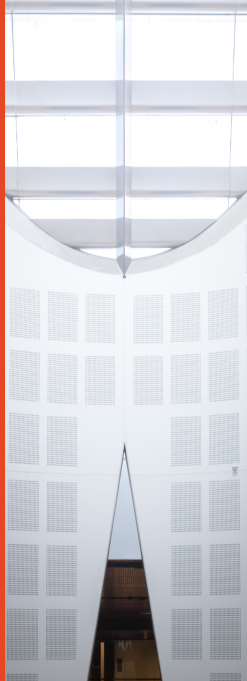
Incremental Datalog Prototype in Soufflé

David Zhao[1] **Bernhard Scholz**[1] **Pavle Subotić**[2]

[1]University of Sydney

[2]Amazon

THE UNIVERSITY OF SYDNEY



Introduction

Introduction

Datalog is a logic programming language, used for

- Program analysis - security property checking in OpenJDK
- Network security analysis - enforcing security rules in AWS networks
- Binary disassembly - Grammatech
- Smart contract analysis - checking for exploits on EVM

Introduction

Datalog is a logic programming language, used for

- Program analysis - security property checking in OpenJDK
- Network security analysis - enforcing security rules in AWS networks
- Binary disassembly - Grammatech
- Smart contract analysis - checking for exploits on EVM

Datalog is *succinct* and *easy to use*

Introduction

Datalog is a logic programming language, used for

- Program analysis - security property checking in OpenJDK
- Network security analysis - enforcing security rules in AWS networks
- Binary disassembly - Grammatech
- Smart contract analysis - checking for exploits on EVM

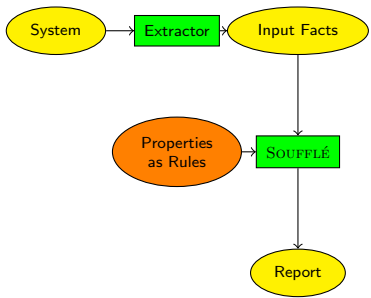
Datalog is *succinct* and *easy to use*

Parallel Datalog engines (e.g. SOUFFLÉ) allow high performance

- Comparable to optimized, hand-crafted tools
- Maintaining ease of use

Soufflé

- System as a set of facts
- System properties as a set of Datalog rules
- Datalog engine (SOUFFLÉ) evaluates rules using facts to check if properties are satisfied



Example - Pointer Analysis

System

```
1  a = new O();
2  b = a;
3
4  a.f = c;
5  d = a.f;
```

Example - Pointer Analysis

System

```
1 a = new O();
2 b = a;
3
4 a.f = c;
5 d = a.f;
```

Input Facts

```
1 alloc(a,l1).
2 assign(b,a).
3
4 store(a,f,c).
5 load(d,a,f).
```

Example - Pointer Analysis

System

```
1 a = new O();
2 b = a;
3
4 a.f = c;
5 d = a.f;
```

Input Facts

```
1 alloc(a,l1).
2 assign(b,a).
3
4 store(a,f,c).
5 load(d,a,f).
```

Output:

- `vpt(a, l1).`
- `vpt(b, l1).`
- `alias(c, d).`
- `...`

Example

Pointer Analysis

```
1  // allocation sites (x = new O())
2  vpt(Var, Obj) :- alloc(Var, Obj).
3
4  // assignments (x = y)
5  vpt(Var, Obj) :- assign(Var, Var2), vpt(Var2, Obj).
6
7  // load-store pairs (v = y.f; p.f = q; y and p alias)
8  vpt(Var, Obj) :- load(Var, Y, F), store(P, F, Q),
9                   vpt(Q, Obj), vpt(P, AliasObj), vpt(Y, AliasObj).
10
11 // generating the alias relation
12 alias(Var1, Var2) :- vpt(Var1, Obj), vpt(Var2, Obj), Var1 != Var2.
```

Incremental Evaluation

Motivation

Security analysis can be slow - up to 2 weeks for OpenJDK

Motivation

Security analysis can be slow - up to 2 weeks for OpenJDK

During software development:

- Make small change to codebase
- Run program analysis

Motivation

Security analysis can be slow - up to 2 weeks for OpenJDK

During software development:

- Make small change to codebase
- Run program analysis

Small change $\rightarrow$ *fast re-run of analysis*

Incremental Evaluation

Small change to system:

$$(input_1 \cup input^+) \setminus input^- = input_2$$

Incremental Evaluation

Small change to system:

$$(input_1 \cup input^+) \setminus input^- = input_2$$

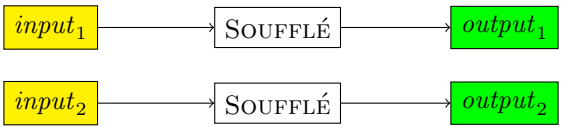
Without incremental:

Incremental Evaluation

Small change to system:

$$(input_1 \cup input^+) \setminus input^- = input_2$$

Without incremental:

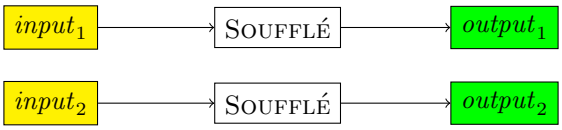


Incremental Evaluation

Small change to system:

$$(input_1 \cup input^+) \setminus input^- = input_2$$

Without incremental:



Incremental Evaluation

Small change to system:

$$(input_1 \cup input^+) \setminus input^- = input_2$$

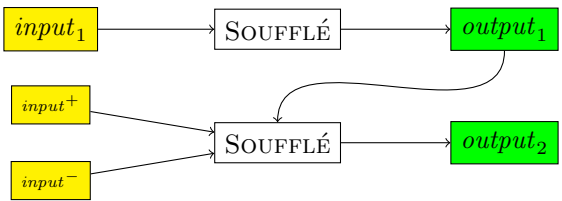
With incremental:

Incremental Evaluation

Small change to system:

$$(input_1 \cup input^+) \setminus input^- = input_2$$

With incremental:

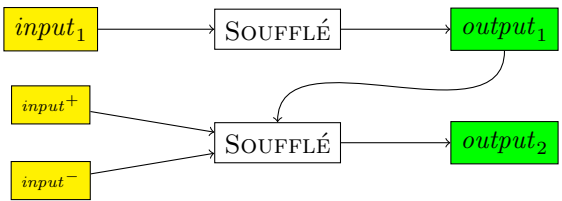


Incremental Evaluation

Small change to system:

$$(input_1 \cup input^+) \setminus input^- = input_2$$

With incremental:



Counting Method

For non-recursive, initially proposed by [Gupta et al., 1993]

Counting Method

For non-recursive, initially proposed by [Gupta et al., 1993]

- Maintain a *count* for number of ways to derive each tuple

Counting Method

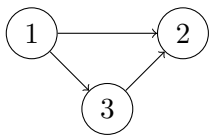
For non-recursive, initially proposed by [Gupta et al., 1993]

- Maintain a *count* for number of ways to derive each tuple

Simpler Example

```
1 hop2(X, Y) :- edge(X, Y).
2 hop2(X, Z) :- edge(X, Y), edge(Y, Z).
```

$\texttt{edge}(1,2), \texttt{edge}(1,3), \texttt{edge}(3,2)$



Counting Method

All Derivations

$$\texttt{hop2}(1,2) \texttt{ :- } \texttt{edge}(1,2)$$
$$\texttt{hop2}(1,3) \texttt{ :- } \texttt{edge}(1,3)$$
$$\texttt{hop2}(3,2) \texttt{ :- } \texttt{edge}(3,2)$$
$$\texttt{hop2}(1,2) \texttt{ :- } \texttt{edge}(1,3), \texttt{edge}(3,2)$$

Counting Method

All Derivations

$$\texttt{hop2}(1,2) \texttt{ :- } \texttt{edge}(1,2)$$
$$\texttt{hop2}(1,3) \texttt{ :- } \texttt{edge}(1,3)$$
$$\texttt{hop2}(3,2) \texttt{ :- } \texttt{edge}(3,2)$$
$$\texttt{hop2}(1,2) \texttt{ :- } \texttt{edge}(1,3), \texttt{edge}(3,2)$$

Relations are multisets instead of sets

$$\texttt{hop2} = \{(1,2) \times 2, (1,3) \times 1, (3,2) \times 1\}$$

Counting Method

Deleting a tuple

$$\Delta \texttt{edge} = \{(1, 2) \times -1\}$$

Counting Method

Deleting a tuple

$$\Delta\texttt{edge} = \{(1,2) \times -1\}$$

```
1  Δhop2(X, Y) :- Δedge(X, Y).
2  Δhop2(X, Z) :- Δedge(X, Y), edge(Y, Z).
3  Δhop2(X, Z) :- edge^ν(X, Y), Δedge(Y, Z). // edge^ν = edge + Δedge
```

Counting Method

Deleting a tuple

$$\Delta\texttt{edge} = \{(1,2) \times -1\}$$

```
1  Δhop2(X, Y) :- Δedge(X, Y).
2  Δhop2(X, Z) :- Δedge(X, Y), edge(Y, Z).
3  Δhop2(X, Z) :- edge^ν(X, Y), Δedge(Y, Z). // edge^ν = edge + Δedge
```

From rule 1: $\Delta\texttt{hop2}(1,2)^{-1}$:- $\Delta\texttt{edge}(1,2)^{-1}$

$$\Delta\texttt{hop2} = \{(1,2) \times -1\}$$
$$\implies \texttt{hop2} = \{(1,2) \times 1, (1,3) \times 1, (3,2) \times 1\}$$

Problems with Recursion

Transitive Closure

```
1 path(X, Y) :- edge(X, Y).
2 path(X, Z) :- edge(X, Y), path(Y, Z).
```

Problems with Recursion

Transitive Closure

```
1 path(X, Y) :- edge(X, Y).
2 path(X, Z) :- edge(X, Y), path(Y, Z).
```

A cycle:
`edge(a,a), edge(a,b), edge(b,a)`

Problems with Recursion

Transitive Closure

```
1 path(X, Y) :- edge(X, Y).
2 path(X, Z) :- edge(X, Y), path(Y, Z).
```

A cycle:
`edge(a,a), edge(a,b), edge(b,a)`

How many times is `path(a,a)` *derived?*
How many times to decrement count when deleting?

Extending to Recursive Datalog

An additional dimension to the counting
Maintain counts *per iteration* of Datalog evaluation

Inspired by Differential Dataflow [McSherry et al., 2013]

Extending to Recursive Datalog

An additional dimension to the counting
Maintain counts *per iteration* of Datalog evaluation

Inspired by Differential Dataflow [McSherry et al., 2013]

$$\text{Iteration}_0 : \{\text{edge}(a,a)^{0,1}, \text{edge}(a,b)^{0,1}, \text{edge}(b,a)^{0,1}\}$$

Extending to Recursive Datalog

An additional dimension to the counting
Maintain counts *per iteration* of Datalog evaluation

Inspired by Differential Dataflow [McSherry et al., 2013]

$$\text{Iteration}_0 : \{\texttt{edge}(a,a)^{0,1}, \texttt{edge}(a,b)^{0,1}, \texttt{edge}(b,a)^{0,1}\}$$
$$\text{Iteration}_1 : \{\texttt{path}(a,a)^{1,1}, \texttt{path}(a,b)^{1,1}, \texttt{path}(b,a)^{1,1}\}$$

Extending to Recursive Datalog

An additional dimension to the counting
Maintain counts *per iteration* of Datalog evaluation

Inspired by Differential Dataflow [McSherry et al., 2013]

$$\text{Iteration}_0 : \{\texttt{edge}(a,a)^{0,1}, \texttt{edge}(a,b)^{0,1}, \texttt{edge}(b,a)^{0,1}\}$$
$$\text{Iteration}_1 : \{\texttt{path}(a,a)^{1,1}, \texttt{path}(a,b)^{1,1}, \texttt{path}(b,a)^{1,1}\}$$
$$\text{Iteration}_2 : \{\texttt{path}(a,a)^{2,1}, \texttt{path}(b,b)^{2,1}\}$$

Extending to Recursive Datalog

An additional dimension to the counting
Maintain counts *per iteration* of Datalog evaluation

Inspired by Differential Dataflow [McSherry et al., 2013]

$$\text{Iteration}_0 : \{\texttt{edge}(a,a)^{0,1}, \texttt{edge}(a,b)^{0,1}, \texttt{edge}(b,a)^{0,1}\}$$
$$\text{Iteration}_1 : \{\texttt{path}(a,a)^{1,1}, \texttt{path}(a,b)^{1,1}, \texttt{path}(b,a)^{1,1}\}$$
$$\text{Iteration}_2 : \{\texttt{path}(a,a)^{2,1}, \texttt{path}(b,b)^{2,1}\}$$

$\texttt{path}(a,a)$ derived:

- Once in iteration 1
- Once in iteration 2

Unrolling the Recursion

Storing per-iteration counts is like unrolling the recursion

```
1 path(X, Y)_1 :- edge(X, Y).
2 path(X, Y)_2 :- edge(X, Y1), edge(Y1, Y).
3 path(X, Y)_3 :- edge(X, Y1), edge(Y1, Y2), edge(Y2, Y).
4 path(X, Y)_4 :- edge(X, Y1), edge(Y1, Y2), edge(Y2, Y3), edge(Y3, Y).
5 ...
```

Our Implementation

Our Ideas

Motivation

Integrate counting inside semi-naïve

- Minimal changes to evaluation strategy

Our Ideas

Motivation

Integrate counting inside semi-naïve

- Minimal changes to evaluation strategy

Maintaining the Δ version of the relation

1. Maintaining Δ explicitly - requires extra indexes and rules (new evaluation strategy)
2. Simulating Δ with extra counts - requires more storage per tuple

Simulating $\triangle$ Relations

Introduce a third dimension for each tuple - *epoch*

Simulating $\triangle$ Relations

Introduce a third dimension for each tuple - *epoch*

1. Initial evaluation is epoch 0
2. Each subsequent insertion/deletion and Datalog re-evaluation is a new epoch

Epoch value for a tuple $\iff$ latest epoch in which it was updated
E.g. $\texttt{path}(a, a)^3 \iff \texttt{path}(a, a)$ was last updated in epoch 3

Simulating Δ Relations

Introduce a third dimension for each tuple - *epoch*

1. Initial evaluation is epoch 0
2. Each subsequent insertion/deletion and Datalog re-evaluation is a new epoch

Epoch value for a tuple $\iff$ latest epoch in which it was updated
E.g. $\texttt{path}(a, a)^3 \iff \texttt{path}(a, a)$ was last updated in epoch 3

Simulating Δ

Epoch value = *current* epoch $\iff$ tuple is in Δ

Challenges

How do we know if an updated tuple is *actually* new?

Challenges

How do we know if an updated tuple is *actually* new?

Reverse Path

```
1 path(X, Y) :- edge(X, Y).
2 path(X, Z) :- edge(X, Y), path(Y, Z).
3 reversepath(X, Y) :- path(Y, X).
```

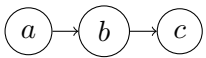
Challenges

How do we know if an updated tuple is *actually* new?

Reverse Path

```
1 path(X, Y) :- edge(X, Y).
2 path(X, Z) :- edge(X, Y), path(Y, Z).
3 reversepath(X, Y) :- path(Y, X).
```

$$\texttt{edge} = \{(a, b)^{0,1}, (b, c)^{0,1}\}$$
$$\texttt{path} = \{(a, b)^{1,1}, (b, c)^{1,1}, (a, c)^{2,1}\}$$



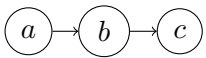
Challenges

How do we know if an updated tuple is *actually* new?

Reverse Path

```
1 path(X, Y) :- edge(X, Y).
2 path(X, Z) :- edge(X, Y), path(Y, Z).
3 reversepath(X, Y) :- path(Y, X).
```

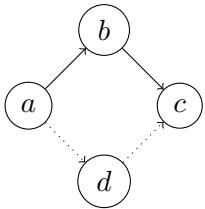
$$\texttt{edge} = \{(a,b)^{0,1}, (b,c)^{0,1}\}$$
$$\texttt{path} = \{(a,b)^{1,1}, (b,c)^{1,1}, (a,c)^{2,1}\}$$
$$\texttt{reversepath} = \{(b,a)^{2,1}, (c,b)^{2,1}, (c,a)^{3,1}\}$$



Challenges

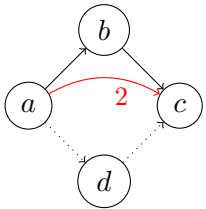
$$\texttt{edge} = \{(a,b)^{0,1}, (b,c)^{0,1}\}$$
$$\texttt{path} = \{(a,b)^{1,1}, (b,c)^{1,1}, (a,c)^{2,1}\}$$
$$\texttt{reversepath} = \{(b,a)^{2,1}, (c,b)^{2,1}, (c,a)^{3,1}\}$$

Consider: $\Delta\texttt{edge} = \{(a,d)^{0,1}, (d,c)^{0,1}\}$



Challenges

$$\texttt{edge} = \{(a,b)^{0,1}, (b,c)^{0,1}\}$$
$$\texttt{path} = \{(a,b)^{1,1}, (b,c)^{1,1}, (a,c)^{2,1}\}$$
$$\texttt{reversepath} = \{(b,a)^{2,1}, (c,b)^{2,1}, (c,a)^{3,1}\}$$

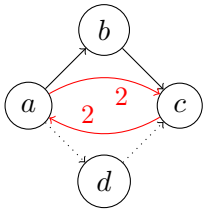


Consider: $\Delta\texttt{edge} = \{(a,d)^{0,1}, (d,c)^{0,1}\}$

$$\Delta\texttt{path} = \{(a,d)^{1,1}, (d,c)^{1,1}, (a,c)^{2,1}\} \implies \texttt{path}(a,c)^{2,2}$$

Challenges

$$\texttt{edge} = \{(a,b)^{0,1}, (b,c)^{0,1}\}$$
$$\texttt{path} = \{(a,b)^{1,1}, (b,c)^{1,1}, (a,c)^{2,1}\}$$
$$\texttt{reversepath} = \{(b,a)^{2,1}, (c,b)^{2,1}, (c,a)^{3,1}\}$$



Consider: $\Delta\texttt{edge} = \{(a,d)^{0,1}, (d,c)^{0,1}\}$

$$\Delta\texttt{path} = \{(a,d)^{1,1}, (d,c)^{1,1}, (a,c)^{2,1}\} \implies \texttt{path}(a,c)^{2,2}$$
$$\Delta\texttt{reversepath} = \{(d,a)^{2,1}, (c,d)^{2,1}, (c,a)^{3,1}\} \implies \texttt{reversepath}(c,a)^{3,2}$$

Challenges

$\Delta\texttt{reversepath}(c, a)^{3,1} \implies \texttt{reversepath}(c, a)^{3,2}$

But actually

The only derivation is:

```
reversepath(c,a) :- path(a,c).
```

Prototype in Soufflé

Implemented in the Soufflé Datalog compiler

- Per-iteration count
- Simulating Δ relations with epoch

```
y = x; a.f = x; y = a.f;

---------------
vpt
x       y       @iteration      @epoch  @count
===============
x       l1      1       0       1
y       l1      2       0       2
a       l2      1       0       1
===============
alias
x       y       @iteration      @epoch  @count
===============
x       y       3       0       1
y       x       3       0       1
===============
Incremental is invoked.
Enter command >
```

Prototype in Soufflé

Implemented in the Soufflé Datalog compiler

- Per-iteration count
- Simulating Δ relations with epoch

~~y = x;~~ a.f = x; y = a.f;

```
Enter command > remove assign("y","x")
vpt
x       y       @iteration      @epoch  @count
===============
x       l1      1       0       1
y       l1      2       -1      1
a       l2      1       0       1
===============
alias
x       y       @iteration      @epoch  @count
===============
x       y       3       0       1
y       x       3       0       1
===============
Incremental is invoked.
Enter command >
```

Prototype in Soufflé

Implemented in the Soufflé Datalog compiler

- Per-iteration count
- Simulating Δ relations with epoch

~~y = x;~~ a.f = x; ~~y = a.f;~~

```
Enter command > remove load("y","a","f")
vpt
x       y       @iteration      @epoch  @count
===============
x       l1      1       0       1
y       l1      2       -2      0
a       l2      1       0       1
===============
alias
x       y       @iteration      @epoch  @count
===============
x       y       3       -2      0
y       x       3       -2      0
===============
Incremental is invoked.
Enter command >
```

Prototype in Soufflé

Implemented in the Soufflé Datalog compiler

- Per-iteration count
- Simulating Δ relations with epoch

```
y = x; a.f = x; y = a.f;

Enter command > insert assign("y","x")
vpt
x       y       @iteration      @epoch  @count
===============
x       l1      1       0       1
y       l1      2       3       1
a       l2      1       0       1
===============
alias
x       y       @iteration      @epoch  @count
===============
x       y       3       3       1
y       x       3       3       1
===============
Incremental is invoked.
Enter command >
```

Preliminary Results

Super early prototype:

- The `reversepath` issue
- B-Tree deletion not yet implemented

Preliminary Results

Super early prototype:

- The `reversepath` issue
- B-Tree deletion not yet implemented

Benchmark: initial evaluation of transitive closure, path graph of 1000 nodes

Soufflé (no inc.)	Soufflé (inc.)	DDlog (Differential Dataflow)
0.3 sec	0.5 sec	18 sec

Preliminary Results

Super early prototype:

- The `reversepath` issue
- B-Tree deletion not yet implemented

Benchmark: initial evaluation of transitive closure, path graph of 1000 nodes

Soufflé (no inc.)	Soufflé (inc.)	DDlog (Differential Dataflow)
0.3 sec	0.5 sec	18 sec

Incremental update: removing edge$(499, 500)$

Soufflé (no inc.)	Soufflé (inc.)	DDlog (Differential Dataflow)
-	0.2 sec	14 sec

Outlook

- We propose some new ideas for incremental evaluation of recursive Datalog by counting, with promising preliminary results
- We need to resolve the `reversepath` issue
- We need to extend to negations and aggregates