Logic and Algebras for Cloud Computing

JOE HELLERSTEIN
UC BERKELEY
SUTTER HILL VENTURES

CONOR POWER
UC BERKELEY
Sea Changes in Computing

- Supercomputers:
  - Cray-1, 1976

- Minicomputers:
  - PDP-11, 1970

- Personal Computers:
  - Macintosh, 1984

- Smart Phones:
  - iPhone, 2007
New Platform + New Language = Innovation

Minicomputers
- PDP-11, 1970

Supercomputers
- Cray-1, 1976

Personal Computers
- Macintosh, 1984

Smart Phones
- iPhone, 2007
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!
- Today’s compilers don’t address distributed concerns
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 $\rightarrow$ 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.
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 95$^{th}$
    - 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 95$^{th}$
- 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 for:
  - $$\text{not latency.}$$
  - 99'th percentile, not 95'th
  - Etc.

New Directions in Cloud Programming
Alvin Cheung  Natacha Crooks  Joseph M. Hellerstein  Matthew Milano
CIDR 21
HYDRO Stack

- Sequential Code
- Futures (e.g. Ray)
- Actors (e.g. Orleans)
- Functional (e.g. Spark)
- Logic (e.g. Bloom)

HYDRAULIC Verified Lifting

HYDROLOGIC (global)

HYDROLYSIS Compiler

HYDROFLOW (local)

HYDRODEPLOY

Initial results
“Katara”: Laddad et al. OOPSLA 22

A machine-oblivious logic (or algebra)?

An e-graph based “optimizer”

Futures (e.g. Ray)

A per-node physical algebra
Implemented in Rust.

New DSLs

Actors (e.g. Orleans)

Functional (e.g. Spark)

Logic (e.g. Bloom)

…
Topics for Today (and WIP)

- Automatic Replication (of code and data)
  - Esp. “free” replication — consistency sans coordination (CALM) (\textit{algebraic CALM Theorem})

- Termination detection
  - Esp. “free” termination — detection sans coordination (\textit{threshold morphisms and equivalences})

- Automatic partitioning (of code and data)
  - Esp. “free” partitioning — parallel execution sans coordination (\textit{functional dependencies integrated into algebraic types})

And excellent performance! (vs hand-written C++)
Language/Theory Work: 2010-15

Formalism: Dedalus

Dedalus: Datalog in Time and Space
Peter Alvaro¹, William R. Marczak¹, Neil Conway¹, Joseph M. Hellerstein¹, David Maier², and Russell Sears³

CALM Theorem: coordination in its place

Consistency ⇔ Monotonicity

<- bloom: Logic + Lattices
w/stratified neg/agg, morphisms, semi-naïve eval on lattices, etc

A Declarative Semantics for Dedalus
Peter Alvaro
Tom J. Ameloot
Joseph M. Hellerstein
William Marczak
Jan Van den Bussche
TR 2011

Datalog Reloaded 2010

Keeping CALM: When Distributed Consistency Is Easy
JMH PODS Keynote, 2010
Ameloot, et al PODS 2011
Ameloot et al. TODS 2016
Alvaro/Hellerstein CACM 2020

Distributed systems are tricky. Multiple unreliable sites, incipient failures, etc.

Logic and Lattices for Distributed Programming
Neil Conway
Willam R. Marczak
Peter Alvaro
UC Berkeley
UC Berkeley
UC Berkeley
nmcl@cs.berkeley.edu
wrm@cs.berkeley.edu
palvaro@cs.berkeley.edu
Joseph M. Hellerstein
David Maier
UC Berkeley
Portland State University
hellerstein@cs.berkeley.edu
maier@cs.pdx.edu

Datalog Reloaded 2010

Datalog Reloaded 2010
Systems Work: 2015-2021

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

Cloudburst: Stateful Functions-as-a-Service

Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schiefer-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, Alexey Tumanov

U.C. Berkeley, Georgia Tech

VLDB 2020

Scaling Replicated State Machines with Compartmentalization

Michael Whittaker
UC Berkeley
mwhittaker@berkeley.edu

Murat Demirbas
University at Buffalo
demirbas@buffalo.edu

Heidi Howard
University of Cambridge
hhoward@ri.cs.cam.ac.uk

Ailadani Allijiang
Microsoft
allijiang.isa@msn.com

Neil Giridharan
UC Berkeley
giridharan@berkeley.edu

Ion Stoica
UC Berkeley
ion.stoica@berkeley.edu

Aleksey Charapko
University of New Hampshire
alexcharapko@unh.edu

Joseph M. Hellerstein
UC Berkeley
hellerstein@berkeley.edu

Adriana Szekeres
VmWare
aszekeres@vmware.com

VLDB 2021

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

SIGMOD 2015

Debugging Distributed Systems with Why-Across-Time Provenance

Michael Whittaker
UC Berkeley
mwhittaker@berkeley.edu

Peter Alvaro
UC Santa Cruz
palvaro@sccs.ucsc.edu

Cristina T dreaded
UC Berkeley
tc@berkeley.edu

Joseph M. Hellerstein
UC Berkeley
hellerstein@berkeley.edu

SOCC 2018
Systems Highlight: Anna Key-Value Store

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

Anna: A KVS For Any Scale
Chenggang Wu *, Jose M. Faleiro *, Yihan Lin **, Joseph M. Hellerstein ***

Autoscaling Tiered Cloud Storage in Anna
Chenggang Wu, Vikram Sreekanti, Joseph M. Hellerstein

ICDE 18
VLDB 19
Systems Highlight: Anna Key-Value Store

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

Anna: A KVS For Any Scale

Chenggang Wu *, Jose M. Faleiro **, Yihan Lin ***, Joseph M. Hellerstein ***

2022 SIGMOD Jim Gray Doctoral Dissertation Award

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 2018 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 (local disaggregation with physical colocated); these ideas

Autoscaling Tiered Cloud Storage in Anna

Chenggang Wu, Vikram Sreekanti, Joseph M. Hellerstein

ICDE 18
VLDB 19
Examples of lattice composition

- Metadata “wrappers” for various replica consistency mechanisms

### Last Writer Wins

- **MapLattice**
  - **Key**: (\(\mathbb{N}, \text{Max}\))
  - **LexicalPair**: Data: \(T\)

### Causal Consistency

- **MapLattice**
  - **Key**
  - **LexicalPair**: Data: \(L\)
  - **MapLattice**
    - **Key**
    - **(\(\mathbb{N}, \text{Max}\))**
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

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: \((\mathbb{N}, \text{max})\)
  - 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.
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))
  -> destsink_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;
peers_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;

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)) -> 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);

// Peers: when a new peer joins, send them all data.
peer_join = cross_join()
  -> peer_store;
peer_store -> [0]peer_store;
peer_store = union() -> persist();
outbound_gossip = cross_join()
  -> gossip back to same sender.
outbound_gossip
  -> map(|((key, _value, writer_addr), peer_addr)|
      (KvsMessage::PeerGossip { key, value }, peer_addr))
  -> filter(!_key != peer_addr)
  -> map(|((key, value, _writer_addr), peer_addr)|
      (KvsResponse { key, value }, client_addr))
  -> dest_sink_serde(outbound);

// Outbound gossip. Send updates to peers.
peers = [1]peer_join;
peer_join = cross_join()
  -> persist();
outbound_gossip = cross_join()
  -> gossip back to same sender.
outbound_gossip
  -> map(|((key, _value, writer_addr), peer_addr)|
      (KvsMessage::PeerGossip { key, value }, peer_addr))
  -> filter(!_key != peer_addr)
  -> map(|((key, value, _writer_addr), peer_addr)|
      (KvsResponse { key, value }, client_addr))
  -> dest_sink_serde(outbound);

See The Hydro Book: https://hydro.run/docs/hydroflow/
Original Anna KVS. C++
2018 Amazon m4.16xlarge instances
(64 vCPU, 256GB RAM,)

Fast?
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 $R^2 = 0.999$

Throughput (ops/s) vs 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

- **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, CTTI, Universidade Nova de Lisboa, Portugal  
Carlos Baquero, Universidade do Minho, Portugal  
Marek Zawirski, INRIA & UPMC, Paris, France

TR 2011

(S, ⊔)
Conflict-Free Replicated Data Types (CRDTs)
CRDT Example: Shopping Cart

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

INSERTS

<table>
<thead>
<tr>
<th>Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>Potato</td>
</tr>
<tr>
<td>Ferrari</td>
</tr>
</tbody>
</table>

REMOVES

<table>
<thead>
<tr>
<th>Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ferrari</td>
</tr>
</tbody>
</table>

query contents = INSERTS - REMOVES

VLDB 2023

Keep CALM and CRDT On

<table>
<thead>
<tr>
<th>Name</th>
<th>University</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shadaj Laddad</td>
<td>University of California, Berkeley</td>
<td><a href="mailto:shadaj@cs.berkeley.edu">shadaj@cs.berkeley.edu</a></td>
</tr>
<tr>
<td>Conor Power</td>
<td>University of California, Berkeley</td>
<td><a href="mailto:conorpower@cs.berkeley.edu">conorpower@cs.berkeley.edu</a></td>
</tr>
<tr>
<td>Alvin Cheung</td>
<td>University of California, Berkeley</td>
<td><a href="mailto:akcheung@cs.berkeley.edu">akcheung@cs.berkeley.edu</a></td>
</tr>
<tr>
<td>Natacha Crooks</td>
<td>University of California, Berkeley</td>
<td><a href="mailto:ncrooks@cs.berkeley.edu">ncrooks@cs.berkeley.edu</a></td>
</tr>
<tr>
<td>Mae Milano</td>
<td>University of California, Berkeley</td>
<td><a href="mailto:mamilano@cs.berkeley.edu">mamilano@cs.berkeley.edu</a></td>
</tr>
<tr>
<td>Joseph M. Hellerstein</td>
<td>University of California, Berkeley</td>
<td><a href="mailto:hellerstein@cs.berkeley.edu">hellerstein@cs.berkeley.edu</a></td>
</tr>
</tbody>
</table>
### Incremental View Maintenance Shopping Cart

<table>
<thead>
<tr>
<th>Item</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Potato</td>
<td>1</td>
</tr>
<tr>
<td>Ferrari</td>
<td>1</td>
</tr>
</tbody>
</table>

“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
- Read/Write
- Access/Store
- Linearizability
- Serializability
- ... worst-case assumptions

21st Century
- Immutable State
- Monotonicity Analysis
- Functional Dependencies
- Data Provenance
- ... 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?

$$\exists x \ x > 18$$

Who is the person nobody is younger than?

$$\exists x \ \neg \exists y \ (x > y)$$
Theorem (CALM): A distributed program has a consistent, coordination-free distributed implementation if and only if it is monotonic.


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

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

**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:
  - \(x \leq y \iff x + y = y\)

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?
$(\mathcal{P}(\{1, 2, 3\}), \cup)$
$(\mathcal{P}\{\emptyset, \{1\}, \{2\}, \{1, 2\}\}, \cup)$

$(\mathbb{B}, \vee)$
\((\mathcal{P}(\{\text{red}, \text{blue}, \text{green}\}), \cup), (\mathbb{B}, \lor)\)
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!
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”

Hand-written in Scala, correct by assertion.

How much can we formalize/automate?
Mutually Independent Decoupling

C1 and C2 mutually independent

Component C

Component C1

Component C2
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
  - \( P(A, B, C) : \neg R(A, B), S(B, C) \)

**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
  - \( P(A, B, C) : \neg R(A, B), S(B, C) \)

- Avoid re-partitioning across head-body dependencies
  - \( P(A, B, C) : \neg R(A, B), S(B, C) \)
  - \( T(A, C) : P(A, B, C), Q(B, C) \)

**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**

---

### Parallel Disjoint Correctness

[Bruhati, Koutris, Schwentick, Dagstuhl 2020]

- **Co-Hash predicates in a single rule body**
  - \( P(A, B, C) : \neg R(A, B), S(B, C) \)

- **Avoid re-partitioning across head-body dependencies**
  - \( P(A, B, C) : \neg R(A, B), S(B, C) \)
  - \( T(A, C) : P(A, B, C), Q(B, C) \)

---

### Co-Hash by Inverse Functional Dependency

- \( P(A, D) : R(A, B), H(C, B), S(C, D) \)

---

**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
  - \( P(A, B, C) \,:=\, R(A, B), S(B, C) \)
  - \( T(A, C) \,:=\, P(A, B, C), Q(B, C) \)

Co-Hash by Inverse Functional Dependency

- \( P(A, D) \,:=\, R(A, B), H(C, B), S(C, D) \)  \( C \rightarrow B \)

**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 \( f \), \( D(f_1) = D(f_2) \).

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

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

$P(A, D) :- R(A, B), H(C, B), S(C, D)$

$C \rightarrow B, D \rightarrow C$

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 $f$, $D(f_1) = D(f_2)$.
Fast? ☑️

Beats SOTA Paxos implementations

![Graph showing performance comparison between different Paxos implementations. The graph plots median latency against throughput (thousands of commands per second). The x-axis is labeled 'Throughput (thousands of commands per second)'. The y-axis is labeled 'Median latency (ms)'. Legend includes: BasePaxos, ScalablePaxos, CompPaxos, and CompPaxos. There are annotations indicating 'Rule-based optimization (Hydroflow)' and 'Whittaker’s Compartmentalized Paxos (Scala)'.]
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
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.”

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.”

\[ p : \neg p \]
What is Time for?

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

\[ p_{t+1} : \neg p_t \]

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 $(\mathbb{N}, \text{max})$
  - Provides Happens-Before relation

- Vector clock: a semi-lattice $\text{MapLattice} (\text{Id} \Rightarrow (\mathbb{N}, \text{max}))$
  - Provides Happens-Before, Concurrent, Causal relations
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.
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 $(\mathbb{N}, \text{max})$ semi-lattice wrapper?
  - Can a compiler decide this?