Horn Clauses and Datalog

CIS700 — Fall 2024

Kris Micinski

We spent significant amounts of time talking about propositional logic (SAT), along with higher-order logics (intuitionistic type theory ala Coq, etc…)

This week we'll talk about *Datalog*, a language that is, with respect to expressivity, a restriction of SAT to Horn clauses, but in a way which enables writing first-order *chain-forward* constraints.

Datalog is less powerful than SAT, but still surprisingly powerful in many applications, and its restrictions enable extracting massive parallelism; our group here has written the fastest and most scalable Datalog engines (to date) on servers, clusters, and GPUs.

Review: Resolution

The resolution rule tells us how to infer new knowledge from preexisting knowledge

$$\frac{P \lor \ldots \lor Q \qquad \neg Q \lor R \ldots \lor}{P \lor \ldots R \lor \ldots}$$

If we derive $\bot$, we know the original formula is tantamount to $\bot$

We can view resolution as giving us the **transitive closure of our current knowledge base to explicate latent implications**

The Problem with Resolution

Resolution may or may not be helpful—it may produce new clauses which are not useful

Churning on unproductive work can be very costly

$$\frac{\overbrace{P \vee \ldots}^{n} \vee Q \qquad \neg Q \vee \overbrace{R \ldots \vee}^{n}}{P \underbrace{\vee \ldots R \vee}_{2n} \ldots}$$

Resolution-based solvers must judiciously select how to apply resolution

In future lectures, we'll see how DPLL, CDCL, and related systems apply resolution intelligently (but heuristically) to scale SAT solving to formulas with tens of thousands of variables and millions of clauses.

Today, we'll focus on a simpler logic programming language based on a restricted form of clauses

Horn Clauses

A horn clause is a clause with at most one positive (i.e., not negated) literal

$\neg B_0 \vee \ldots \vee \neg B_n \vee H$ or, equivalently… $H \leftarrow B_0 \wedge \ldots B_n$

H is the “head” of the clause, and the B_ns are the “body”
"If everything in the body is true, the head must be true”

Datalog

Horn clauses allow **chain forward** reasoning: if the body is true, then the head must be true

The language **Datalog** implements chain forward Horn clauses over a universe of atoms; in this lecture we'll look at Datalog, its foundations and applications, and its implementation

Datalog

- **Declarative** language used to implement analytics queries over large amounts of data

- Extends **SQL** with the ability to deduce “facts”

- For example, starting with an initial database of **edges**, transitively compute a **path** relation

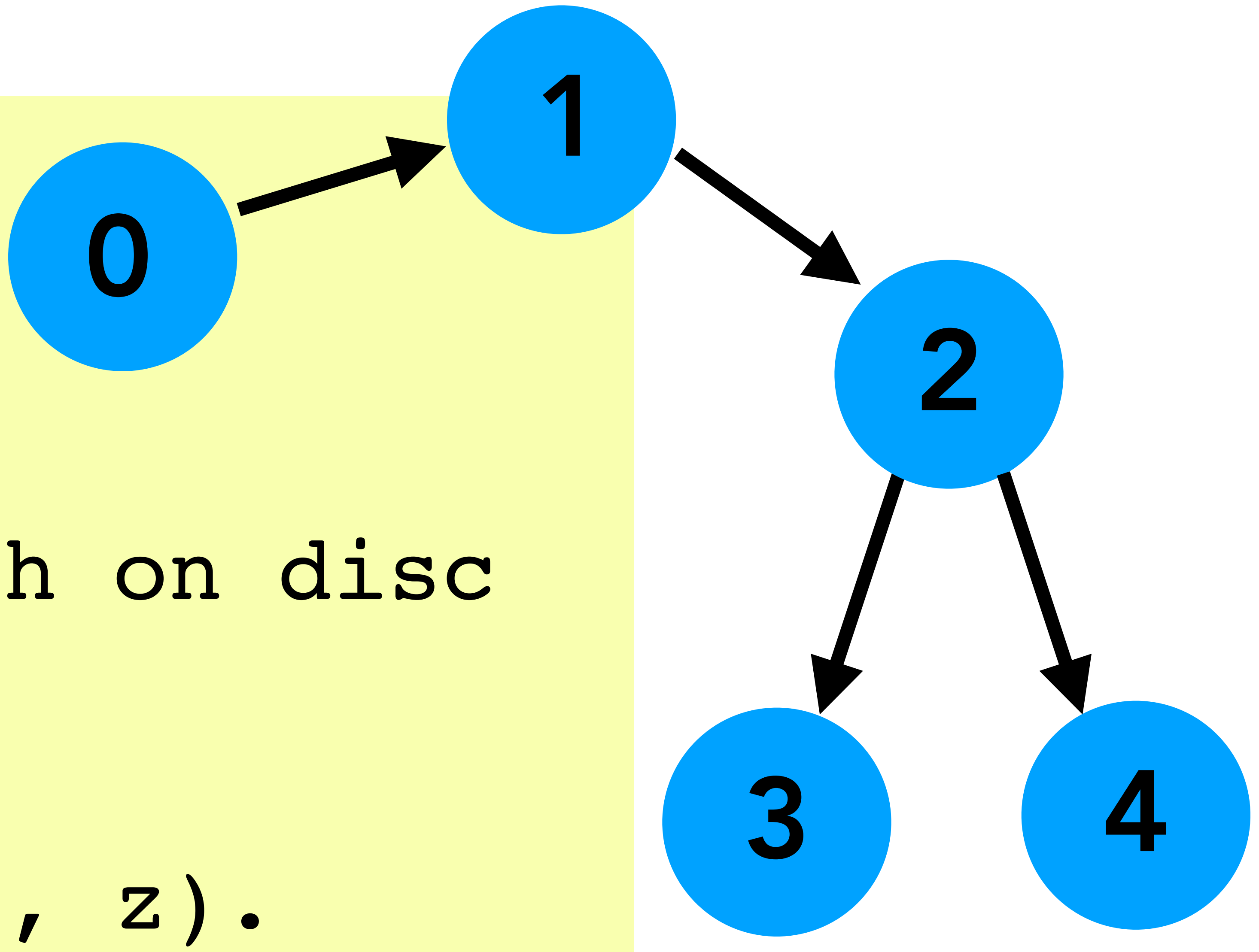
```
.decl edge(x:number, y:number)
.input edge

.decl path(x:number, y:number)
.output path

path(x, y) :- edge(x, y).
path(x, y) :- path(x, z), edge(z, y).
```

Example: Transitive Closure in Soufflé

```
// Transitive Closure
.decl edge(x:number, y:number)
.decl path(x:number, y:number)
.output path // materializes path on disc

path(x, y) :- edge(x, y).
path(x, z) :- path(x, y), edge(y, z).
```

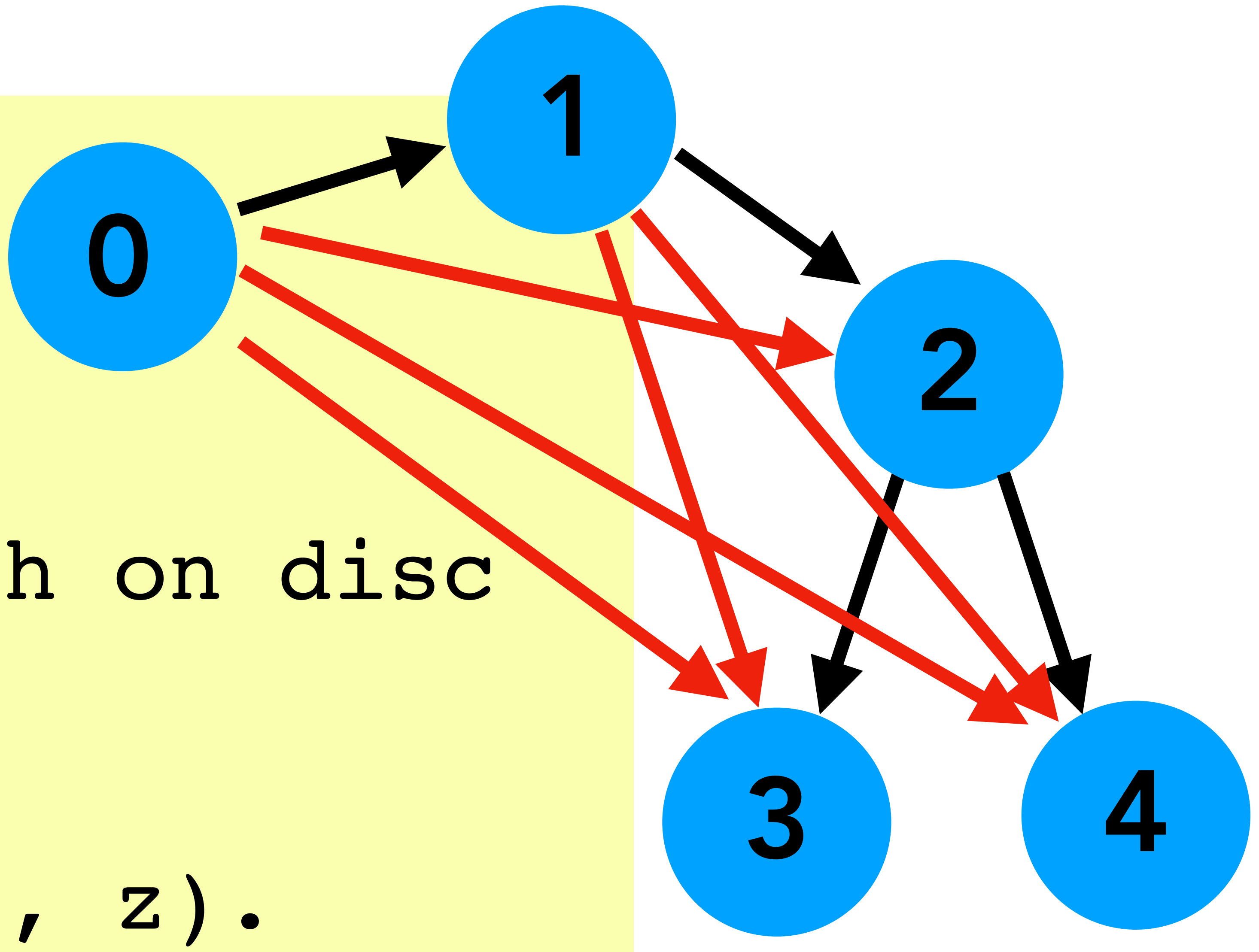
Input: Extensional DataBase (EDB)

```
// Transitive Closure
.decl edge(x:number, y:number)
.decl path(x:number, y:number)
.output path // materializes path on disc

path(x, y) :- edge(x, y).
path(x, z) :- path(x, y), edge(y, z).

// Extensional DataBase (EDB)
edge(0,1). edge(1,2). edge(2,3). edge(2,4).
```



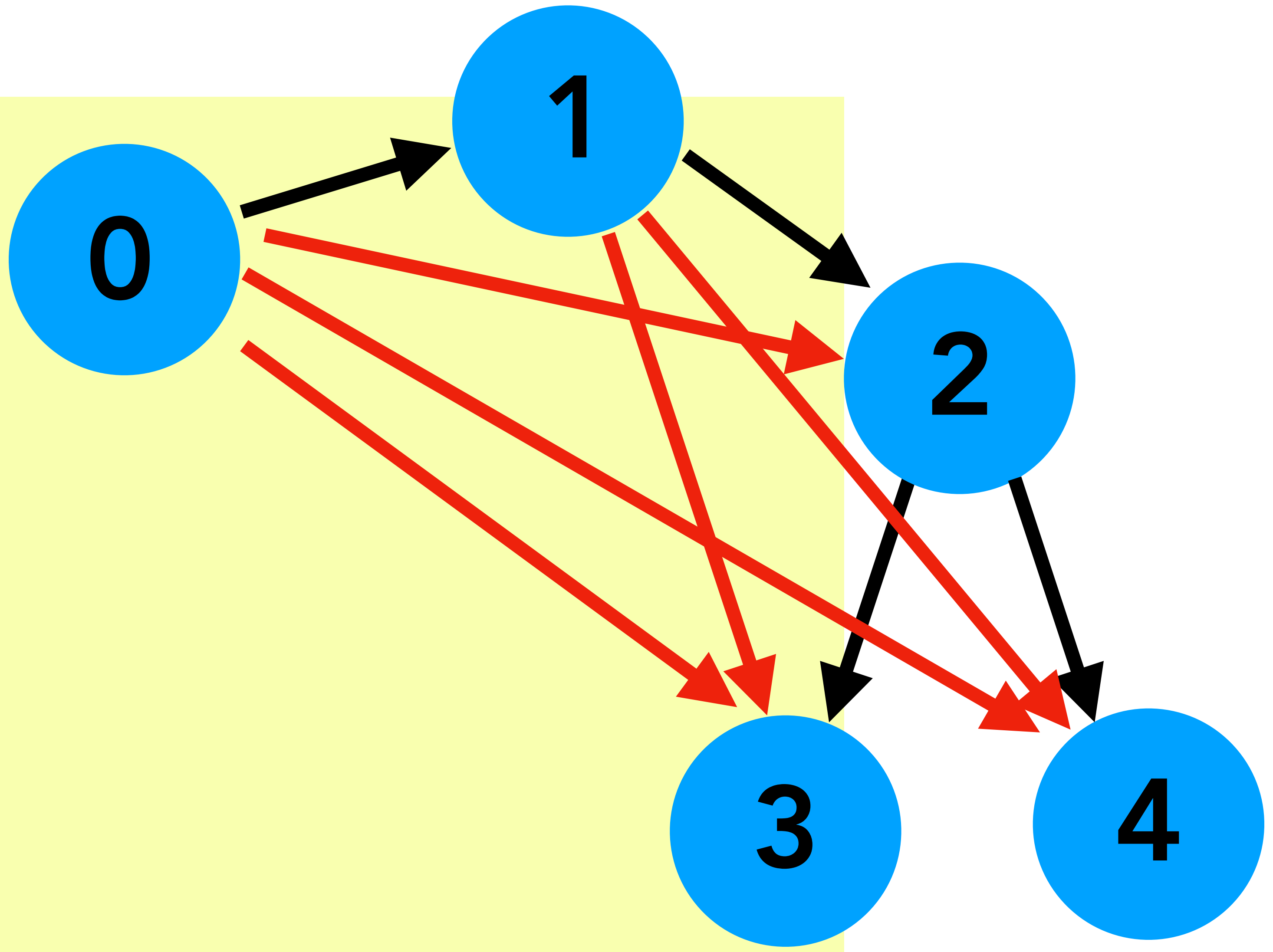
Computation **materializes** the result

```
// Transitive Closure
.decl edge(x:number, y:number)
.decl path(x:number, y:number)
.output path // materializes path on disc

path(x, y) :- edge(x, y).
path(x, z) :- path(x, y), edge(y, z).

// Extensional DataBase (EDB)
edge(0,1). edge(1,2). edge(2,3). edge(2,4).
```



Let’s run it and see

```
kmicinski % souffle tc.dl
kmicinski % cat path.csv
0  1
0  2
0  3
0  4
1  2
1  3
1  4
2  3
2  4
```



Challenge: Triangle Counting, etc…

Write a Soufflé program which takes an input edge of the same form as before. You should output triples

How does this generalize to k-clique (k > 3)?

What is (worst-case) runtime complexity of k-clique, increasing with k? (Hint: k-clique is NP complete!)

Conjunction in the rule heads

A conjunction in the head is technically disallowed:

$$H_0 \land H_1 \leftarrow B_0 \land \ldots B_n$$

But this is only superficial: we can simply refactor this into two rules

$$H_0 \leftarrow B_0 \land \ldots B_n$$

$$H_1 \leftarrow B_0 \land \ldots B_n$$

Disjunction in rule heads

Horn clauses allow **chain forward** reasoning: if the body is true, then the head must be true

Notice that this rules out (a) negation in the body and (b) disjunction in the head; consider the alternative:

$$H_0 \lor H_1 \leftarrow B_0 \land \ldots B_n \qquad H_0 \lor H_1 \lor \neg B_0 \lor \ldots \lor \neg B_n$$

Here, when we know the body is true, we know that either $H_0 \lor H_1$ is true—this means we need to consider *both* possibilities

Extending Datalog to include disjunction in the head is called ***disjunctive Datalog*** and is much more complex

Datalog Programs

- Consist of **facts** and **rules**
- Facts stipulate extensionally-known data
  - Form “input” database, real impls. don’t generally have many facts (instead loaded via CSV)
  - Formal Datalog: facts must be “flat,” i.e., relation arguments must be atoms
- Rules: if everything in the body is true, then head is true

Rules

- Must be Horn-clauses
  - $P(x, \ldots) \leftarrow Q(y, \ldots), R(z, \ldots)$
- There an implicit universal quantification of y, z, …
- Head implied by conjunction (and) of body clauses
- Variables in head must be **ground** (appear in body)
- Negation is **not** allowed, *except* when stratified
  - Stratified negation easy to add metatheoretically: run stratified stuff first; then treat it as an EDB

Datalog Applications — Graph Mining

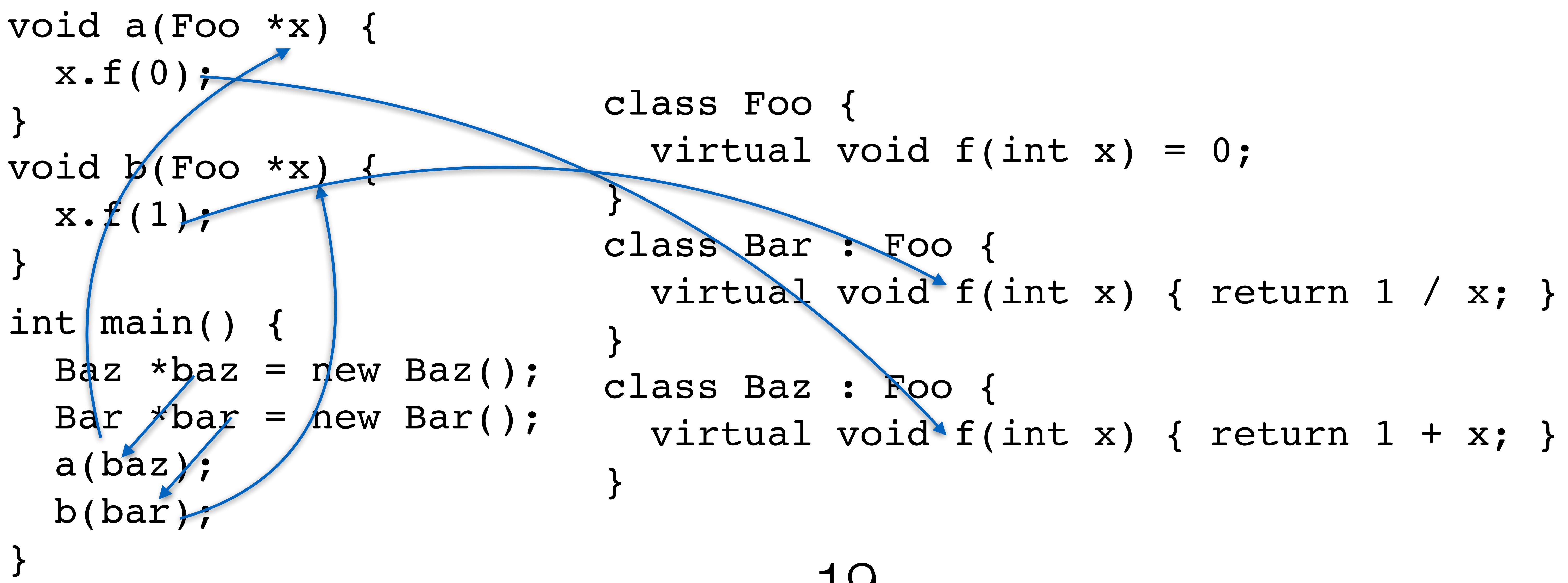
- **k-Clique** computation (e.g., big social network graphs)

```
two_clique(x, y) :- edge(x,y), edge(y,x).
three_clique(x,y,z) :- two_clique(x,y), two_clique(x,z), two_clique(y,z)
four_clique(a,b,c,d) :- three_clique(a,b,c), two_clique(a,d), …
…
```

- **Pagerank, SSSP,** and **Connected Components** can be calculated if we also add *recursive aggregation*
  - Formally, these algorithms
- Datalog a popular implementation target for social-media mining and graph mining broadly.

Datalog Applications — Program Analysis

- Datalog's rough expressive power is reachability-based analyses over graphs, where the graph structure is dynamic
  - Most scalable points-to (and related) analysis to date (DOOP, cclyzer, ddisasm) use Soufflé — fast single-node compilation
    - Scales to hundreds of thousands of lines in hours, variety of experimental context sensitivities



```
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);
}
```

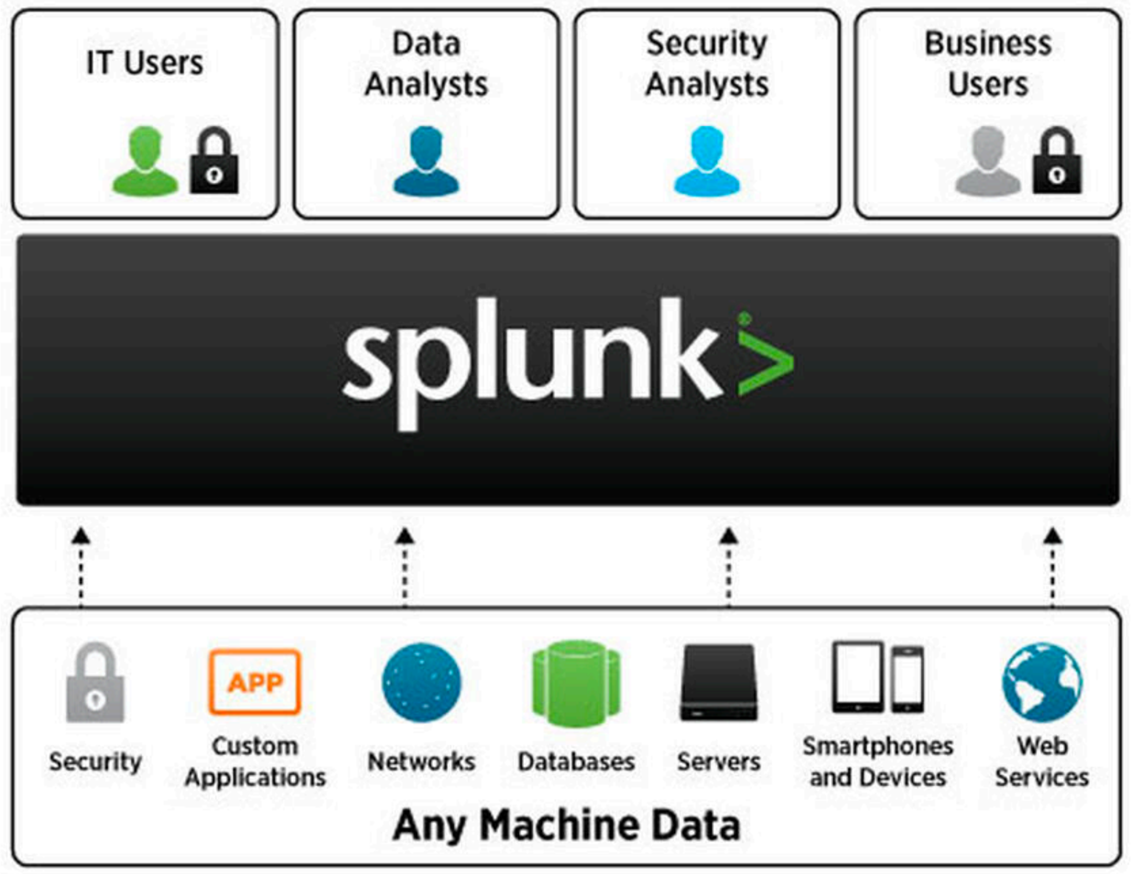
```
class Foo {
  virtual void f(int x) = 0;
}
class Bar : Foo {
  virtual void f(int x) { return 1 / x; }
}
class Baz : Foo {
  virtual void f(int x) { return 1 + x; }
}
```

Datalog Applications — Business Analytics/Databases

- Datalog is roughly the backend structure of many business analytics platforms.

- Lots of industry applications consisting of ad-hoc implementations that scale to things like customer logs, etc…

Datomic: high-speed in-memory (memcached) database via Datalog



Datomic

Data, meet Simple

Model-Theoretic Semantics

- A program P consists of a set of **Rules** and a set of **Facts**. There is a set of **Predicates** whose arguments are variables or ground **Terms**

```
// Facts
edge(0,1). edge(1,2). edge(1,3). edge(2,4). edge(3,4).

// Rules - A <- B /\ C /\ …
path(x, y) :- edge(x, y).
path(x, y) :- path(x, z), edge(z, y).
```

Preds = {path, edge}
Terms = {0,1,2,3,4}

Model-Theoretic Semantics

- A program P consists of a set of **Rules** and a set of **Facts**. There is a set of **Predicates** whose arguments are variables or ground **Terms**

- The **Herbrand base** is the set of all ground instances of predicates from terms in the program

```
// Facts
edge(0,1). edge(1,2). edge(1,3). edge(2,4). edge(3,4).

// Rules – A <- B /\ C /\ …
path(x, y) :- edge(x, y).
path(x, y) :- path(x, z), edge(z, y).
```

Herbrand base is…
{edge(0,0), …, edge(4,4), path(0,0), …, path(4,4)}

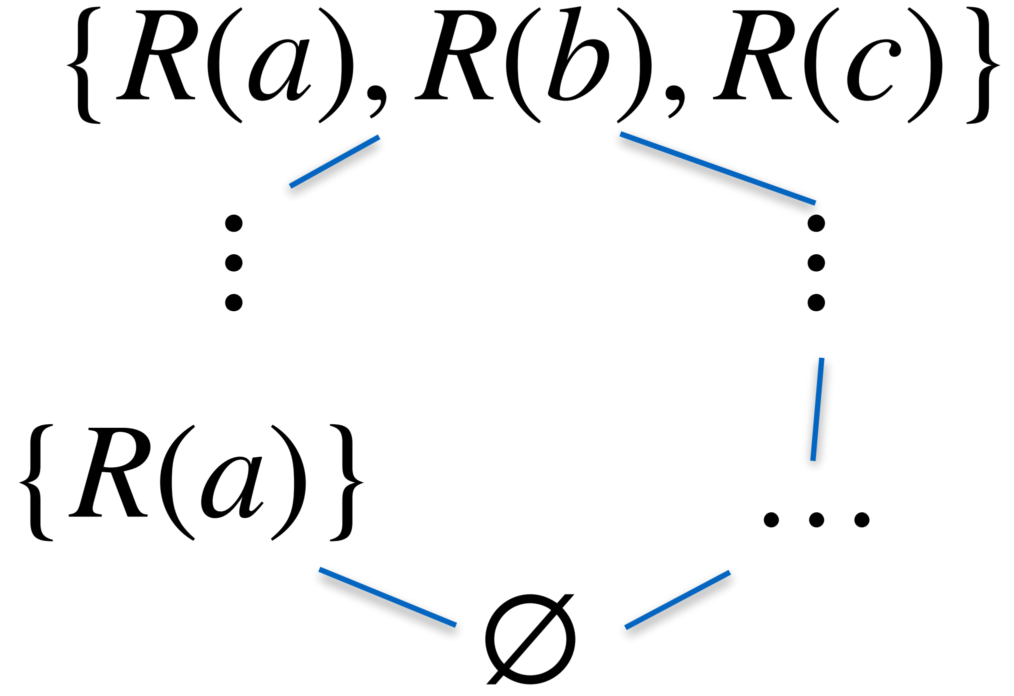
Model-Theoretic Semantics

- A program P consists of a set of **Rules** and a set of **Facts.** There is a set of **Predicates** whose arguments are variables or ground **Terms**

- The **Herbrand base** is the set of all ground instances of predicates from terms in the program

- The Herbrand base forms a *lattice* (it is a set!)—join is ∪, meet is ∩, ordering is via inclusion

- The Herbrand base is **finite**



Herbrand Interpretations

- Any subset of the Herbrand base forms an **interpretation**: a classification of ground atoms as either “true” or “false.”

- Interpretations do not have to be consistent with the program

Herbrand Interpretations

- Any subset of the Herbrand base forms an **interpretation**: a classification of ground atoms as either “true” or “false.”

- Interpretations do not have to be consistent with the program

```
q(1)
p(x) :- q(x)
```

Four possible Herbrand interpretations

{}   {p(1)}   {q(1)}   {p(1),q(1)}

Herbrand *Models*

- An interpretation is a **model** when every rule in the program is satisfied by the model

```
q(1)
p(x) :- q(x)
```

Four possible Herbrand interpretations

{}   {p(1)}   {q(1)}   {p(1),q(1)}

This one is a model

Least Herbrand Models

- Model theory—semantics of P is its **least** Herbrand model
- Many (but not all) larger interpretations may also be models…

```
q(1)
p(x) :- q(x)
r(2)
```

{q(1),r(2),p(1),q(2),p(2)}	Another larger (not least) model
{r(2),q(1),p(1),q(2)}	Not a model (requires p(2))
{q(1),p(1),r(2)}	**Least** Herbrand model for P
{}	Not a model (too small)

Fixed-Point Iteration

- Model theory gives us least-Herbrand models as an extensional representation of a Datalog program

- Computing this least-Herbrand model can be done via a fixed-point (operational, intensional) semantics

- Start with {}, add all facts (equiv: prepare an EDB), and then iterate each rule—Horn clauses force ground implications

```
edge(0,1). edge(1,2).
path(x, y) :- edge(x, y).
path(x, y) :- path(x, z), edge(z, y).
```

$\{\}$

Fixed-Point Iteration

- Model theory gives us least-Herbrand models as an extensional representation of a Datalog program

- Computing this least-Herbrand model can be done via a fixed-point (operational, intensional) semantics

- Start with {}, add all facts (equiv: prepare an EDB), and then iterate each rule—Horn clauses force ground implications

```
edge(0,1). edge(1,2).
path(x, y) :- edge(x, y).
path(x, y) :- path(x, z), edge(z, y).
```

{}   {edge(0,1), edge(1,2)}

Fixed-Point Iteration

- Model theory gives us least-Herbrand models as an extensional representation of a Datalog program

- Computing this least-Herbrand model can be done via a fixed-point (operational, intensional) semantics

- Start with {}, add all facts (equiv: prepare an EDB), and then iterate each rule—Horn clauses force ground implications

```
edge(0,1). edge(1,2).
path(x, y) :- edge(x, y).
path(x, y) :- path(x, z), edge(z, y).
```

{…,path(0,1),path(1,2)}

{}   {edge(0,1), edge(1,2)}

Fixed-Point Iteration

- Model theory gives us least-Herbrand models as an extensional representation of a Datalog program

- Computing this least-Herbrand model can be done via a fixed-point (operational, intensional) semantics

- Start with {}, add all facts (equiv: prepare an EDB), and then iterate each rule—Horn clauses force ground implications

```
edge(0,1). edge(1,2).
path(x, y) :- edge(x, y).
path(x, y) :- path(x, z), edge(z, y).
```

{…,path(0,1),path(1,2)}

{} {edge(0,1), edge(1,2)}

{…,path(0,2)}

Lattice-Theoretic Perspective

- Databases form a *lattice*, a general structure which gives various nice properties (namely, the fixed-point theorems apply to lattices nicely):
  - The underlying set of values is sets of ground tuples
  - Ordered by subset inclusion, i.e., $\sqsubseteq$ (in the lattice) is $\subseteq$
  - The semantics of a Datalog program is given as the least fixed-point of the “immediate consequence” of a program, which is defined as the union of the tuples resulting from applying each rule.
- Claim: the essence of Datalog’s expressive power is the limit of programs you can write as the least fixed-point over power set-generating rules.

The Fixed-Point Theorem

- Let P be a Datalog program composed of a finite set of Horn clauses of the form: r: $A \leftarrow B_1, B_2, \ldots, B_k$

- Let I be an interpretation, i.e., a set of ground atoms representing facts known to be true.

- The immediate consequence operator is a monotonic function: we are accumulating (materializing) a set of facts, so the IC operator is strictly adding facts, *never* removing facts. This makes IC(P), for any Datalog program P, trivially monotone: it always returns a possibly-larger set.

- The fixed-point theorems (Kleene/Tarski) tell us that, because IC is monotone, there is a least fixed-point. In the case of Datalog, there is *always* a finite universe (just form a giant cross product); Thus, Datalog is trivially decidable in principle.

Bag Semantics

- In regular Datalog, the underlying data structure representing a database is a set of tuples. It turns out, if we generalize this structure, we get significantly more expressive power.

- We can generalize this, for example:

  - P(1) <— Q(x,y), R(x)

  - Consider that the input database is {Q(1,1)c1, R(1)c1, Q(1,3)c1}, the output database would be (the input and also) the bag {P(1)c2}, counting P(1) twice. (P(…)ci means count is i)

  - Generalizing Datalog to bag semantics is not too hard: the underlying data is a bag of tuples, databases are ordered by bag inclusion, and things proceed as usual.

Semiring Provenance

- Semiring provenance recognizes that if we generalize our definition of databases from “sets of tuples” to “sets of K-tuples,” where K is an element in some semiring, which are essentially rings where you don’t necessarily have inverses:
    - A semiring is a mathematical object which has $\cdot$ and +
    - There is a zero, which is the identity of +; + commutes and associates as expected from the properties of numbers
    - * has an identity 1, and * associates, but does not necessarily commute

The Semiring Provenance Recipe

- Semiring provenance can be seen as a cookbook technique for building extensions of Datalog wherein tuples are tagged with (differentiated by) values which form a semiring

- “Provenance Semirings” by Green, Karvounarakis, and Tannen is a famous PODS paper which presents this technique:

  - The fixed-point semantics of Datalog nicely generalizes to Datalog over K-relations, for an arbitrary semiring K

  - Many useful semirings. E.g., probabilistic Datalog, which is related to neurosymbolic programming

  - Notice that K may be *infinite*: now, even if we have an underlying finite set of tuples, each tuple is tagged with an element from a semiring—but in practice, the semiring may be infinite!

Semiring Semantics

- Given a semiring K, we annotate each tuple with an element from the semiring in the following manner: each time a tuple is used additively (e.g., a union), we use the semiring + operator, each time a tuple is used multiplicatively (e.g., joins), we use the * operator.
    - Surprisingly, this actually works! Straightforward extension to Datalog, but one that offers profoundly more power without much complication.
- E.g., say we have R(x) ← Q(x,y), R(y) and let the semiring be K, now, for each time we come up with a join result from Q ⋈ R, we obtain its tag via multiplying the tags of the tuples from Q, R
    - If we have Q(1,2)c3 and R(2)c2, we get R(1)c6 (because we have 6 ways we could pair up the different Q(1,2)s and R(2)s).

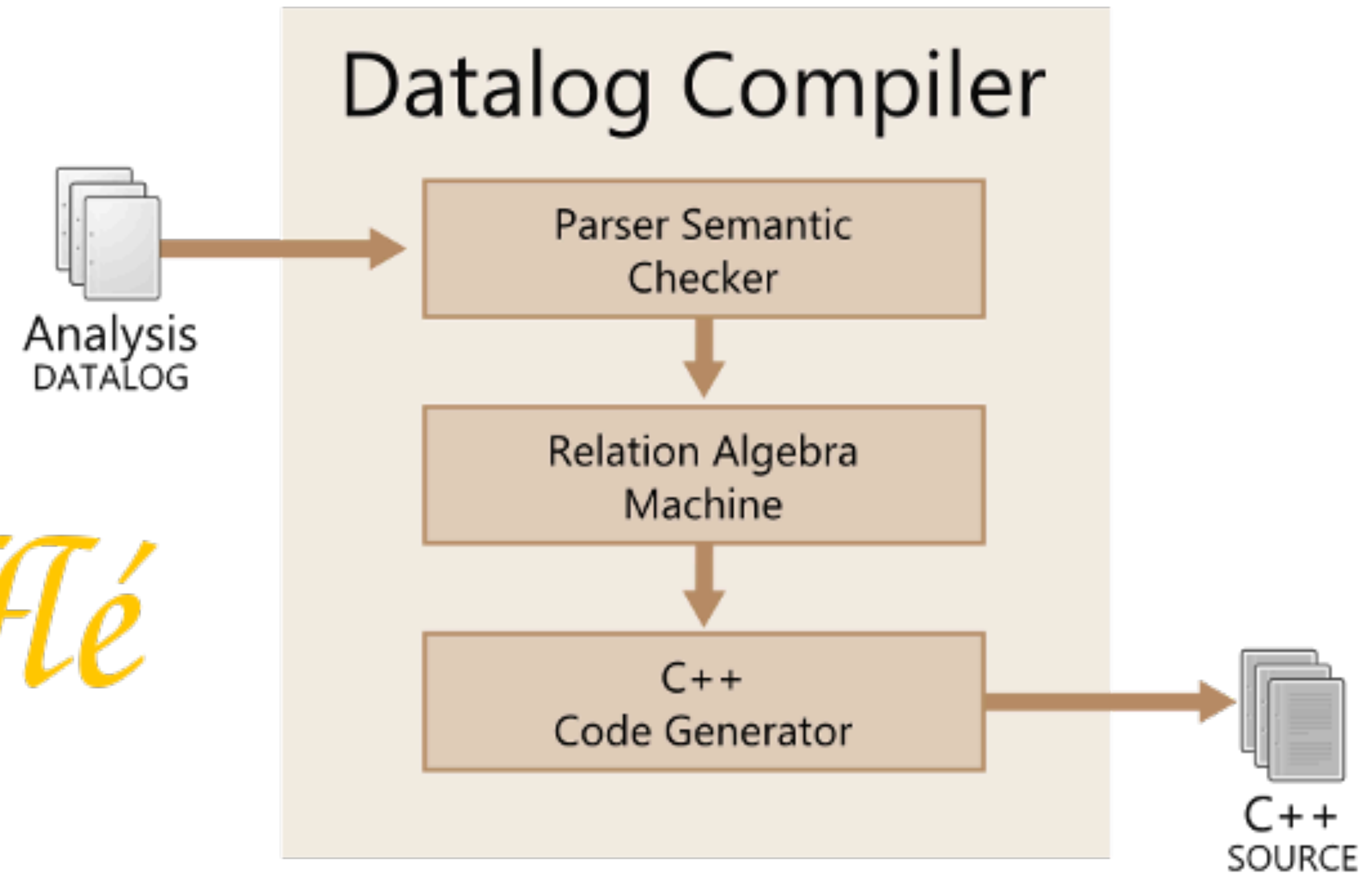
Useful Semirings

- The boolean semiring (Datalog's): $(\{0,1\},\wedge,\vee,0,1)$
- The natural numbers (bag semantics): $(\mathbb{N},+,\times, 0, 1)$
- The tropical semiring (shortest-path semiring): $(\mathbb{R} \cup \infty ,\min,+, \infty, 0)$
- Probabilistic Datalog: $([0,1], \max, \times, 0, 1)$
- The polynomial semiring: $(\mathbb{N}[X], +, *, 0, 1)$
  - Allows us to symbolically track the lineage of facts using symbolic expressions that show which base facts contribute to each derived fact
- The power set semiring ($\mathscr{P}(\mathscr{B}^*)$): finite subsets of strings of boolean variables

Modern Datalog Systems

- Continued resurgences in Datalog: semi-naive evaluation, BDDs (Whaley et al.), compilation to relational algebra
- Modern engines work by generating compiled relational algebra kernels (for loops), pushes stress onto high-performance tuple representation
- Focus on extending Datalog w/ extra-semantic features: ontological reasoning, etc…

Soufflé



Implementing Datalog via RA compilation

- For the rest of the lecture, I will assume we are compiling Datalog to the standard boolean-valued (set) semantics, rather than bemiring-provenance

- In this setting, we typically **compile** Datalog rules into *relational algebra plans* which operationalize each Datalog query as a set of RA operators

- Relational algebra allows working abstractly over relations, and includes selection (σ), join (⋈), projection (π), cartesian product (×), union (∪), etc…

  - To efficiently perform the work, these operators are often *fused* together to produce a combined kernel which does, e.g., projection/selection

- These fused kernels are then iterated in a tight loop—this forms the dominating aspect of the workload, in terms of compute-intensive work

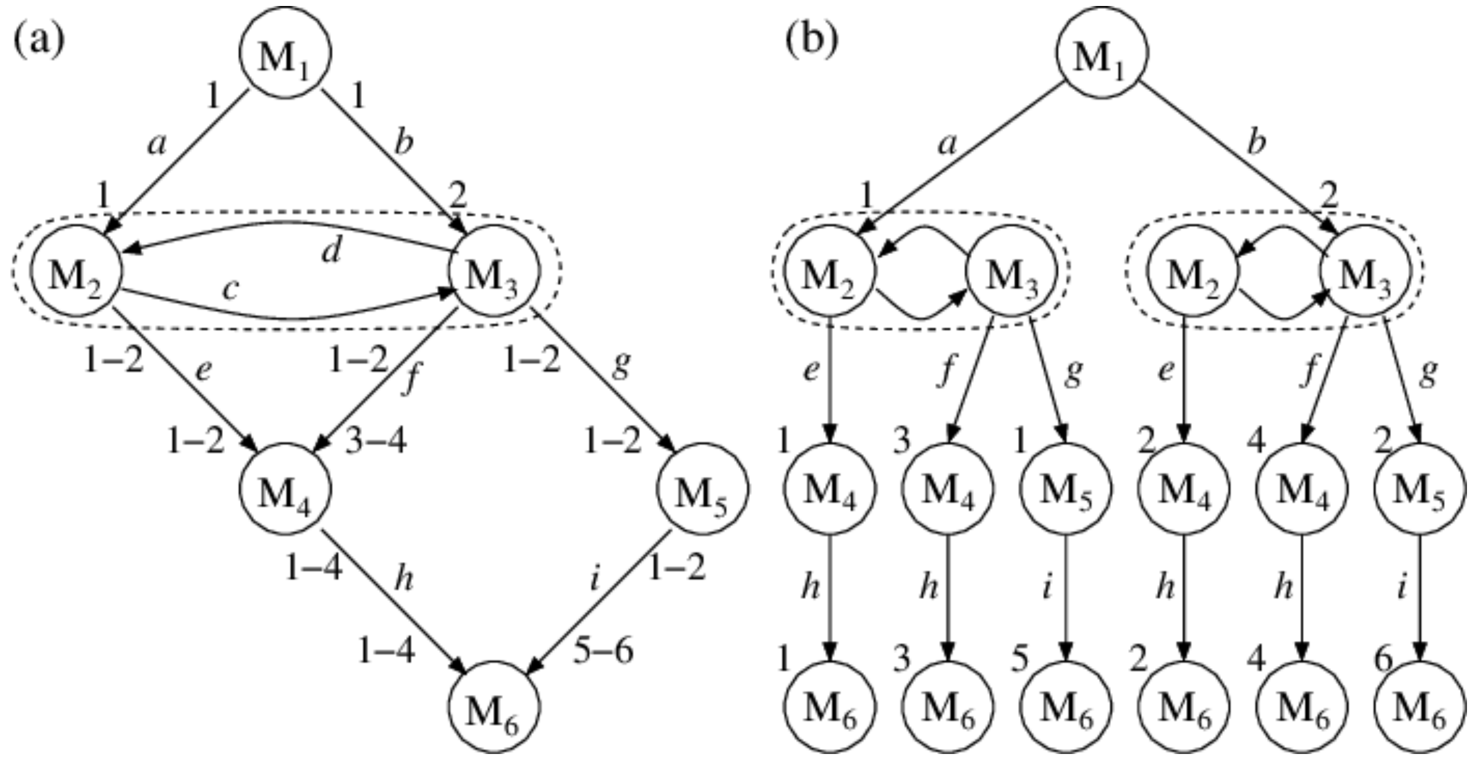
Representing Tuples

- You need a **tuple representation strategy** and a **computation strategy**

- Early 2000s: bddbddb, Whaley and Lam scale inclusion-based alias analysis to Java-sized systems via *Binary Decision Diagrams (BDDs)*

  - Variable ordering posed a significant problem

  - Modern implementations use **relational algebra** w/ **explicit representation** (BTrees and tries)



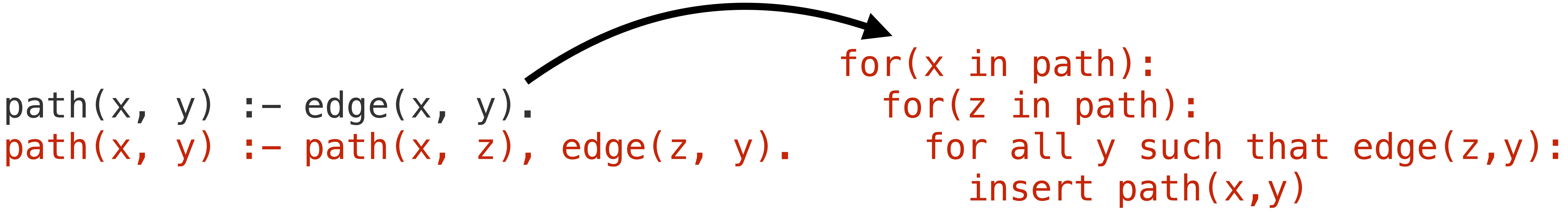
bddbddb

OVER ONE QUATTUORDECILLION RELATIONS SERVED

Joins are compiled

- There have been myriad ways of implementing RA, using a vast number of data structures (BDDs, BTrees, tries, …).

- We will explain the SOTA Datalog representation, which relies upon explicit nested-loop joins (as opposed to a fancier kind of join, e.g., Leapfrog Triejoin)



```
path(x, y) :- edge(x, y).
path(x, y) :- path(x, z), edge(z, y).
```

```
for(x in path):
  for(z in path):
    for all y such that edge(z,y):
      insert path(x,y)
```

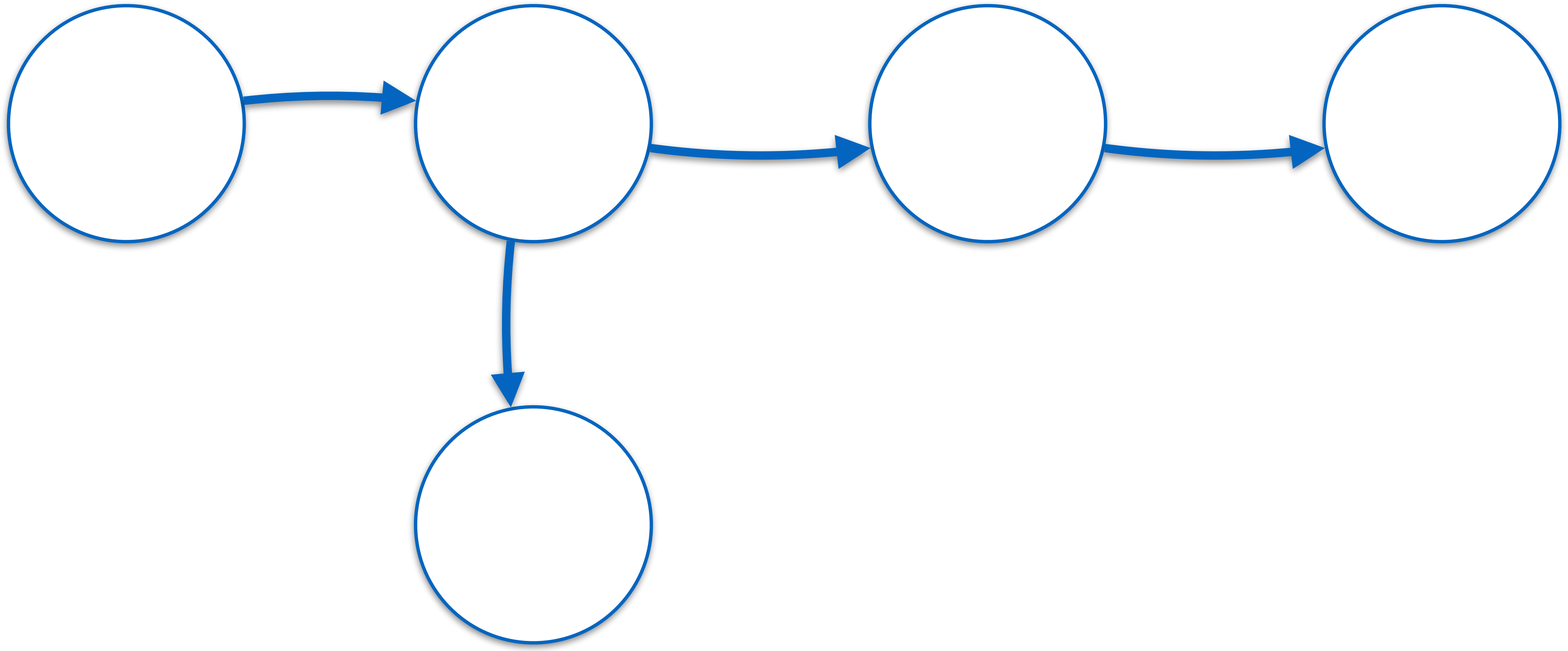
Indexing

- Indexing is crucial for performance: when doing a join like path(x, z), edge(z, y), we need to either:
  - Scan path(x,z) and efficiently select sets of ys for a given z (edge's first arg)
  - Scan edge(z,y) and efficiently select sets of xs for a given z
- This is called *range-indexing* and all modern Datalogs implement it ubiquitously; the downside is that it takes more space: you need a copy of the relation for each index
  - Some index compression techniques, e.g., "Automatic Index Selection for Large-Scale Datalog Computation" (VLDB '19).

Semi-Naive Evaluation

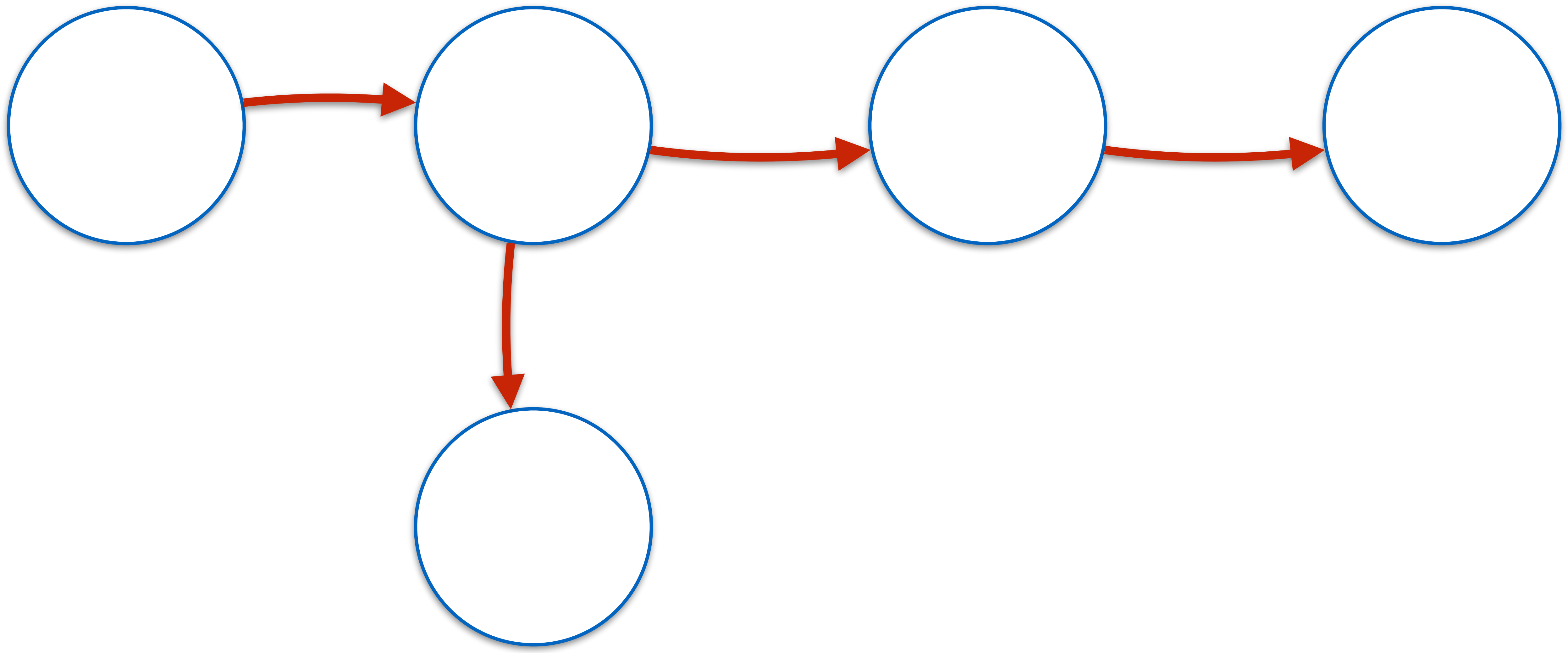
- Each iteration we **reexamine** lots of tuples
- Datalog is *monotonic*: each iteration strictly grows result
- Here: result is monotonically-increasing set of tuples
    - “Sets of tuples” is the **only** lattice DL supports!
- Thus, no need to look at old tuples; only need to consider new tuples that may cause rules to “fire”

Semi-Naive Evaluation



Semi-Naive Evaluation

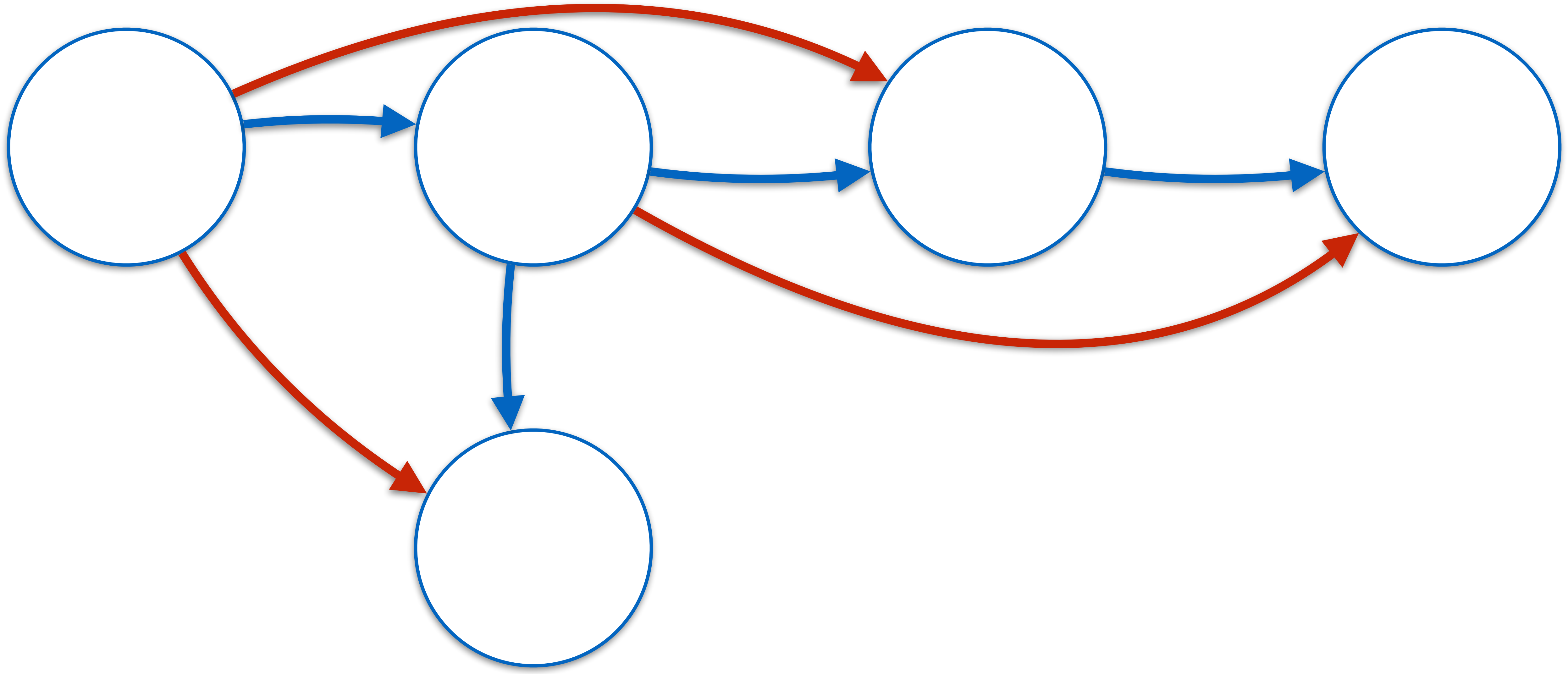
First, discover all edges in `path`



Semi-Naive Evaluation

First, discover all edges in `path`

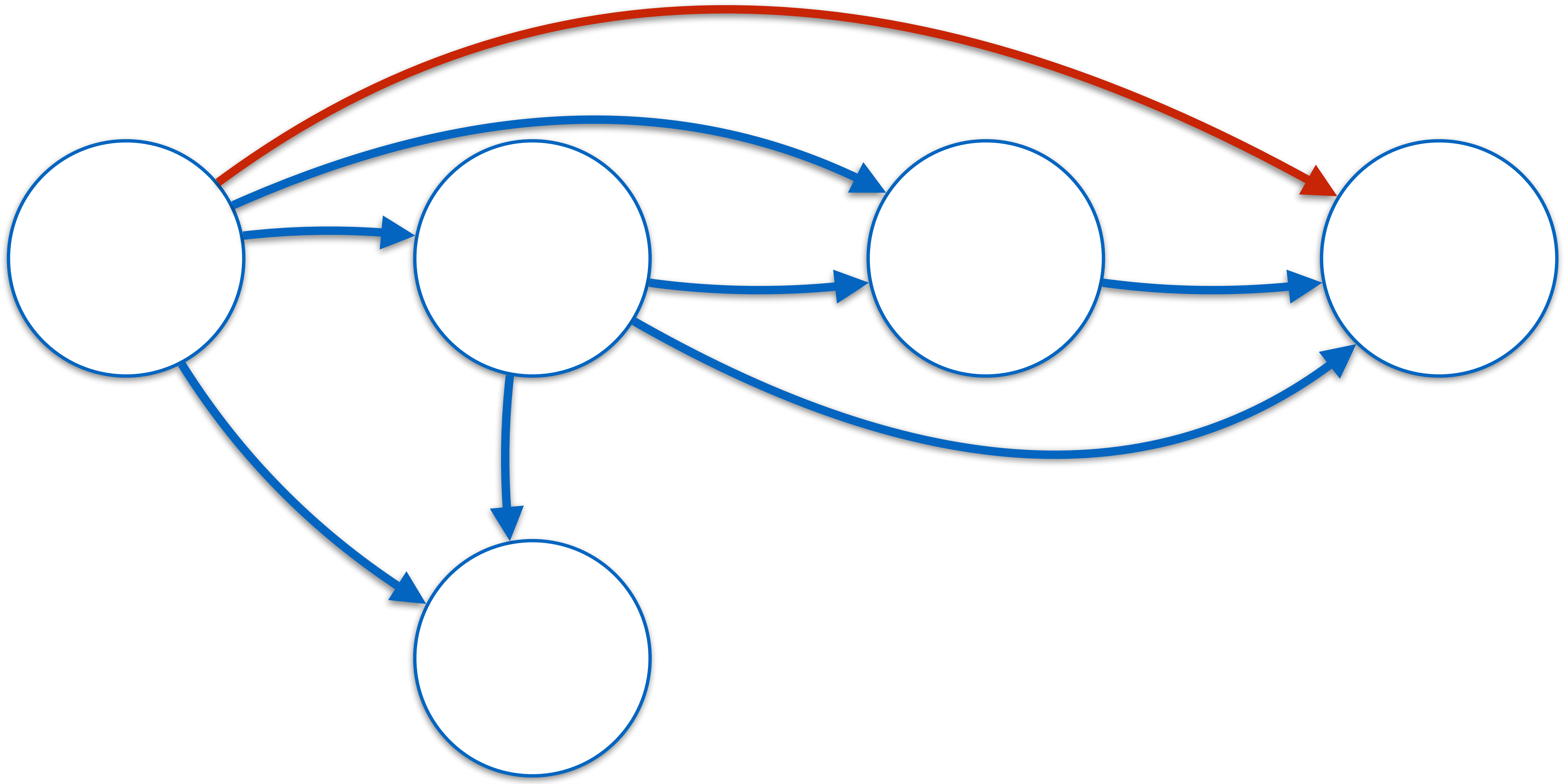
Now, find next iteration…



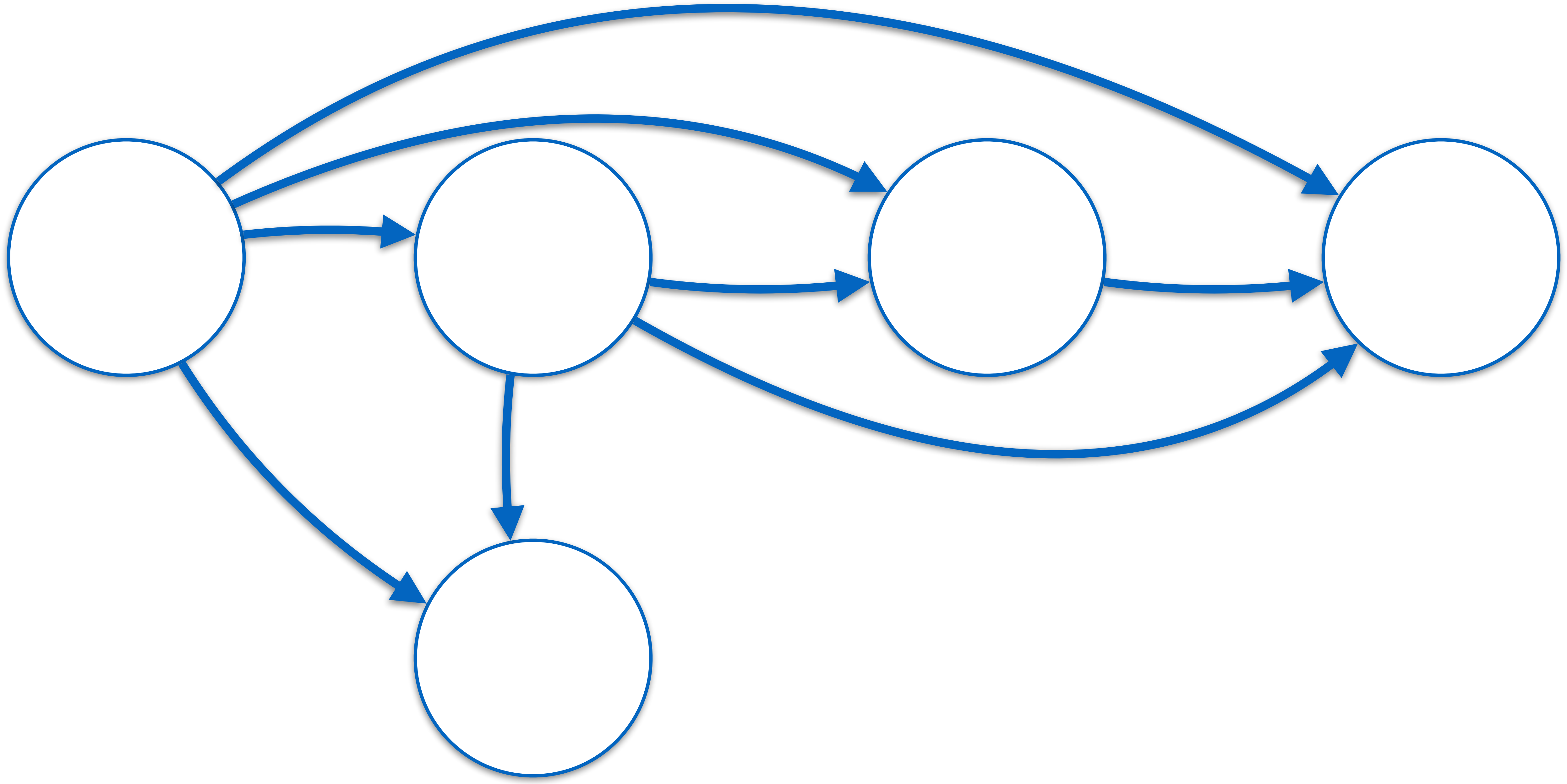
Those all go into Δ, then move into full as a new iteration enters Δ

Semi-Naive Evaluation

Eventually get to a point where nothing new can be discovered…



Semi-Naive Evaluation



At which point `full` contains the result set

Semi-Naïve Evaluation

- Compiler adds delta versions (in below rule: join Δ with full, joining Δ with Δ doesn’t work—would force facts to be discovered at same iteration).
- Heads implicitly add to “fresh” version, which becomes delta at end of each iteration
- After each iteration, delta merged into free; free becomes the “fresh” tuples
- Harmonious with semiring provenance

```
p(x) :- q(x), s(x)
p(x) :- q_delta(x), s_full(x)
```

Slog, massively-parallel Datalog

- Our engine Slog extends Datalog to the distributed setting, associating each tuple with a symbolic identity, giving us a language we call Datalog$^{\exists!}$
  - Facts have a *unique* identity, orthogonal to semiring provenance: allows the program to “introspect” on tuples, and use fact existence to trigger rules, ultimately giving us a paradigm that feels much like functional programming

```
; Rules for free vars
(free x ?(var-ref x))

[(free x ?(app e0 e1))
 <--
 (or (free x e0)
     (free x e1))]

[(free x ?(lambda (args a) e0))
 <--
 (=/= x a)
 (free x e0)]
```

```
; An example:
(lambda (args 0)
  (app (lambda (args 1) (var-ref 2))
       (lambda (args 2) (app (var-ref 3)
                               (var-ref 2)))))
```

Compiling Slog to distributed RA

- We use a distributed hash-index-based strategy for sharding relations across a cluster, with each tuple existing on a single node as dictated by its hash

- Our RA kernels are extended to support Datalog$^{\exists!}$ via fast, communication-avoiding ID generation (intern-key assignment)

- Implemented these parallel RA kernels on top of MPI, which targets supercomputers with InfiniBand, interconnects which offer extremely high speed and low communication latency between nodes

- Our system, Slog1.0 compiled to this MPI-everywhere model, which we found to be effective at parallelizing small programs (e.g., transitive closure) up to 16k+ cores, but did not see as amazing scalability (only up to 512-1k cores) for larger applications (analysis of the Linux kernel)

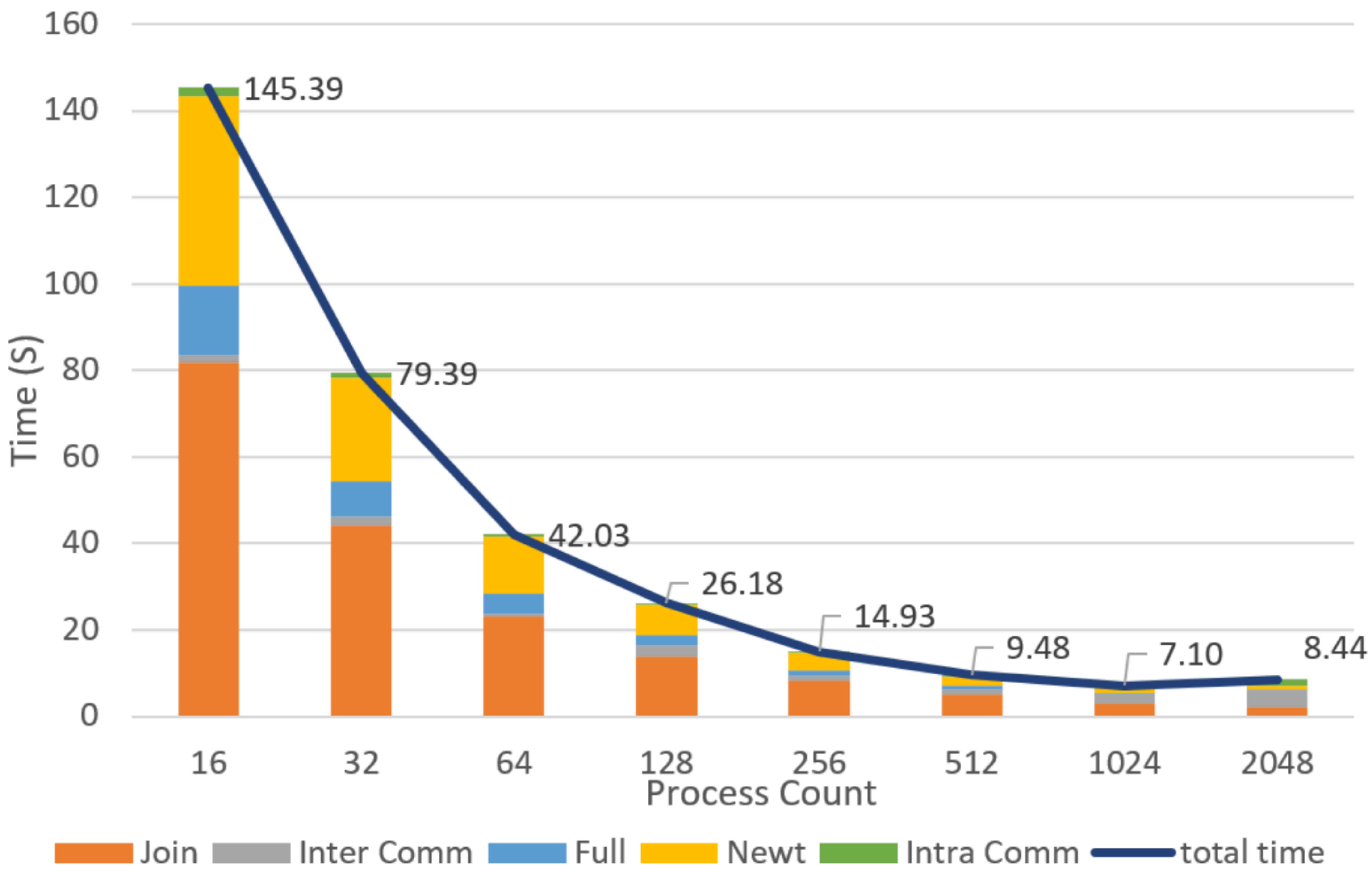


Figure 8: Scaling CSPA (of Linux) on Theta.

Ascent, a Rust-based Datalog

- Arash Sahebolamri implemented Ascent, a Rust crate which implements a modern, SOTA Datalog engine via Rust procedural macros; achieves good performance on a CPU using Rayon's parallelism

- E.g., transitive closure in Ascent:

```
ascent! {
   relation edge(i32, i32);
   relation path(i32, i32);

   path(x, y) <-- edge(x, y);
   path(x, z) <-- edge(x, y), path(y, z);
}
```

- Ascent also invented the "Bring Your Own Data Structures" (OOPSLA '23) technique to integrate user-proved relation-backing data structures

Large LLVM-based context-sensitive points-to analysis in Ascent (Galois' yapall)

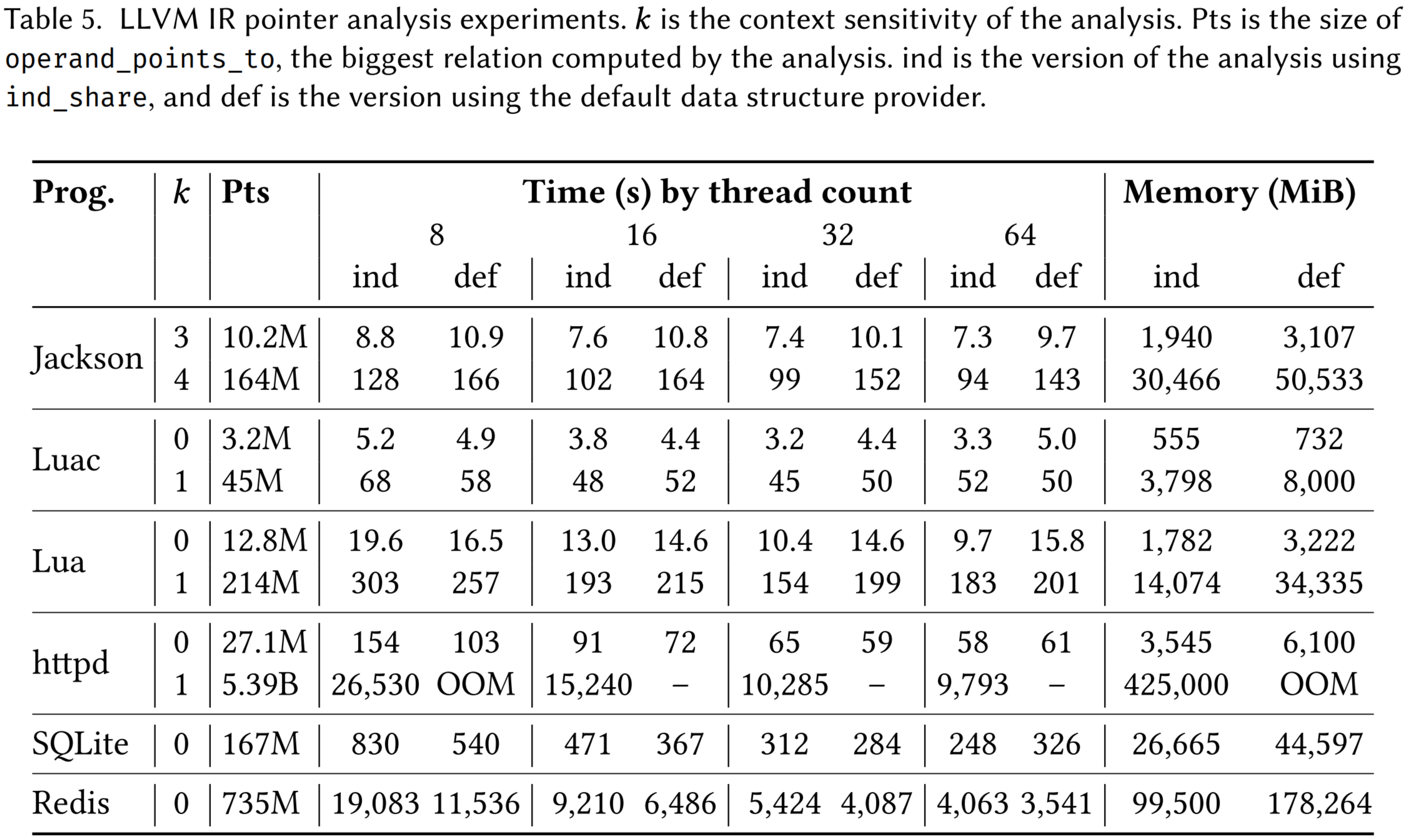
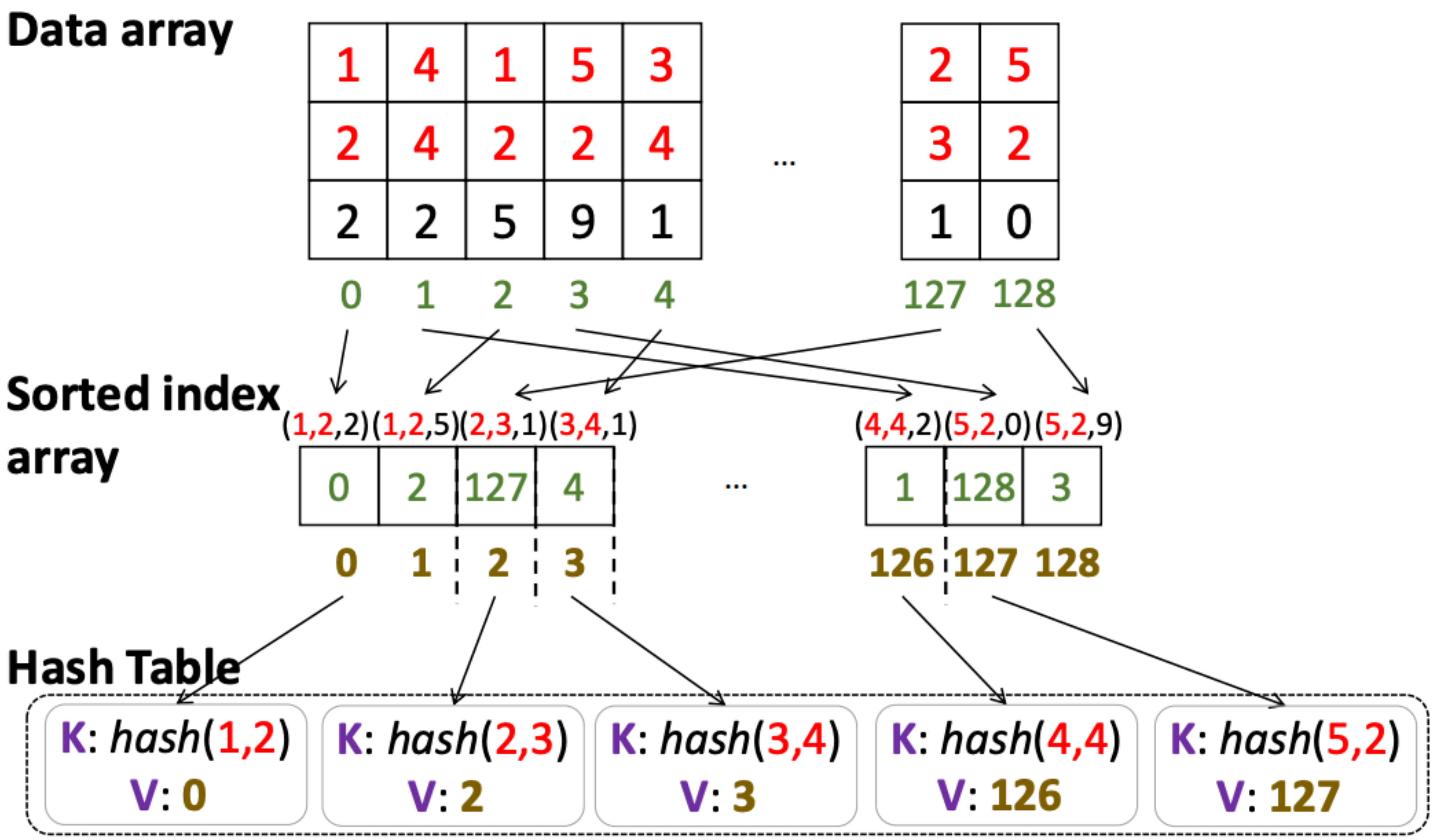


Table 5. LLVM IR pointer analysis experiments. *k* is the context sensitivity of the analysis. Pts is the size of `operand_points_to`, the biggest relation computed by the analysis. ind is the version of the analysis using `ind_share`, and def is the version using the default data structure provider.

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

Compiling to HISA, GPUs, etc…

- In some future lecture, we will talk about compiling Datalog to GPU kernels. Yihao has led this work, and has seen massive improvements using modern GPUs when leveraging HISA, a new GPU-oriented data structure



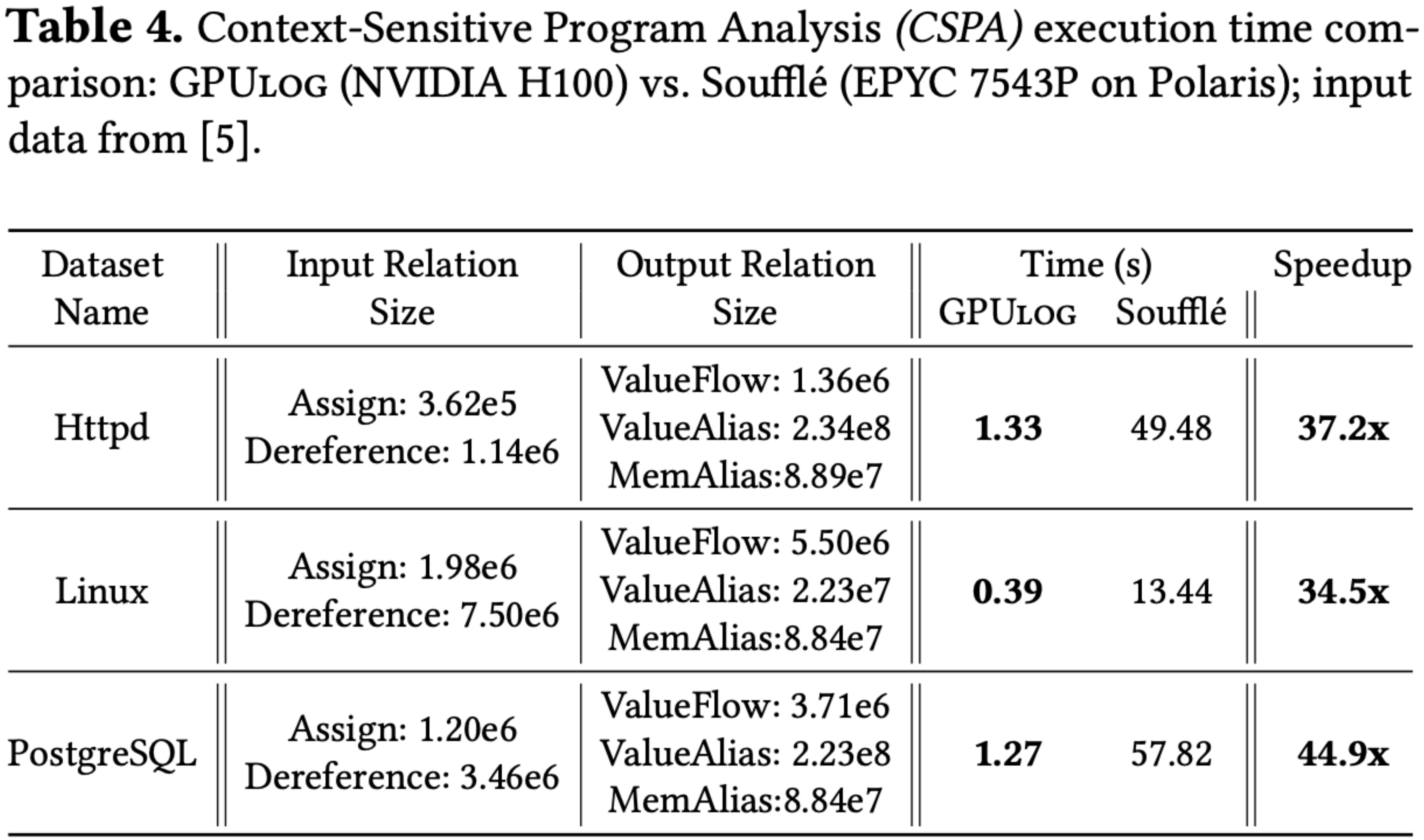


Table 4. Context-Sensitive Program Analysis *(CSPA)* execution time comparison: GPULOG (NVIDIA H100) vs. Soufflé (EPYC 7543P on Polaris); input data from [5].

Dataset Name	Input Relation Size	Output Relation Size	Time (s) GPULOG	Time (s) Soufflé	Speedup
Httpd	Assign: 3.62e5 Dereference: 1.14e6	ValueFlow: 1.36e6 ValueAlias: 2.34e8 MemAlias:8.89e7	**1.33**	49.48	**37.2x**
Linux	Assign: 1.98e6 Dereference: 7.50e6	ValueFlow: 5.50e6 ValueAlias: 2.23e7 MemAlias:8.84e7	**0.39**	13.44	**34.5x**
PostgreSQL	Assign: 1.20e6 Dereference: 3.46e6	ValueFlow: 3.71e6 ValueAlias: 2.23e8 MemAlias:8.84e7	**1.27**	57.82	**44.9x**