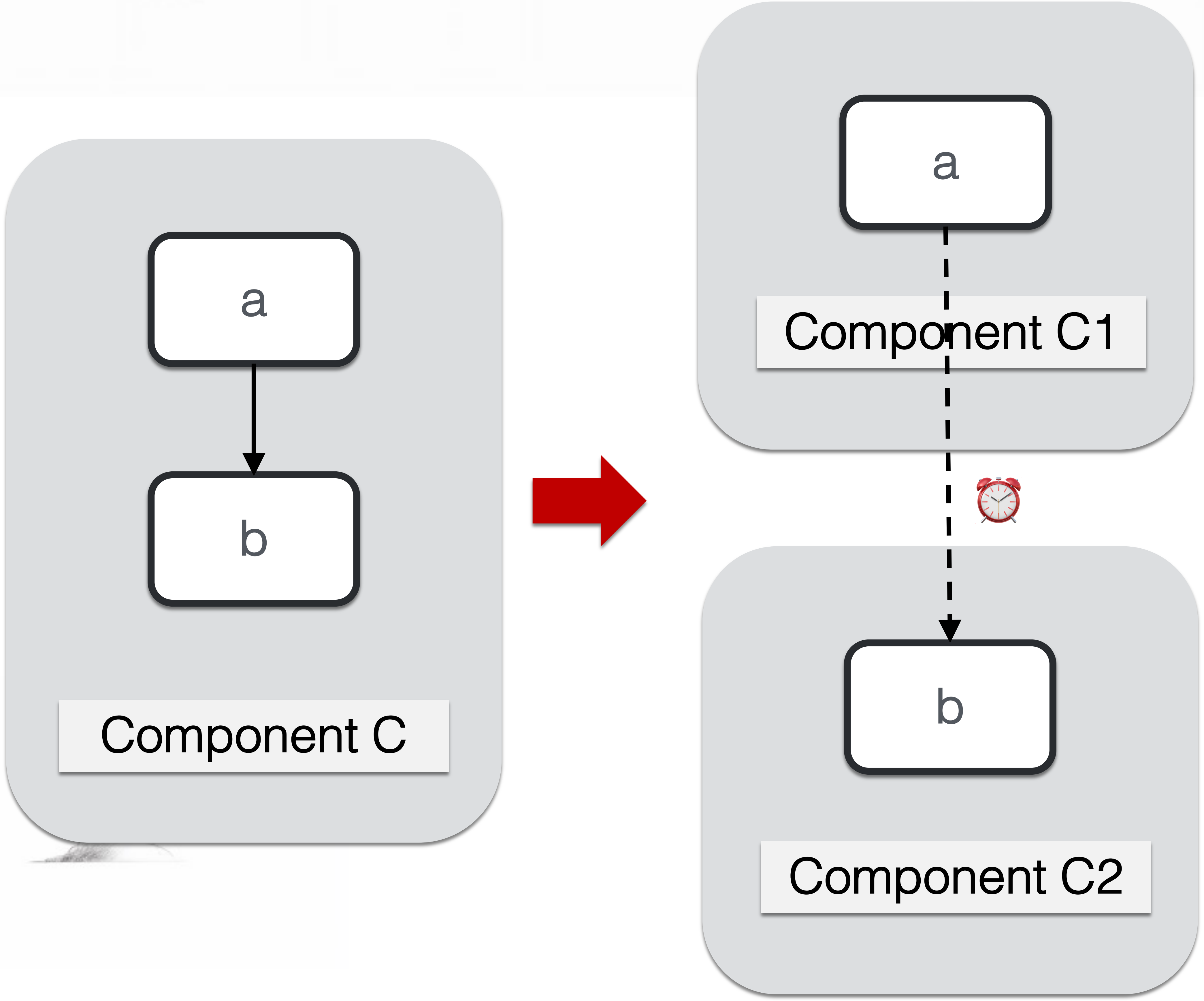
Monotonic Decoupling

- C1 and C2 mutually independent
- C2 monotonic (persistent)



Functional Decoupling

- C1 and C2 mutually independent
- C2 monotonic (persistent)
- C2 a pure function



Partitioning Discovery

- Parallel Disjoint Correctness
  [Bruhati, Koutris, Schwentick, Dagstuhl 2020]
  - Co-Hash predicates in a single rule body
    - P(A, B, C) :- R(A, B), S(B, C)

Definition 4.1. A distribution policy D over component C is *parallel disjoint correct* if for any fact f of C, for any two facts f_1, f_2 in the proof tree of f, $D(f_1) = D(f_2)$.

Partitioning Discovery

- Parallel Disjoint Correctness
  [Bruhati, Koutris, Schwentick, Dagstuhl 2020]
  - Co-Hash predicates in a single rule body
    - P(A, B, C) :- R(A, B), S(B, C)
  - Avoid re-partitioning across head-body dependencies
    - P(A, B, C) :- R(A, B), S(B, C)
    - T(A, C) :- P(A, B, C), Q(B, C)

Definition 4.1. A distribution policy D over component C is *parallel disjoint correct* if for any fact f of C, for any two facts f_1, f_2 in the proof tree of f, $D(f_1) = D(f_2)$.

Partitioning Discovery

Definition 4.1. A distribution policy D over component C is *parallel disjoint correct* if for any fact f of C, for any two facts f_1,f_2 in the proof tree of f, $D(f_1)=D(f_2)$.

- Parallel Disjoint Correctness
  [Bruhati, Koutris, Schwentick, Dagstuhl 2020]
  - Co-Hash predicates in a single rule body
    - P(A, B, C) :- R(A, B), S(B, C)
  - Avoid re-partitioning across head-body dependencies
    - P(A, B, C) :- R(A, B), S(B, C)
    - T(A, C) :- P(A, B, C), Q(B, C)
- Co-Hash by Inverse Functional Dependency
  - P(A, D) :- R(A, B), H(C, B), S(C, D)

Partitioning Discovery

- Parallel Disjoint Correctness
  [Bruhati, Koutris, Schwentick, Dagstuhl 2020]
  - Co-Hash predicates in a single rule body
    - P(A, B, C) :- R(A, B), S(B, C)
  - Avoid re-partitioning across head-body dependencies
    - P(A, B, C) :- R(A, B), S(B, C)
    - T(A, C) :- P(A, B, C), Q(B, C)

Definition 4.1. A distribution policy D over component C is *parallel disjoint correct* if for any fact f of C, for any two facts f_1, f_2 in the proof tree of f, $D(f_1) = D(f_2)$.

- Co-Hash by Inverse Functional Dependency
  - P(A, D) :- R(A, B), H(C, B), S(C, D)

$C \rightarrow B$

Given: $h(c_1) = h(c_2) \Rightarrow$ same partition

$(c_1 = c_2) \Rightarrow h(c_1) = h(c_2) \Rightarrow$ same partition

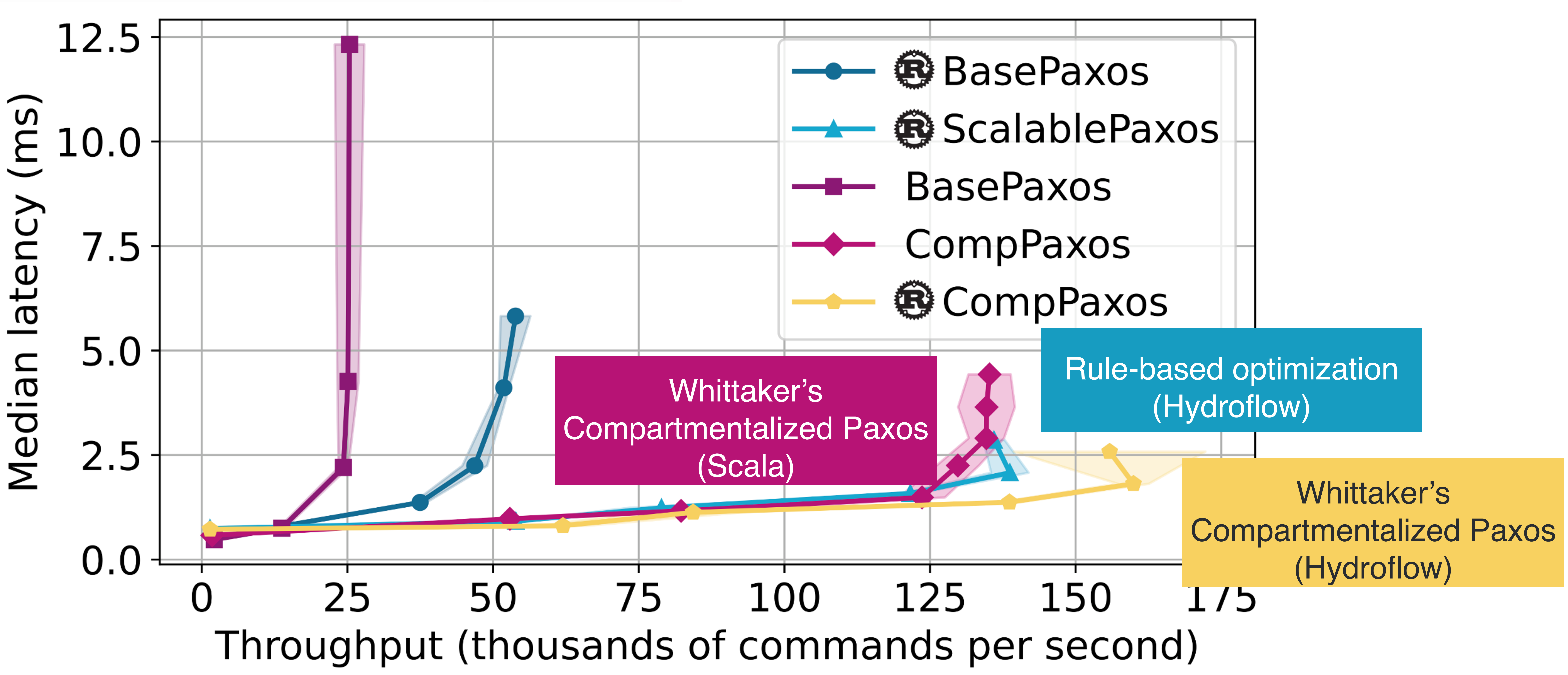
Partitioning Discovery

Definition 4.1. A distribution policy D over component C is *parallel disjoint correct* if for any fact f of C, for any two facts f_1, f_2 in the proof tree of f, $D(f_1) = D(f_2)$.

- Parallel Disjoint Correctness
  [Bruhati, Koutris, Schwentick, Dagstuhl 2020]
  - Co-Hash predicates in a single rule body
    - P(A, B, C) :- R(A, B), S(B, C)
  - Avoid re-partitioning across head-body dependencies
    - P(A, B, C) :- R(A, B), S(B, C)
    - T(A, C) :- P(A, B, C), Q(B, C)
- Co-Hash by Inverse Functional Dependency
  - P(A, D) :- R(A, B), H(C, B), S(C, D)   C → B, D → C

Fast? ✅

Beats SOTA Paxos implementations



Halfway there!

- Rules proven correct, provide desired wins
- Need:
  - Cost model for an objective function
  - Search techniques to find optimal rewritings
- E-graphs meet Query Optimizers
  - See Max's prior talk on Egg and Egglog
  - Very similar technologies!

Open Questions

Four Open Questions

1. Can we build a unified theory for all this business?
2. What's a good type system for a physical algebra (Hydroflow)?
3. What role declarative languages in the era of generative AI?
4. What is time for? When should we spend time?

1. A Unifying Theory, Please?

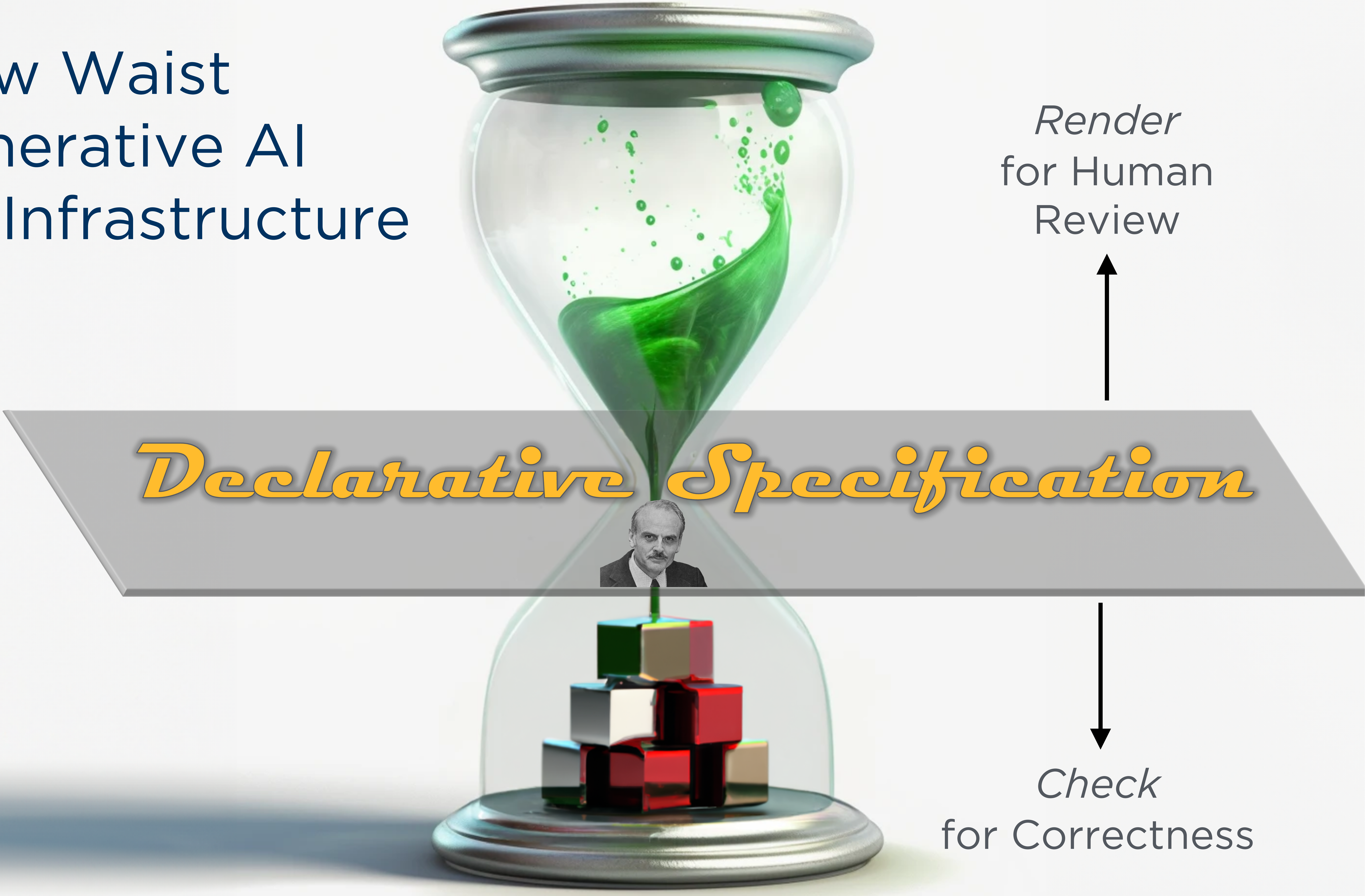
- CRDTs are semi-lattices for monotonic update across time/space
- Dedalus has a model theoretic semantics of time/space
- CALM Theorem proved using relational transducers for time/space
- Distributed system time often discussed in order theory terms
- Programmers willing to embrace functional/algebraic dataflow
- People often want to reason about transactions
- Not to mention … semi-rings!

2. Hydroflow Properties

- Stream *S* characterized by properties ($V,O,P,T,M,@,X$):
  - *V:* a multiset of *values*
  - *O:* a total *order* of arrival
  - *P:* a *parenthesization* (batching)
  - *T:* a *type* (possibly algebraic)
  - *M: monotonicity* relationship between $\leq_T$ and *O*
  - *@: atomistic* or not, *i.e.* is each item an atom of *T*
  - *X:* are all pairs *x, y* of items *exclusive,* i.e. if $z \leq_T x$, $z \leq_T y$ then $z = 0$
- Operators act on properties
  - Output **invariant** to input
  - Output **preserves** input
  - Output **enforces** property
    - **Deterministically**
    - **Non-Deterministically**

4. The Narrow Waist Between Generative AI and Reliable Infrastructure

Render
for Human
Review



Check
for Correctness

4. What is Time for?

" *Time is what keeps everything from happening at once.*"

Ray Cummings, *The Girl in the Golden Atom, 1922*

What is Time for?

" *Time is what keeps everything from happening at once.*"

Ray Cummings, *The Girl in the Golden Atom, 1922*

```
path(X,Z) :- link(X,Y), path(Y,Z)
```

Pipeline Semi-Naïve Evaluation

Loo, et al. SIGMOD 2006

What is Time for?

" *Time is what keeps everything from happening at once.*"

Ray Cummings, *The Girl in the Golden Atom, 1922*

p :- ¬p

What is Time for?

" *Time is what keeps everything from happening at once.*"

Ray Cummings, *The Girl in the Golden Atom, 1922*

p *@t+1* :- ¬p *@t*

Dedalus:

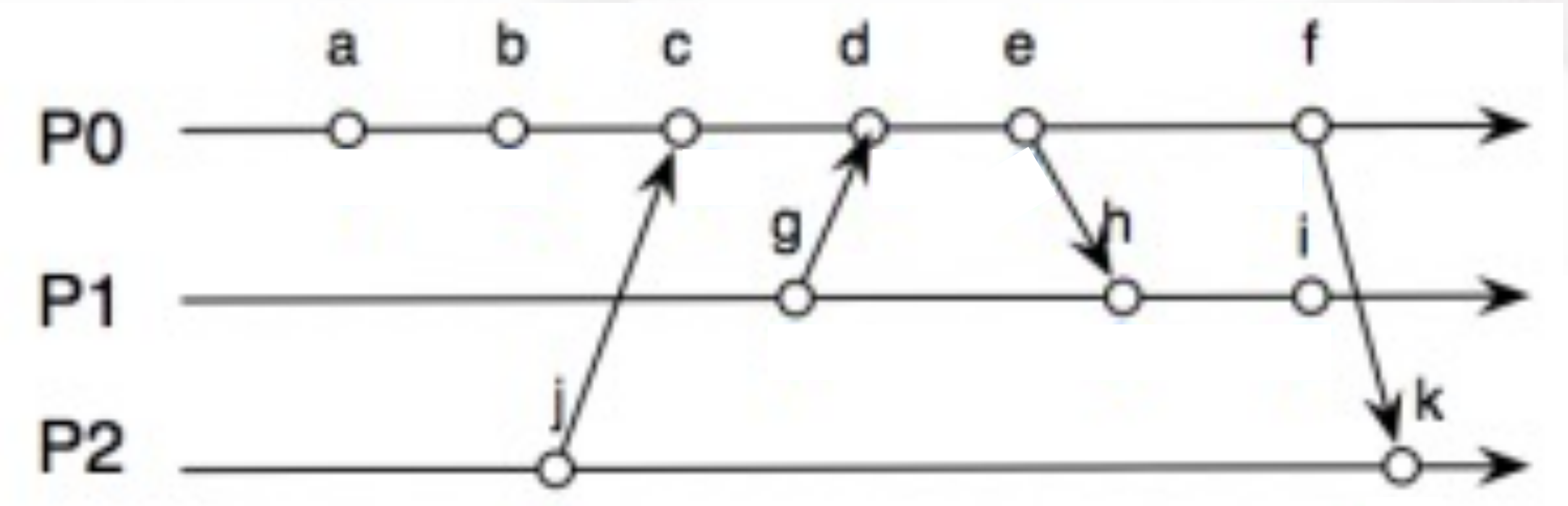
Time provides *local stratification*

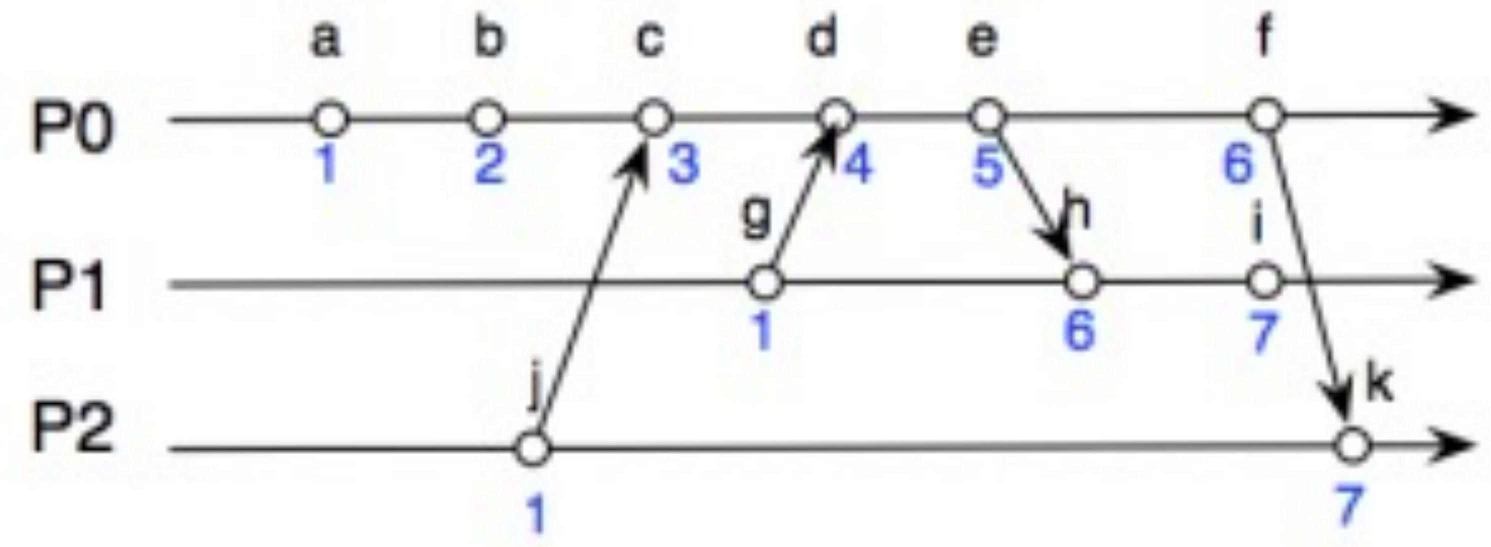
of cycles through negation

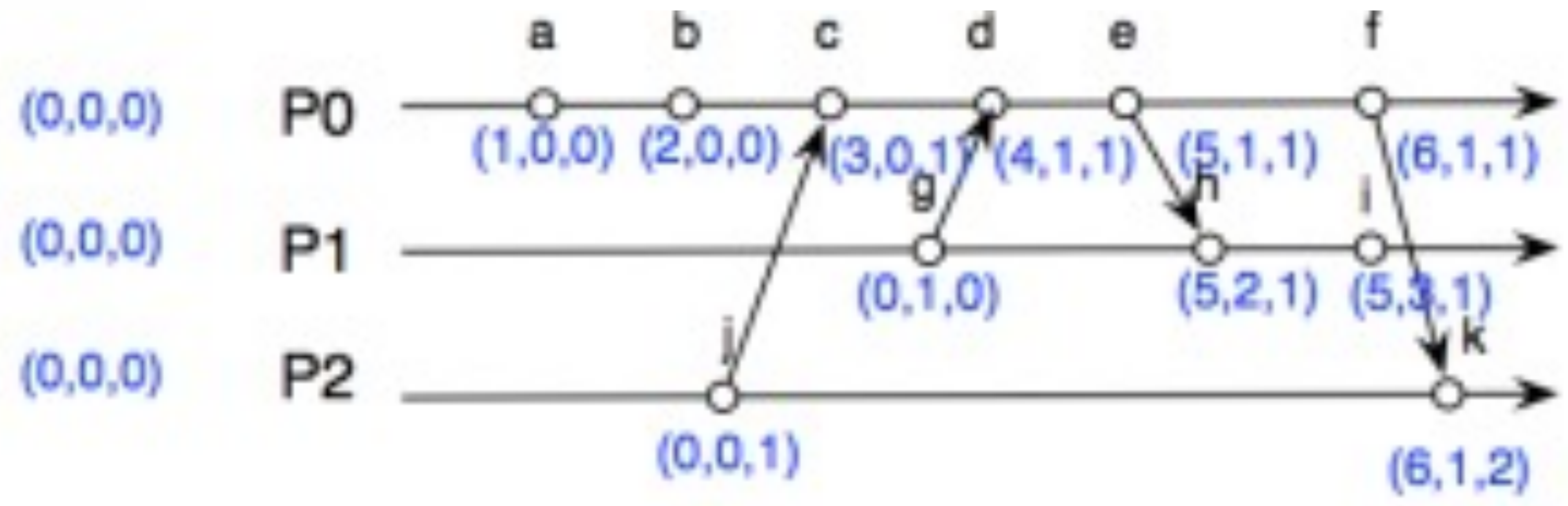
(every proof tree has finite depth)

Time In Distributed Systems is Semi-Lattice Based

- Lamport's Happens-Before: A Partial Order
  - Total-order per node (a "clock")
  - Message send precedes receive
- Lamport clock: a semi-lattice $(\mathbb{N}, \max)$
  - Provides Happens-Before relation
- Vector clock: a semi-lattice $\mathrm{MapLattice}(\mathrm{Id} \Rightarrow (\mathbb{N}, \max))$
  - Provides Happens-Before, Concurrent, Causal relations







https://newbiettn.github.io/2014/05/03/lamport-clock-vs-vector-clock/

Time can be Immaterial

Wild assertion: systems folk often “pay too much” to track time.

- Dedalus says time is irrelevant unless we cannot stratify, or we are awaiting a message
  - But details in the Dedalus paper do enforce *causality* of messages

- Causal order is *not needed* for positive Datalog.
  - Can assert facts before their antecedents are known! (E.g. during recovery).
  - A variant of the “CRON Conjecture”, which was too broad.

Datalog 2.0, 2012

On the CRON Conjecture

Tom J. Ameloot * and Jan Van den Bussche

How Many Clocks?

- Dedalus has one "clock" per node
  - Increment on "event", and to compute a cycle through negation
- Timely/Differential Dataflow: Chronomania?
  - E.g. using clocks to track async-but-monotonic iteration
- When/why do we employ a $(\mathbb{N}, \max)$ semi-lattice wrapper?
  - Can a compiler decide this?

Hydro

Thank You!

https://hydro.run

hellerstein@berkeley.edu
conorpower@berkeley.edu