Bring Your Own Data Structures to Datalog

OOPSLA '23

Arash Sahebolamri (Syracuse), Langston Barrett (Galois), Scott Moore (Galois), Kristopher Micinski (Syracuse)

Datalog

Datalog is positive Horn clauses over atomic literals

$$\text{Path}(x, y) \leftarrow \text{Edge}(x, y)\,.$$

"When there's an edge between x and y, there's a path between x and y."

$$\text{Path}(x, z) \leftarrow \text{Path}(x, y), \text{Edge}(y, z)\,.$$

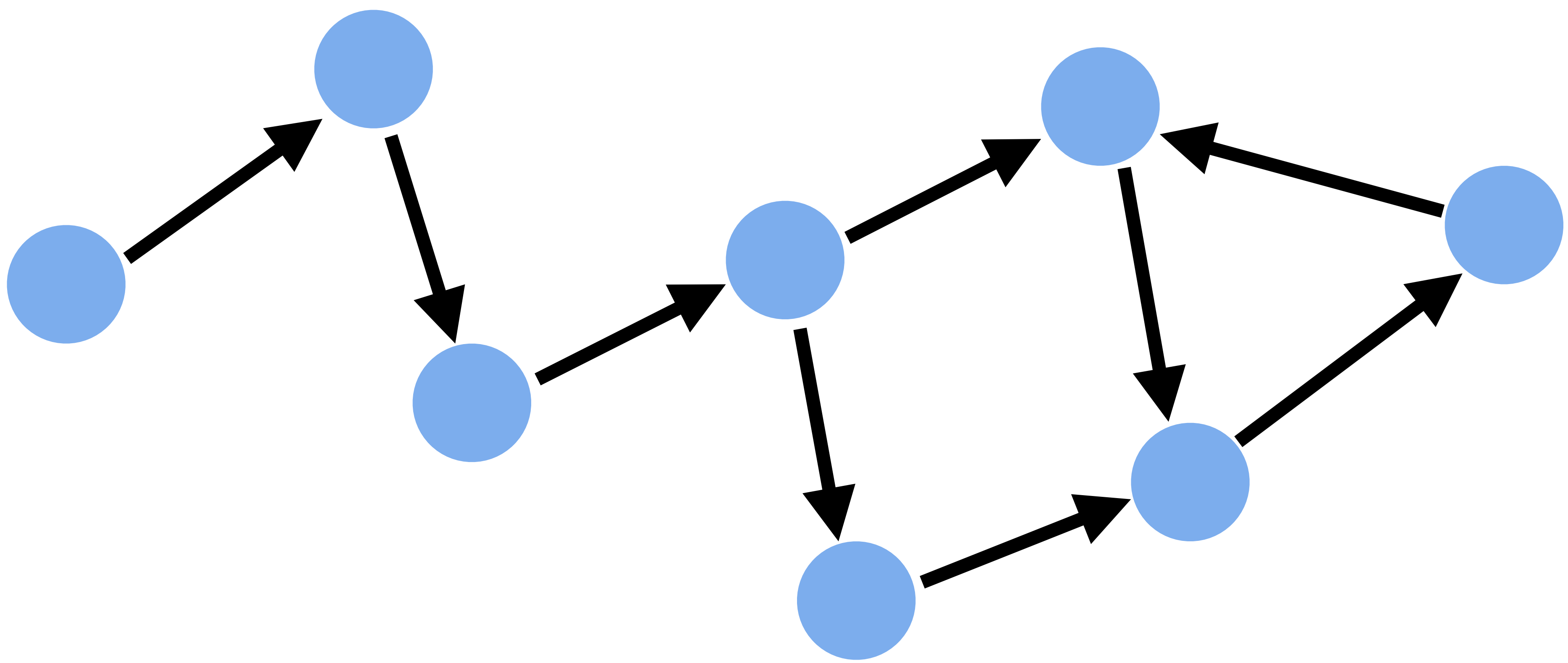
"When there's a path between x and y, and an edge from y to z, there's a path between x and z."

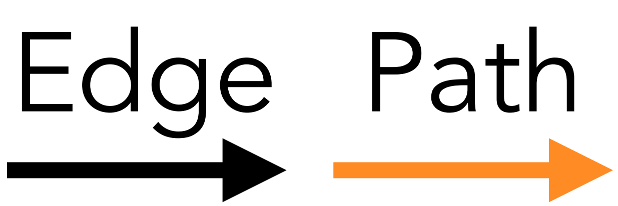
$$\mathrm{Path}(x, y) \leftarrow \mathrm{Edge}(x, y)\,.$$

Datalog is a generalization of SQL to recursive rules

This first rule is plain old SQL:

```
Path ← SELECT x, y FROM Edge
```



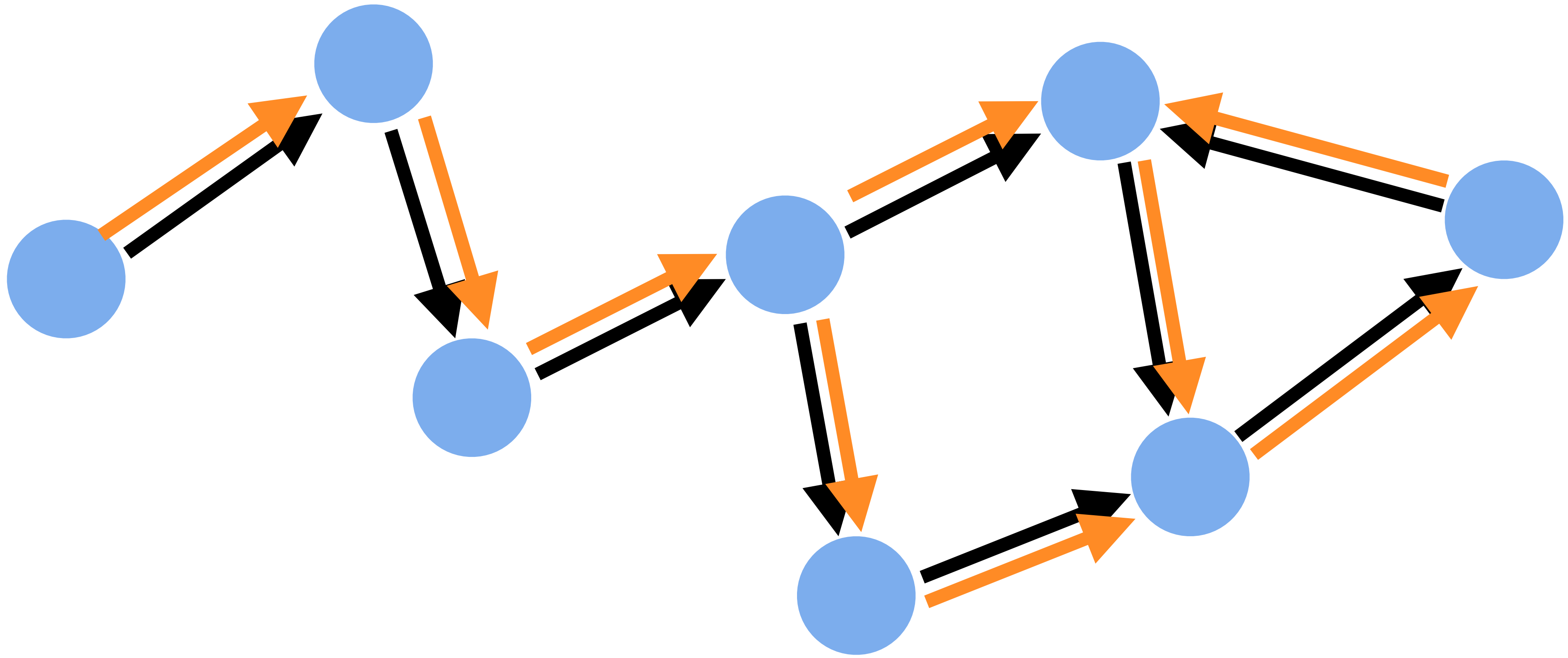


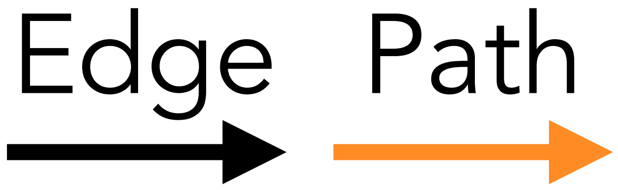
$$\text{Path}(x, y) \leftarrow \text{Edge}(x, y)\,.$$

Datalog is a generalization of SQL to recursive rules

This first rule is plain old SQL:

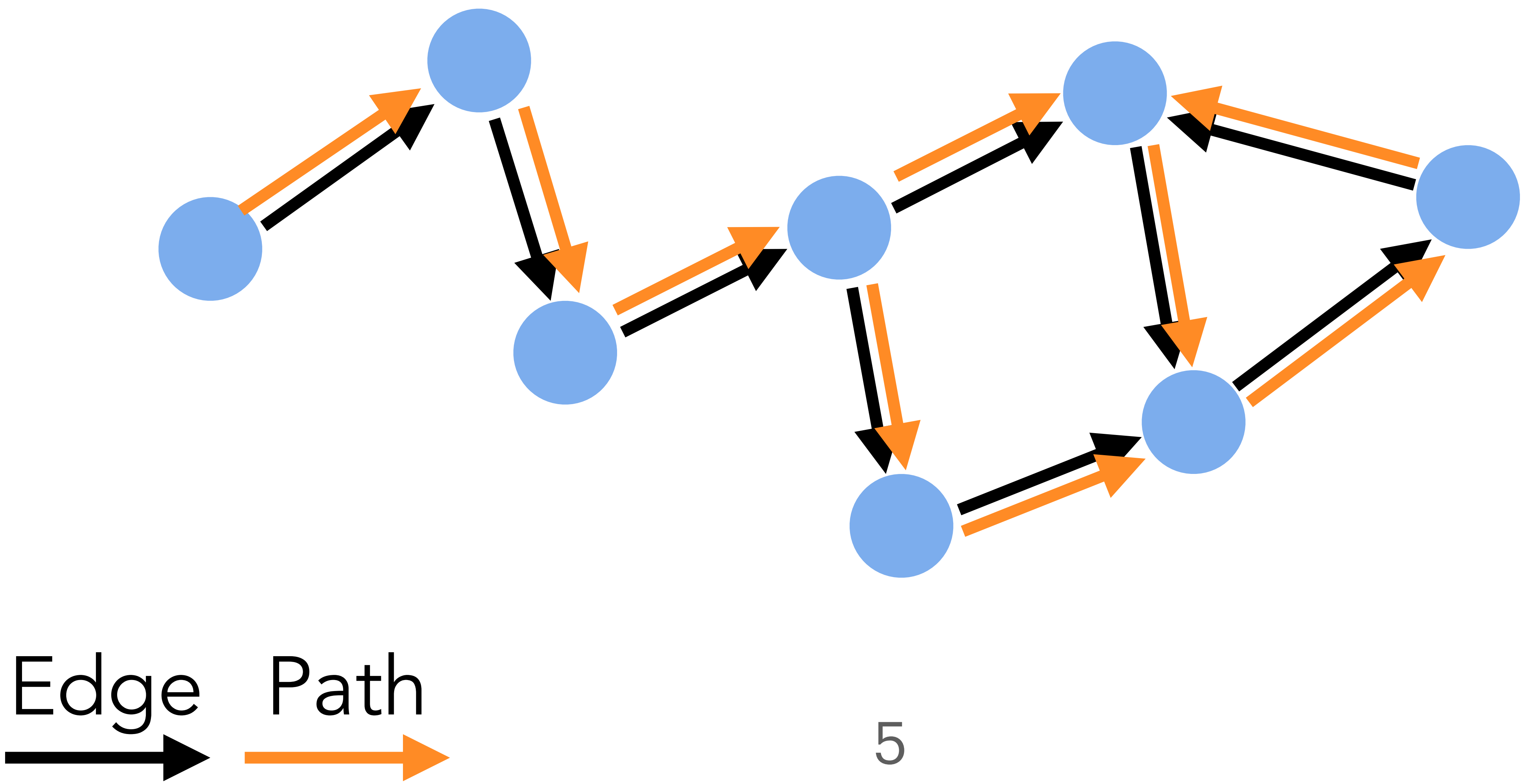
```
Path ← SELECT x, y FROM Edge
```





$$\mathrm{Path}(x, y) \leftarrow \mathrm{Edge}(x, y)\,.$$
$$\mathrm{Path}(x, z) \leftarrow \mathrm{Path}(x, y), \mathrm{Edge}(y, z)\,.$$

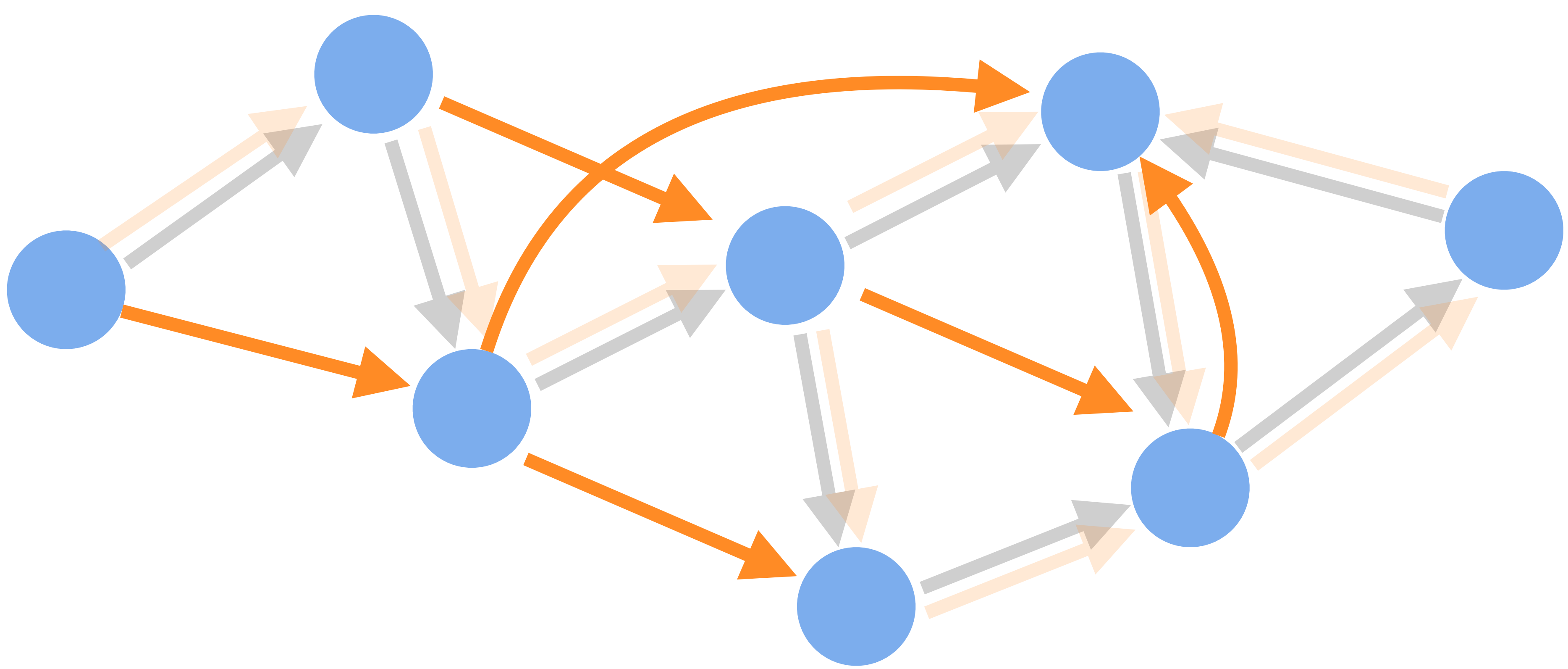
The second rule is inductive, and can't be written in SQL
(without CTEs, which are too slow for us)

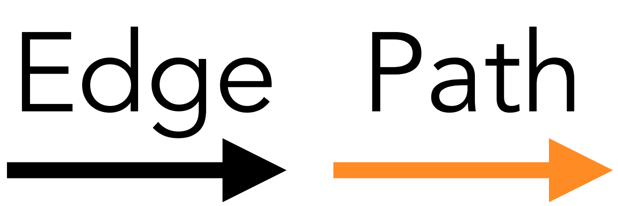


$$\text{Path}(x, y) \leftarrow \text{Edge}(x, y)\,.$$
$$\text{Path}(x, z) \leftarrow \text{Path}(x, y), \text{Edge}(y, z)\,.$$

The rule is evaluated in a fixed-point loop.

Each iteration we calculate Path ⋈ Edge and add it to Path





The first iteration gives us the "two hop reachable" edges

We need to be careful: if we "rediscover" previously-discovered edges, we are doing asymptotically-more work

This is the "naïve" evaluation strategy

We need to be careful: if we “rediscover” previously-discovered edges, we are doing asymptotically-more work

This is the “naïve” evaluation strategy

Don’t be naïve

We need to be careful: if we “rediscover” previously-discovered edges, we are doing asymptotically-more work

This is the “naïve” evaluation strategy

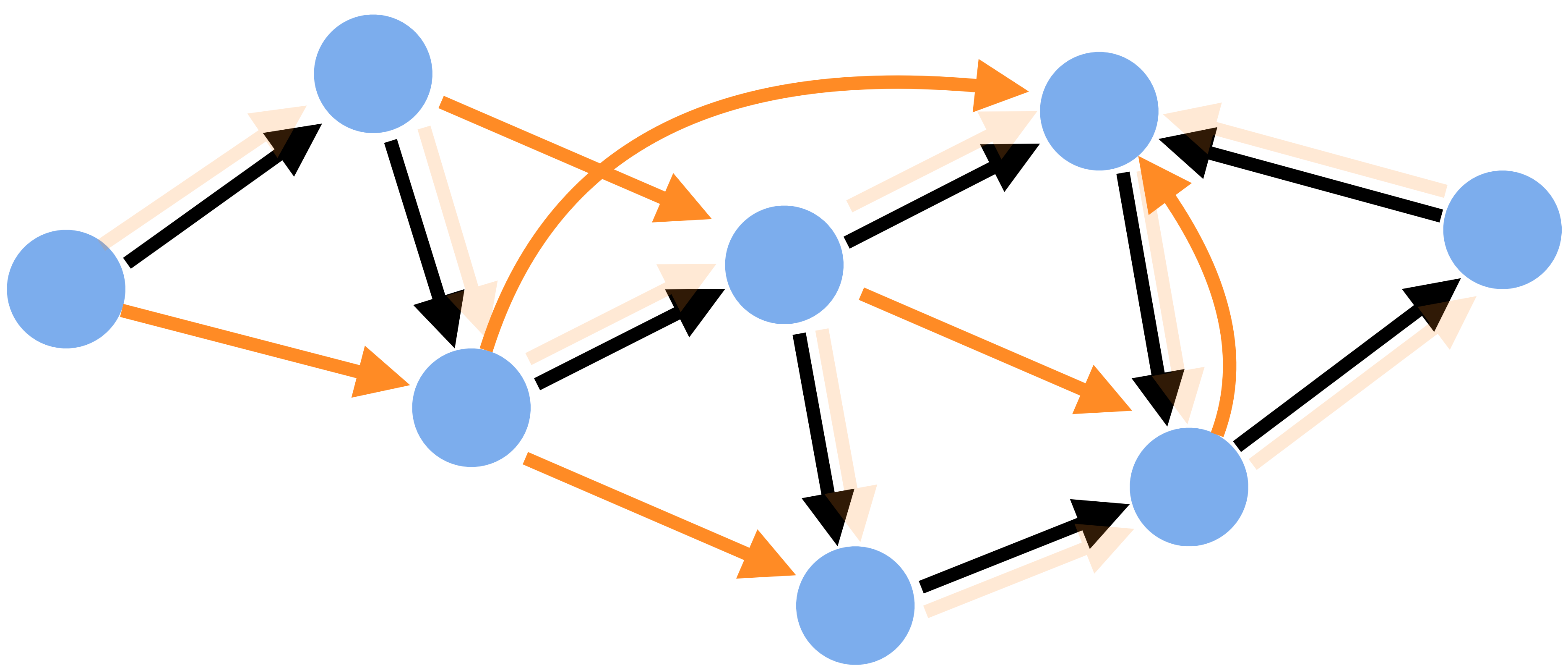
Don’t be naïve

Be *semi*-naïve

With respect to the cartoon here:

Instead of Path ⋈ Edge, compute Path_Δ ⋈ Edge

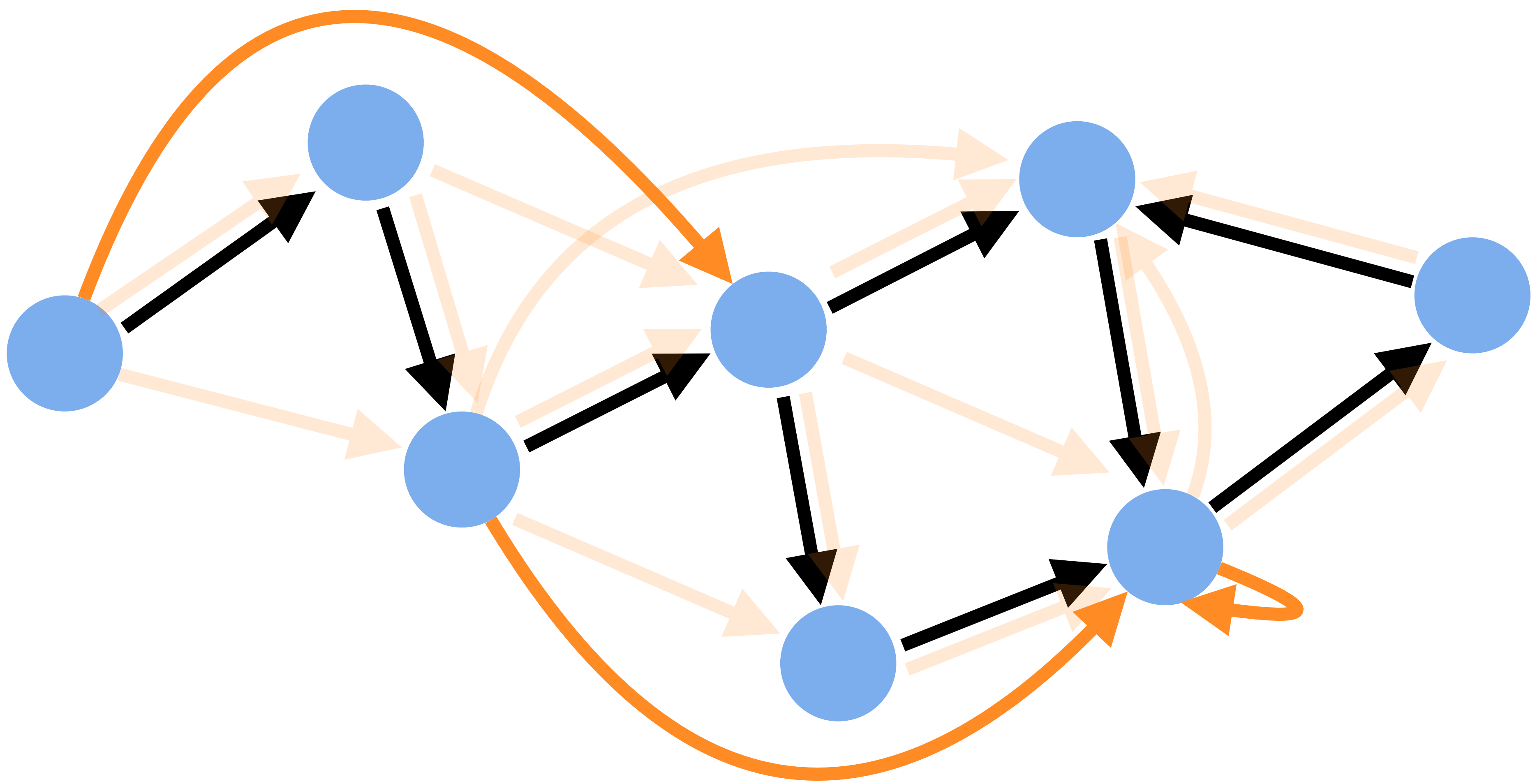
Path_Δ is all of edges discovered in the *most recent iteration*

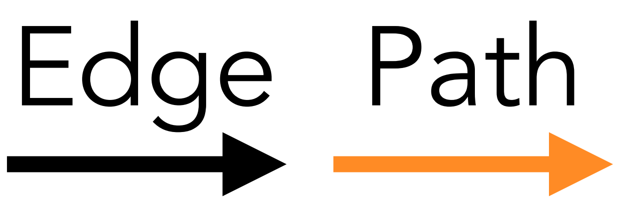


Edge Path

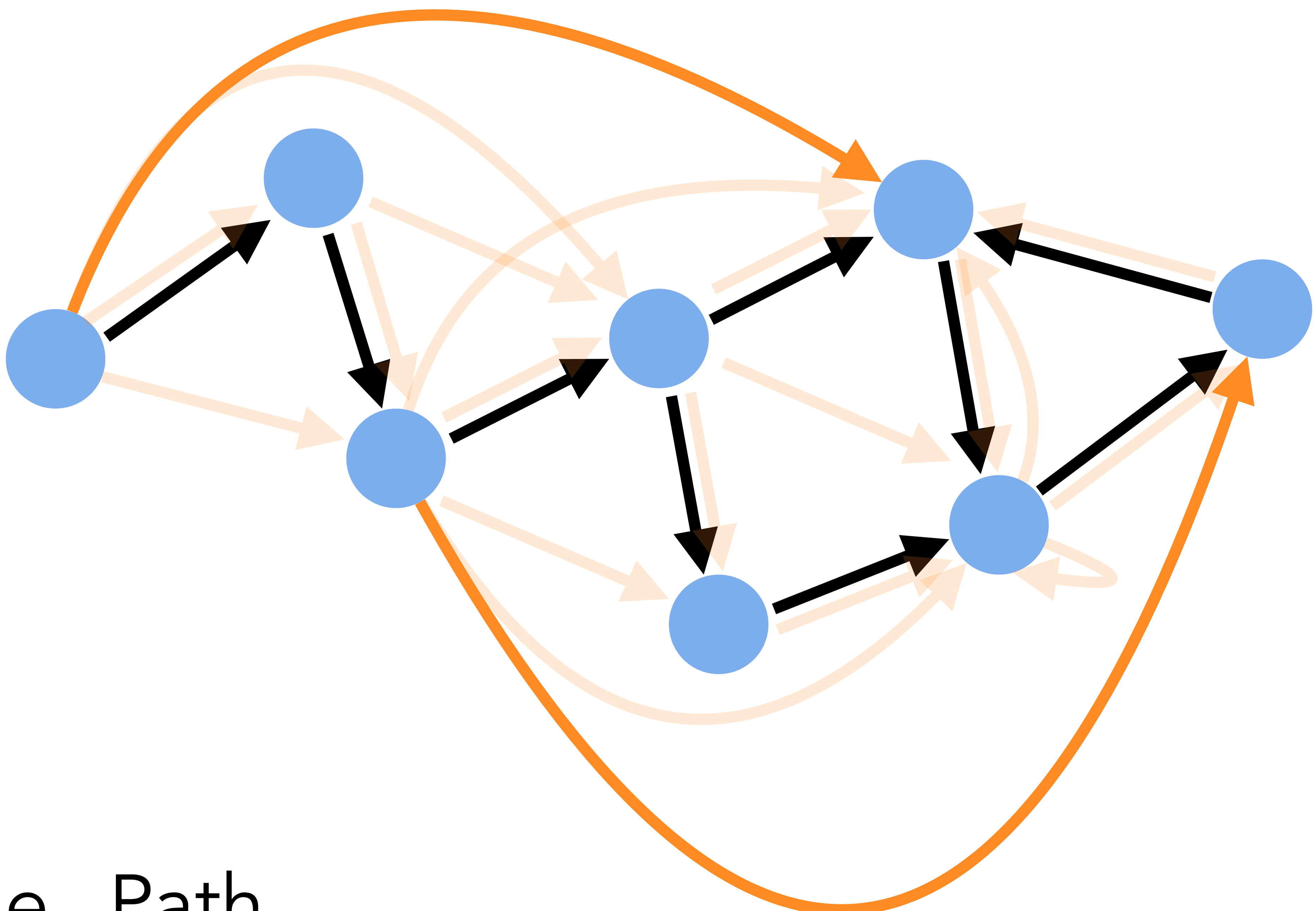
I.e., turn back up the opacity of Edge and do *this* join

Now we get some more transitive edges…



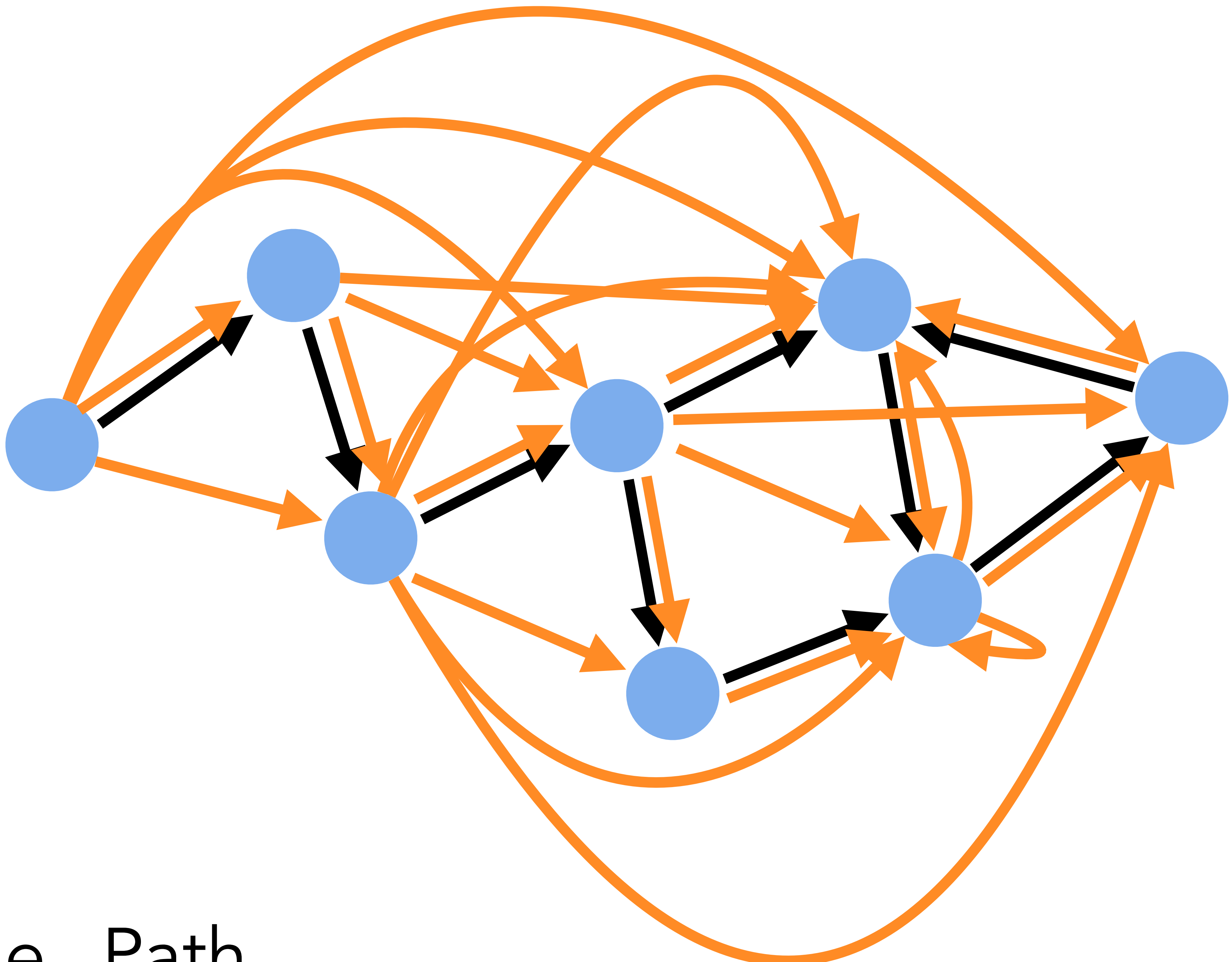


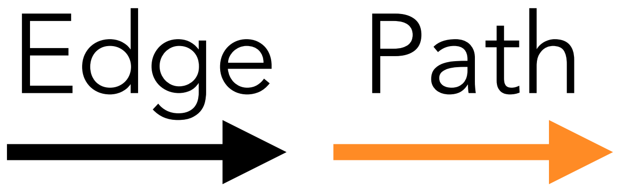
And so on until a fixed point…



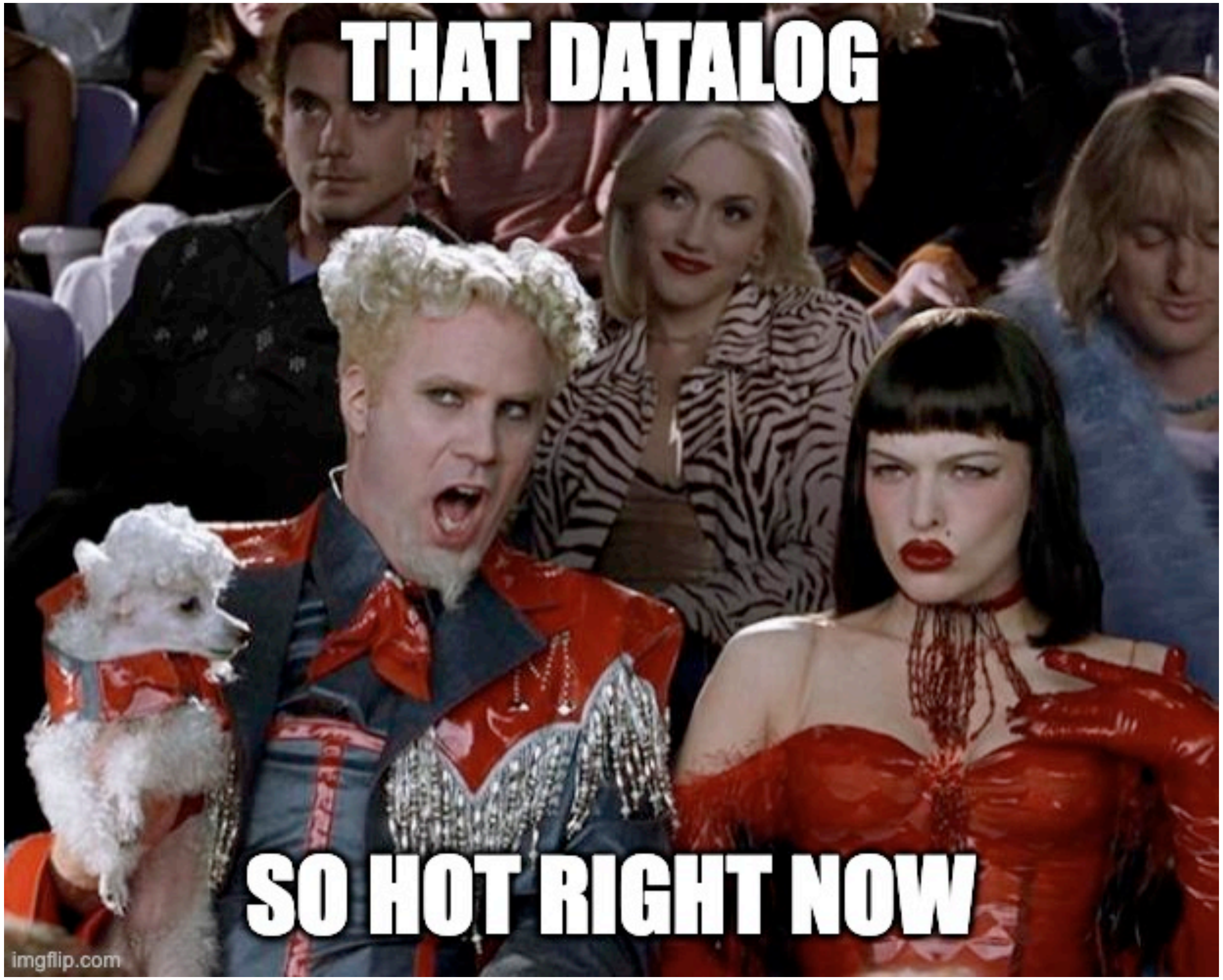
Edge Path

And so on until a fixed point…





If Datalog is just Horn-SAT, why does anyone care?



- Program analysis (DOOP, ddisasm, cclyzer):
    - Orders-of-magnitude less code
    - Let the engine take care of making it fast/parallel
- Graph/social-media mining:
    - Transitive closure/k-clique/…
- Recursive aggregation:
    - Shortest paths, PageRank, …
    - Extends Datalog to non-powerset lattices

```
void a(Foo *x) {
  x.f(0);
}
void b(Foo *x) {
  x.f(1);
}


int main() {
  Baz *baz = new Baz();
  Bar *bar = new Bar();
  a(baz);
  b(bar);
}
```

```
class Foo {
  virtual void f(int x) = 0;
}
class Bar : Foo {
  virtual void f(int x) { return 1 / x; }
}
class Baz : Foo {
  virtual void f(int x) { return 1 + x; }
}
```

Datalog's restrictions enable modern implementations to leverage several tricks to achieve extreme speed:

- Semi-naïve evaluation
- Indexing
- Parallelization
- Efficient tuple representations (locking, space, …)

A modern engine needs **all** of these

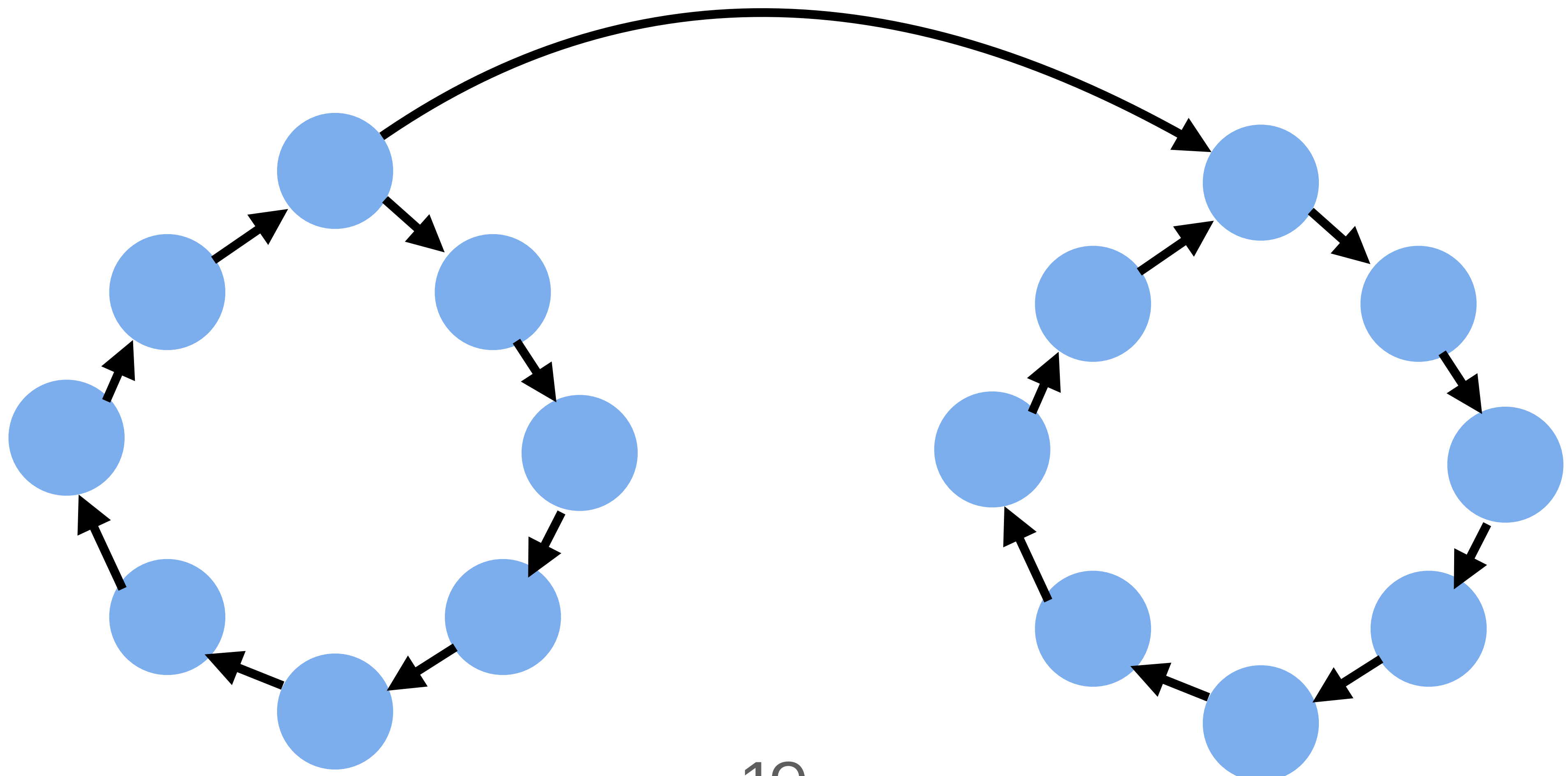
SOTA is nested-loop joins over explicit representations (tries) w/ optimal indexing

So what’s the catch!?

The engine’s data structures can become *leaky abstractions* when the data structure you need isn’t provided by the engine

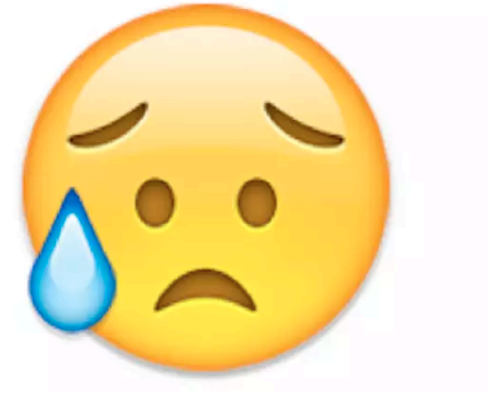
For example, what about two cliques?

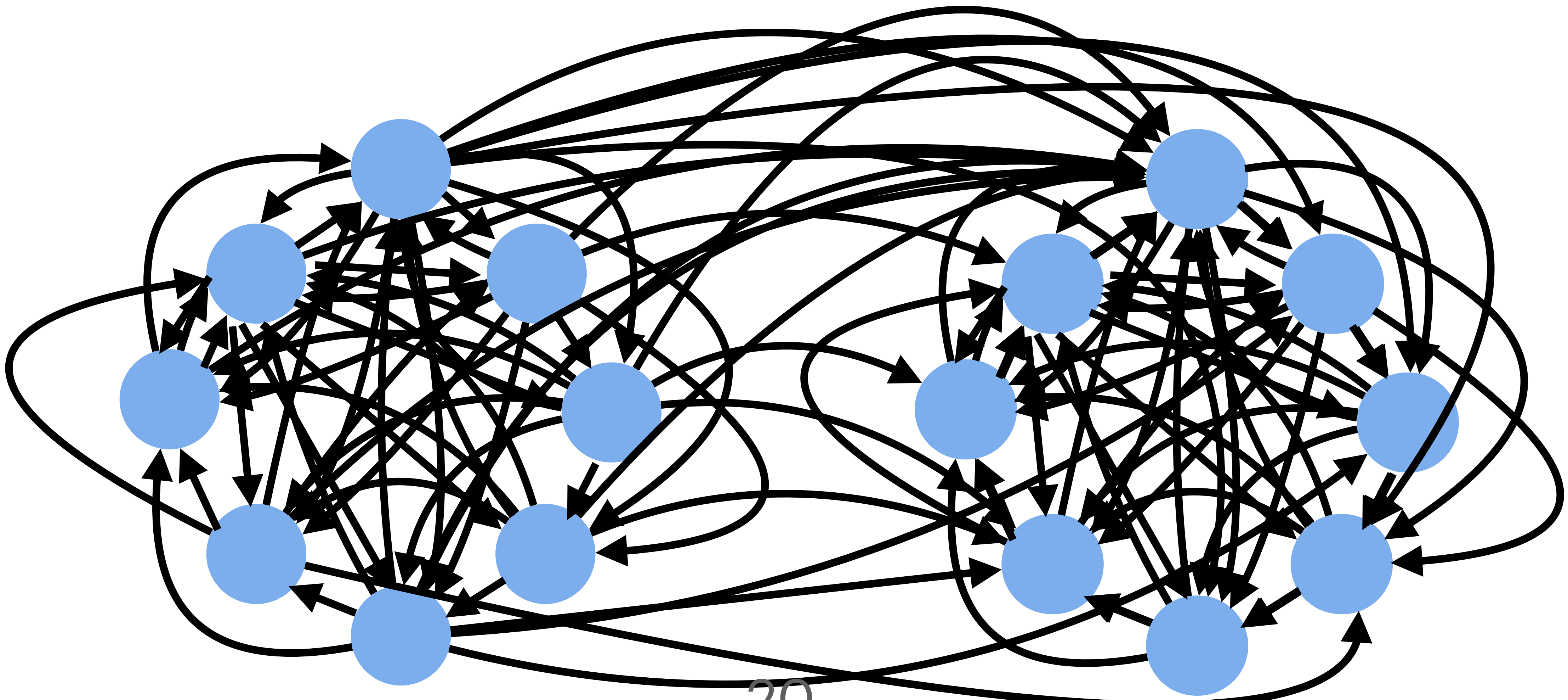
$$\text{Path}(x, y) \leftarrow \text{Edge}(x, y)\,.$$
$$\text{Path}(x, z) \leftarrow \text{Path}(x, y), \text{Edge}(y, z)\,.$$



For example, what about two cliques?

$$\mathrm{Path}(x, y) \leftarrow \mathrm{Edge}(x, y)\,.$$
$$\mathrm{Path}(x, z) \leftarrow \mathrm{Path}(x, y), \mathrm{Edge}(y, z)\,.$$





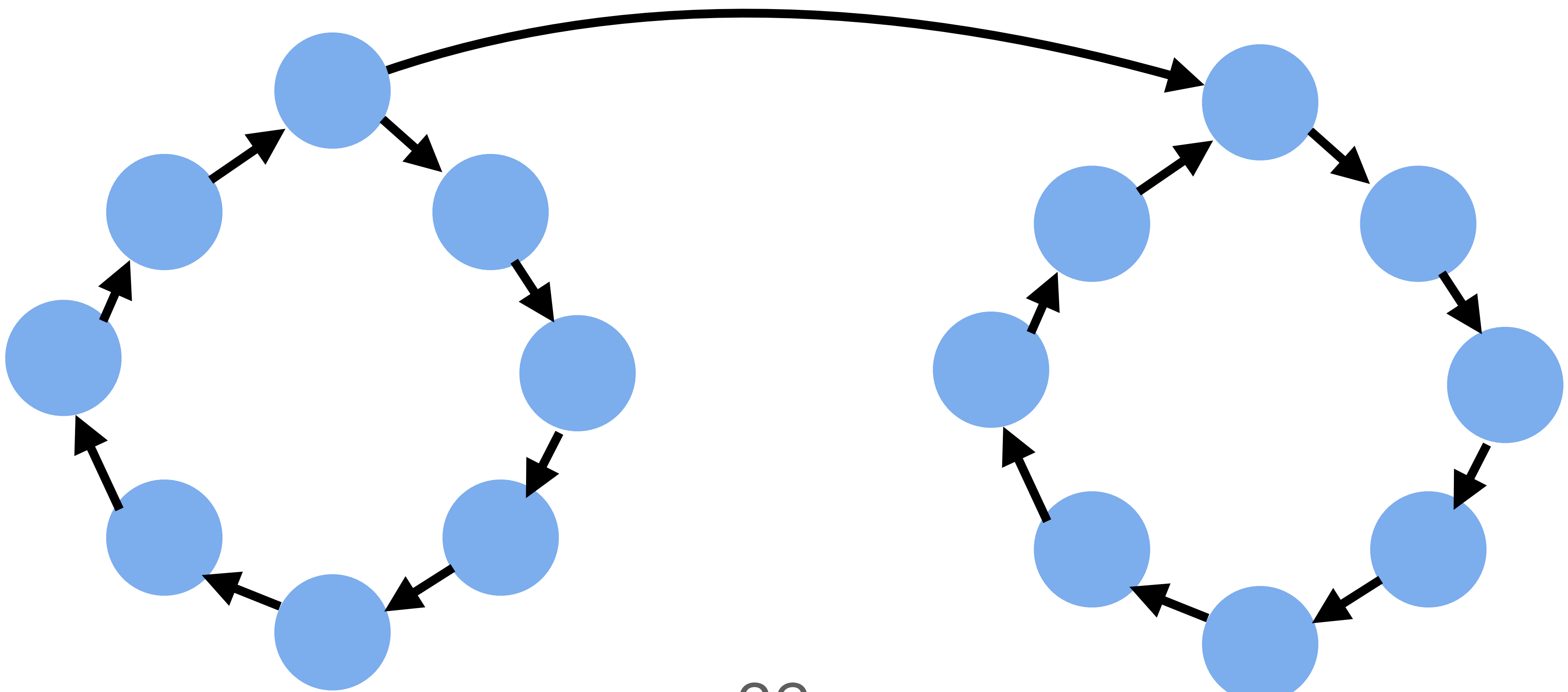
Not just graphs: writing a Steensgaard-style analysis is impossible in Datalog

Some modern engines try to assuage this by offering a fixed set of boutique data structures (e.g., Soufflé's `eqrel`)

We present a new approach, wherein users may "bring their own" data structures to Datalog

In Byods, you can write:

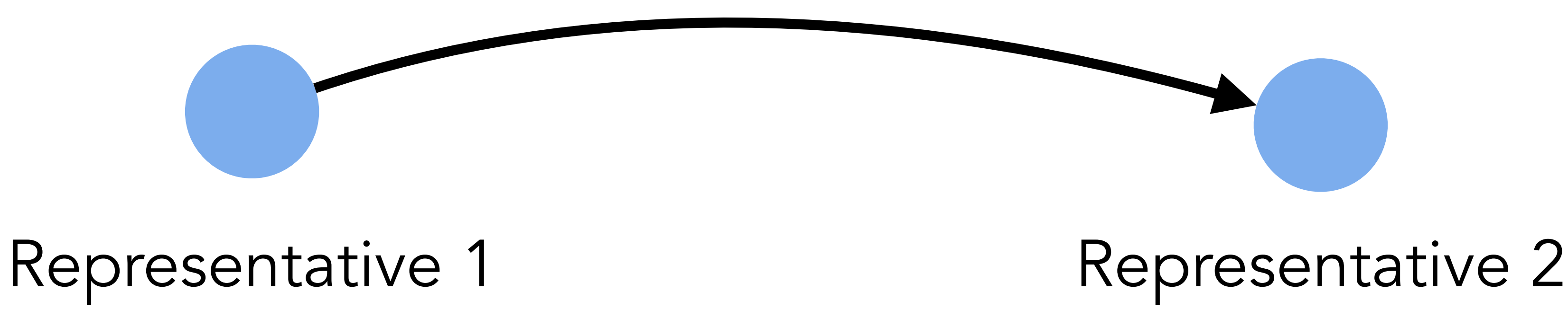
```
#[ds(trrel_uf)]
relation path(Node, Node)
path(x, y) :- edge(x, y).
```



We present a new approach, wherein users may "bring their own" data structures to Datalog

In Byods, you can write:

```
#[ds(trrel_uf)]
relation path(Node, Node)
path(x, y) :- edge(x, y).
```



`trrel_uf` is backed by a union-find-based data structure we defined for transitive relations

Byods is a…

- macro-based Rust-embedded DSL
- which compiles Datalog to efficient kernels
- which hook into user-provided data structures
- via an object protocol
- realized programmatically as Rust traits

We define DL_{DS}, a core formalism for harmonizing user-provided data structures with Datalog.

Data structures are defined via:

$$(D, inj, \gamma)$$

- D is a lattice
- $inj : T \to D$ (T is the relation's tuple type)
- $\gamma : D \to \wp(T)$ is a concretization function

Naïve semantics uses *inj* and γ:

$$T_R(\mathrm{db}) = \sqcup\Big\{ inj_{\,\mathrm{headrel}(R)}\big(\mathrm{head}(R)[\theta]\big) \;\Big|\; \theta : \mathtt{Var} \rightharpoonup \mathtt{Val}.\ \forall \mathtt{b}(xs) \in \mathrm{body}(R).\ xs[\theta] \in \boldsymbol{\gamma}_b(\mathrm{db} \mathbin{@} \mathtt{b})\Big\}$$

Theorem: if all concretization functions are monotonic, then so is the immediate consequence operator of the program as a whole

More details in paper, including semi-naïve semantics of DL_{DS}

We implement DL_{DS} via a two-stage protocol

- (***1st: Compile time***) macros specify (D, *inj*, γ)
  - E.g., `rel_ind` computes type of logical indices
  - `rel_ind_common` computes the carrier type D

E.g., if we define a relation `foo`

```
#[ds(my_provider)] relation foo(Col0, Col1).
```

Then Byods invokes…

```
my_provider::rel_ind_common!(
  foo,          // rel name
  (Col0, Col1), // column types
  [[1]],        // logical indices
  ser,          // parallel or serial
  (),           // user-specified params
)
```

Data structure providers can use arbitrarily-complex logic to construct types for relations/indices

For example, we implement Soufflé's *optimal index selection*, a major and useful optimization pass

The second stage is runtime. Relation-backing data structures must implement four Rust traits:

- `RelIndexRead` — Read from an index
- `RelIndexReadAll` — Iterate over a relation
- `RelIndexWrite` — Write to an index
- `RelIndexMerge` — semi-naïve eval, $\sqcup=$

- We built Byods as an extension to Ascent
- Implements a fully-featured SOTA Datalog
- Including parallelization via Rayon's work stealing
- We observe performance on par with Soufflé
  - Better parallelism / speed, worse memory usage
  - Soufflé still has some innovations we exclude
  - E.g., feedback-directed join planning

Rust Borrow Checker (Polonius)

Implemented in Byods, compare explicit vs. transitive relations vs. index sharing.

Program	LOC	Time (s)					Memory (MiB)		
		explicit	`trrel`	speedup	`ind_share`	speedup	explicit	`trrel`	`ind_share`
clap-rs	2100	8.5	5.1	1.7x	9.65	0.88x	621	328	414
serde-fmt	170	3.74	1.77	2.1x	4.00	0.93x	483	311	360
ascent-codegen	800	1.38	0.65	2.1x	1.14	1.21x	182	125	148
polonius_comp	1000	32	6.2	5.2x	15.8	2.02x	1461	768	1034
chess-search	600	39	14.5	2.7x	26	1.50x	2879	2224	2344

Rust Borrow Checker (Polonius)

Implemented in Byods, compare explicit vs. transitive relations vs. index sharing.

Program	**LOC**	**Time (s)**					**Memory (MiB)**		
		explicit	`trrel`	speedup	`ind_share`	speedup	explicit	`trrel`	`ind_share`
clap-rs	2100	8.5	5.1	1.7x	9.65	0.88x	621	328	414
serde-fmt	170	3.74	1.77	2.1x	4.00	0.93x	483	311	360
ascent-codegen	800	1.38	0.65	2.1x	1.14	1.21x	182	125	148
polonius_comp	1000	32	6.2	5.2x	15.8	2.02x	1461	768	1034
chess-search	600	39	14.5	2.7x	26	1.50x	2879	2224	2344

Transitive relations give ~2x speedup, ~1.5-2x compaction

Rust Borrow Checker (Polonius)

Implemented in Byods, compare explicit vs. transitive relations vs. index sharing.

Program	LOC	Time (s)					Memory (MiB)		
		explicit	`trrel`	speedup	`ind_share`	speedup	explicit	`trrel`	`ind_share`
clap-rs	2100	8.5	5.1	1.7x	9.65	0.88x	621	328	414
serde-fmt	170	3.74	1.77	2.1x	4.00	0.93x	483	311	360
ascent-codegen	800	1.38	0.65	2.1x	1.14	1.21x	182	125	148
polonius_comp	1000	32	6.2	5.2x	15.8	2.02x	1461	768	1034
chess-search	600	39	14.5	2.7x	26	1.50x	2879	2224	2344

Index sharing means more write contention

Transitive Closure via Union-Find

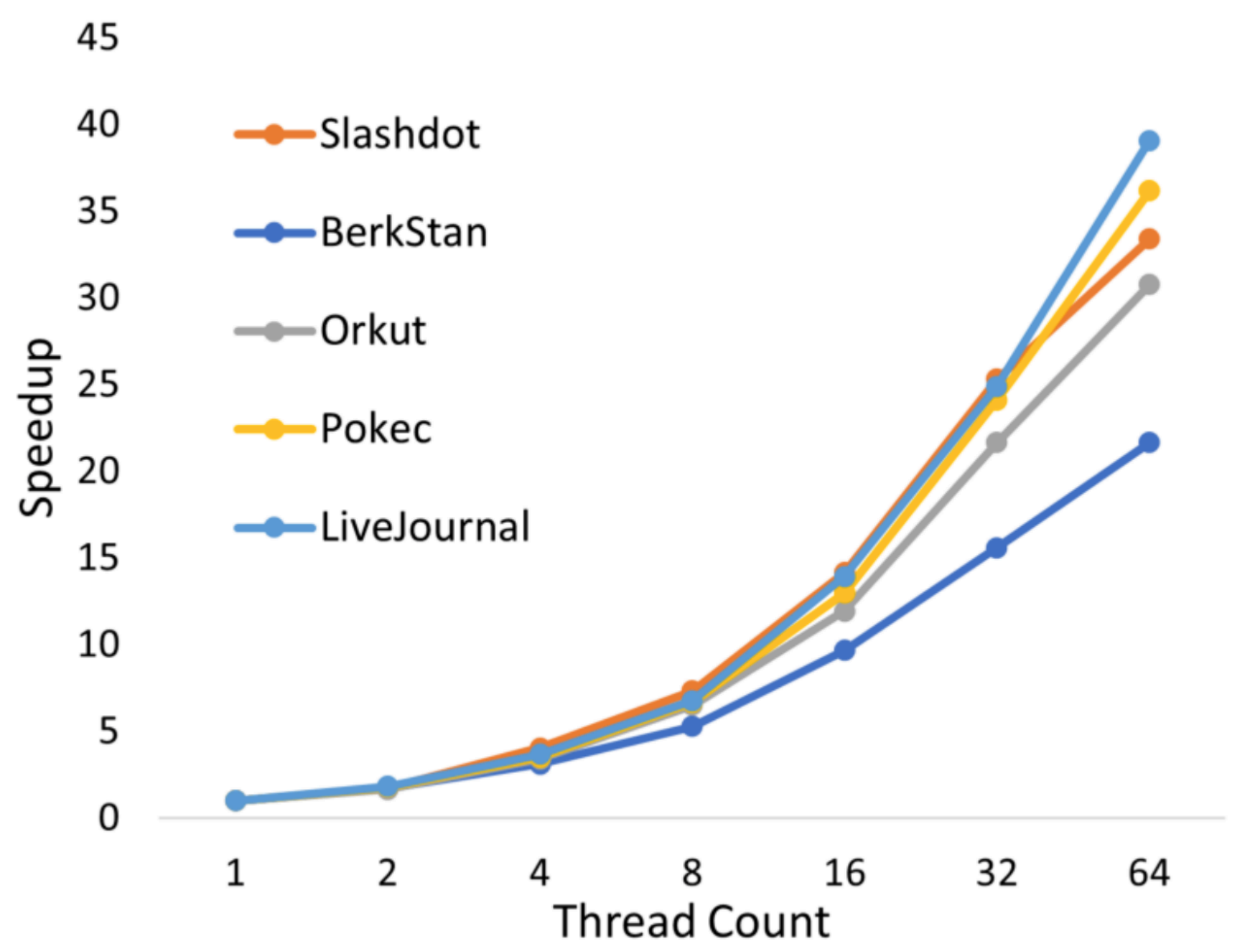
Time improves by 10x, memory by up to 30x!

Graph			Time (s)				Memory (MiB)			
Name	Edges	TC size	`trrel_uf`	`trrel`	explicit	Soufflé	`trrel_uf`	`trrel`	explicit	Soufflé
email-Eu	26K	793K	0.022	0.509	0.844	1.5	1.93	13.4	31.7	39.3
Wiki-Vote	104K	12M	1.8	11.9	10.2	15.0	145	182	533	469
HepTh	52K	74.6M	6.0	51	44	80	49	1222	3590	2408
ca-AstroPh	396K	320M	39	600	792	595	174	5074	15361	12328
BrightKite	428K	3.2B	775	OOM	OOM	OOM	2382	OOM	OOM	OOM

Parallel PageRank

PageRank is an example of *monotonic aggregation*
I.e., deduction in a loop with an aggregator

Graph			**Time (s)**
Name	Nodes	Edges	1 thread
Slashdot	77K	905K	26.3
BerkStan	685K	7.6M	77.2
Orkut	3.1M	117M	838
Pokec	1.6M	30M	904
LiveJournal	4.3M	69M	2004



Scalability generally improves w/ graph size

`yapall` — Yet Another Pointer Analysis for LLVM

- ~2.5kloc analysis of LLVM in Byods + Rust
- Use off-the-shelf LLVM parser via Byods
- Andersen-style m-CFA ("m top stack frames")
- Follows the abstracting abstract machines style
- Comparable to cclyzer (not sound, but `yappal` is)
- Measure impact of index sharing

- 1-context-sensitive analysis of Apache `httpd`
- 5.3 Billion control-flow points
- From 7.3 hours (8 threads) to 2.7 hr (32 threads)

Prog.	k	Pts	Time (s) by thread count								Memory (MiB)	
			8		16		32		64			
			ind	def	ind	def	ind	def	ind	def	ind	def
Jackson	3	10.2M	8.8	10.9	7.6	10.8	7.4	10.1	7.3	9.7	1,940	3,107
	4	164M	128	166	102	164	99	152	94	143	30,466	50,533
Luac	0	3.2M	5.2	4.9	3.8	4.4	3.2	4.4	3.3	5.0	555	732
	1	45M	68	58	48	52	45	50	52	50	3,798	8,000
Lua	0	12.8M	19.6	16.5	13.0	14.6	10.4	14.6	9.7	15.8	1,782	3,222
	1	214M	303	257	193	215	154	199	183	201	14,074	34,335
httpd	0	27.1M	154	103	91	72	65	59	58	61	3,545	6,100
	1	5.39B	26,530	OOM	15,240	–	10,285	–	9,793	–	425,000	OOM
SQLite	0	167M	830	540	471	367	312	284	248	326	26,665	44,597
Redis	0	735M	19,083	11,536	9,210	6,486	5,424	4,087	4,063	3,541	99,500	178,264

- Optimal index sharing cuts memory to 1/2
- Allows httpd to terminate on our 512GB server
- Sometimes faster, sometimes slower (write contention)

Prog.	*k*	Pts	Time (s) by thread count								Memory (MiB)	
			8		16		32		64			
			ind	def	ind	def	ind	def	ind	def	ind	def
Jackson	3	10.2M	8.8	10.9	7.6	10.8	7.4	10.1	7.3	9.7	1,940	3,107
	4	164M	128	166	102	164	99	152	94	143	30,466	50,533
Luac	0	3.2M	5.2	4.9	3.8	4.4	3.2	4.4	3.3	5.0	555	732
	1	45M	68	58	48	52	45	50	52	50	3,798	8,000
Lua	0	12.8M	19.6	16.5	13.0	14.6	10.4	14.6	9.7	15.8	1,782	3,222
	1	214M	303	257	193	215	154	199	183	201	14,074	34,335
httpd	0	27.1M	154	103	91	72	65	59	58	61	3,545	6,100
	1	5.39B	26,530	OOM	15,240	–	10,285	–	9,793	–	425,000	OOM
SQLite	0	167M	830	540	471	367	312	284	248	326	26,665	44,597
Redis	0	735M	19,083	11,536	9,210	6,486	5,424	4,087	4,063	3,541	99,500	178,264

Thanks!

- Possible to get SOTA Datalog as a library
- Engine’s data structures become leaky abstractions
- Instead, ***bring your own*** data structures
- Ascent now public, Byods being merged gradually
- `yapall` recently open-sourced by Galois

`github.com/s-arash/ascent`

`github.com/GaloisInc/yapall`