

# Topic 1: Data models and query languages

## Unit 4: Datalog

### Lecture 9

Wolfgang Gatterbauer

CS7240 Principles of scalable data management (sp23)

<https://northeastern-datalab.github.io/cs7240/sp23/>

2/7/2023

# Pre-class conversations

- Last class summary
- Please ask questions directly (not in chat)
- Alloy: seems to be declarative, why RA?
- today:
  - Datalog with stratified negation
- next time:
  - Datalog with negation (stable models)

## contributed articles



DOI:10.1145/3338843

**Exploiting a simple, expressive logic based on relations to describe designs and automate their analysis.**

BY DANIEL JACKSON

### **Alloy: A Language and Tool for Exploring Software Designs**

ALLOY IS A language and a toolkit for exploring the kinds of structures that arise in many software designs. This article aims to give readers a flavor of Alloy in action, and some examples of its applications to date, thus giving a sense of how it can be used in software design work.

Software involves structures of many sorts: architectures, database schemas, network topologies, ontologies, and so on. When designing a software system, you need to be able to express the structures essential to the design and to check that they have the properties you expect.

You can express a structure by sketching it on a napkin. That's a good start, but it's limited. Informal representations give inconsistent interpretations, and they cannot be analyzed mechanically. So people have turned to formal notations that define structure and behavior precisely and objectively, and that can exploit the power of computation.

By using formality early in development, you can minimize the costs of ambiguity and get feedback on your work by running analyses. The most popular approach to advocate this is agile development, in which the formal representation is code in a traditional programming language and the analysis is conventional unit testing.

As a language for exploring designs, however, code is imperfect. It's verbose and often indirect, and it does not allow *partial* descriptions in which some details are left to be resolved later. And testing, as a way to analyze designs, leaves much to be desired. It's notoriously incomplete and burdensome, since you need to write test cases explicitly. And it's very difficult to use code to articulate design without getting mired in low-level details (such as the choice of data representations).

An alternative, which has been explored since the 1970s, is to use a design language built not on conventional machine instructions but on logic. Partiality is free because rather than listing each step of a computation, you write a logical constraint saying what's true after, and that constraint can say as little or as much as you please. To analyze such a language, you use specialized algorithms such as model checkers or satisfiability solvers (more on these later). This usually requires much less effort than testing, since you



#### » key insights

- Using a simple logic of relations, Alloy lets you model software designs that involve complex, evolving structures.
- Alloy's tool uses SAT technology to simulate designs and find subtle flaws, and has been used in a wide variety of applications from networking and security to critical systems.
- A key advantage of logical modeling is that you can construct a design incrementally, in an agile way, representing and analyzing only an essential subset of the behavioral constraints.
- Alloy is complementary to a class of tools called model checkers, and is a valuable addition to the software designer's toolkit.

66 COMMUNICATIONS OF THE ACM | SEPTEMBER 2019 | VOL. 62 | NO. 9

- Getting Started ▲
- Welcome
- Install Soufflé
- Build Soufflé
- A Simple Example
- Run Soufflé
- Examples
- Tutorial
- Source Code and Documentation
- Developer Tutorial
- Applications
- Language ▼
- Advanced Topics ▼
- Publications ▼

# Welcome

 Edit me 

Soufflé is a logic programming language inspired by Datalog. It **overcomes some of the limitations in classical Datalog**. For example, programmers are not restricted to finite domains, and the usage of **functors** (intrinsic, user-defined, records/constructors, etc.) is permitted. Soufflé has a component model so that large logic projects can be expressed. Soufflé was initially designed for crafting static analysis in logic at Oracle Labs. Since then, there have been many other applications written in the Soufflé language, including applications in reverse engineering, network analysis and data analytics.

Soufflé provides the ability to rapid prototype and make deep design space explorations possible. A wide range of **applications** have been implemented in the Soufflé language, e.g., static program analysis for Java [DOOP](#), parallelizing compiler framework [Insieme](#), binary disassembler [DDISASM](#), [security analysis for cloud computing](#), and security analysis for smart contracts [Gigahorse](#), [Securify](#), [Secuify V2.0](#), [VANDAL](#). More applications are listed [here](#).

Soufflé language project is led by [Prof Bernhard Scholz](#), and commenced at [Oracle Labs in Brisbane](#). Soufflé was open-sourced in March 2016. It is actively supported by universities and industrial research labs. The main contributors to this project have been [The University of Sydney](#), the [University of Innsbruck](#), the [University College London](#), the [University of Athens](#), [Oracle Labs, Brisbane](#), and many more.

One of the major challenges in logic programming is performance and scalability. Soufflé applies advanced compilation techniques for logic programs. We use a range of techniques to achieve high-performance: Futamura Projections, staged-compilation with a new abstract machine, partial evaluation, and parallelization with highly-parallel data-structures.



# Introduction to Datalog

## Overview

Datalog is a (declarative) logic-based query language, allowing the user to perform recursive queries. It adopts syntax in the style of Prolog. In its pure form, it is based on a decidable fragment of first-order logic (FOL). Here, the universe – the collection of elements by which computation can be performed within – is finite, and functors are not permitted. Applications of Datalog include program analysis, security, graph databases, and declarative networking.

## Soufflé: The Language

### Motivation

The syntax of Soufflé is inspired by implementations of Datalog, namely [bddbddb](#) and [muZ in Z3](#). There is no unified standard for the specification of Datalog syntax. Thus, each implementation of Datalog may differ. A principle goal of the Soufflé project is speed, tailoring program execution to multi-core servers with large amounts of memory. With this in mind, Soufflé provides software engineering features (components, for example) for large-scale logic-oriented programming. For practical usage, Soufflé extends Datalog to make it Turing-equivalent through arithmetic functors. This results in the ability of the programmer to write programs that may never terminate. An example of non-termination is a program where the fact  $A(0)$ . and rule  $A(i + 1) :- A(i)$ . exist without additional constraints. This causes Soufflé to attempt to output an infinite number of relations  $A(n)$  where  $n \geq 0$ . This is in some way analogous to an infinite while loop in an imperative programming language like C. However, the increased expressiveness afforded by arithmetic functors is very convenient for programming.

# Grounded variables



503

However, note that the following example has an *ungrounded* variable:

```
.decl fib(idx:number, value:number)
fib(1,1).
fib(2,1).
fib(idx, x + y) :- fib(idx-1, x), fib(idx-2, y), idx <= 10.
.output fib
```

The reason for this is that variable `idx` is not bound as an argument of a positive predicate in the body. In the example, variable `idx` occurs in the predicates `fib(idx-1, x)` and `fib(idx-2, y)` but as arguments of a functor rather than as a direct argument.

What can be done ?

Source: <https://souffle-lang.github.io/rules>

Datalog example available at: <https://github.com/northeastern-datalab/cs3200-activities/tree/master/souffle>

Wolfgang Gatterbauer. Principles of scalable data management: <https://northeastern-datalab.github.io/cs7240/>

# Grounded variables

However, note that the following example has an *ungrounded* variable:

```
.decl fib(idx:number, value:number)
fib(1,1).
fib(2,1).
fib(idx, x + y) :- fib(idx-1, x), fib(idx-2, y), idx <= 10.
.output fib
```

The reason for this is that variable `idx` is not bound as an argument of a positive predicate in the body. In the example, variable `idx` occurs in the predicates `fib(idx-1, x)` and `fib(idx-2, y)` but as arguments of a functor rather than as a direct argument. To make variable `idx` bound, we can shift the index by one and obtain a program whose variables are *grounded*:

```
.decl fib(idx:number, value:number)
fib(1,1).
fib(2,1).
fib(idx+1, x + y) :- fib(idx, x), fib(idx-1, y), idx <= 9.
.output fib
```



And the program can produce the following output,

```
-----
fib
idx  value
-----
1    1
2    1
3    2
4    3
5    5
6    8
7   13
8   21
9   34
10  55
-----
```

# Grounded variables



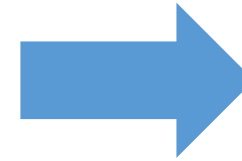
fibonacci

```
souffle -F. -D. fibonacci.dl
```

fibonacci.dl

```
.decl fib(key:number, value:number)
.output fib

fib(1, 1).
fib(2, 1).
fib(id+2, x+y) :- fib(id, x), fib(id+1, y), id <= 13.
```



# Grounded variables



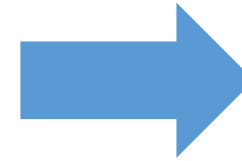
fibonacci

```
souffle -F. -D. fibonacci.dl
```

```
fibonacci.dl
```

```
.decl fib(key:number, value:number)
.output fib

fib(1, 1).
fib(2, 1).
fib(id+2, x+y) :- fib(id, x), fib(id+1, y), id <= 13.
```



```
fib.csv
```

1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21
9	34
10	55
11	89
12	144
13	233
14	377
15	610



# Outline: T1-4: Datalog

- Datalog
  - Datalog rules
  - **Recursion**
  - Recursion in SQL [moved here from T1-U1: SQL]
  - Semantics
  - Datalog<sup>¬</sup>: Negation, stratification
  - Datalog<sup>±</sup>
  - Stable model semantics (Answer set programming)
  - Datalog vs. RA
  - Naive and Semi-naive evaluation (incl. Incremental View Maintenance)

# What is a principled process to determine if a program is recursive?



1

`Local(x) :- Person(x,y,'MA').`

`Relative(x,x) :- Person(x,y,z).`

`Relative(x,y) :- Relative(x,z),Parent(z,y).`

`Relative(x,y) :- Relative(x,z),Parent(y,z).`

`Relative(x,y) :- Relative(x,z),Spouse(z,y).`

`Invited(y) :- Relative('myself',y),Local(y).`

?

2

`Local(x) :- Person(x,y,'MA').`

`Relative(x,x) :- Person(x,y,z).`

`Invited(y) :- Relative('myself',y),Local(y).`

?

3

`MayLike(x,y) :- Close(x,z),Likes(z,y).`

`Visit(x,y) :- MayLike(x,y).`

`Close(x,z) :- Visit(x,y),Visit(z,y).`

?

# Dependency Graph

- The **dependency graph** of a Datalog program is the directed graph  $(V,E)$  where
  - $V$  is the set of **IDB** predicates (relation names)
  - $E$  contains an **arc**  $S \rightarrow T$  whenever there is a rule with  $T$  in the head and  $S$  in the body
- A Datalog program is **recursive** if its dependency graph contains a **cycle**

# Which of these programs is recursive?



1

```
Local(x) :- Person(x,y,'MA').
```

```
Relative(x,x) :- Person(x,y,z).
```

```
Relative(x,y) :- Relative(x,z),Parent(z,y).
```

```
Relative(x,y) :- Relative(x,z),Parent(y,z).
```

```
Relative(x,y) :- Relative(x,z),Spouse(z,y).
```

```
Invited(y) :- Relative('myself',y),Local(y).
```

?

2

```
Local(x) :- Person(x,y,'MA').
```

```
Relative(x,x) :- Person(x,y,z).
```

```
Invited(y) :- Relative('myself',y),Local(y).
```

?

3

```
MayLike(x,y) :- Close(x,z),Likes(z,y).
```

```
Visit(x,y) :- MayLike(x,y).
```

```
Close(x,z) :- Visit(x,y),Visit(z,y).
```

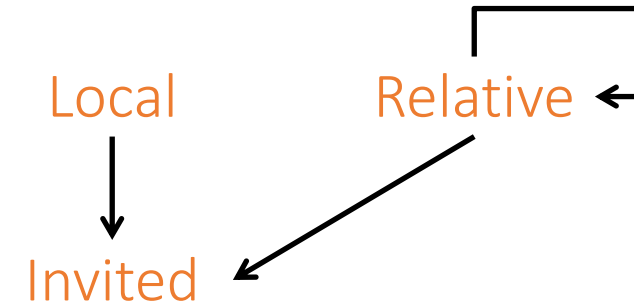
?

# Which of these programs is recursive?



1

```
Local(x) :- Person(x,y,'MA').
Relative(x,x) :- Person(x,y,z).
Relative(x,y) :- Relative(x,z),Parent(z,y).
Relative(x,y) :- Relative(x,z),Parent(y,z).
Relative(x,y) :- Relative(x,z),Spouse(z,y).
Invited(y) :- Relative('myself',y),Local(y).
```



2

```
Local(x) :- Person(x,y,'MA').
Relative(x,x) :- Person(x,y,z).
Invited(y) :- Relative('myself',y),Local(y).
```

?

3

```
MayLike(x,y) :- Close(x,z),Likes(z,y).
Visit(x,y) :- MayLike(x,y).
Close(x,z) :- Visit(x,y),Visit(z,y).
```

?

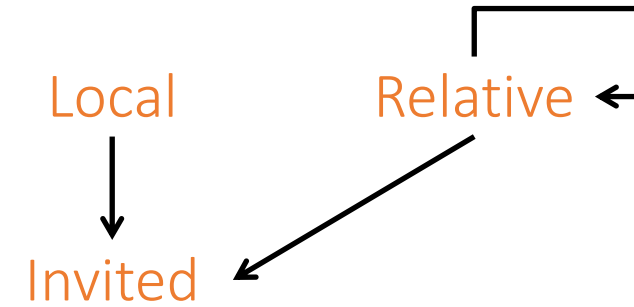


# Which of these programs is recursive?



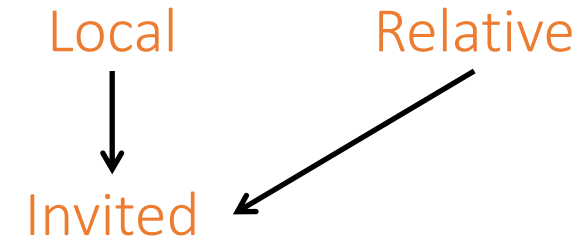
1

```
Local(x) :- Person(x,y,'MA').
Relative(x,x) :- Person(x,y,z).
Relative(x,y) :- Relative(x,z),Parent(z,y).
Relative(x,y) :- Relative(x,z),Parent(y,z).
Relative(x,y) :- Relative(x,z),Spouse(z,y).
Invited(y) :- Relative('myself',y),Local(y).
```



2

```
Local(x) :- Person(x,y,'MA').
Relative(x,x) :- Person(x,y,z).
Invited(y) :- Relative('myself',y),Local(y).
```



3

```
MayLike(x,y) :- Close(x,z),Likes(z,y).
Visit(x,y) :- MayLike(x,y).
Close(x,z) :- Visit(x,y),Visit(z,y).
```

?



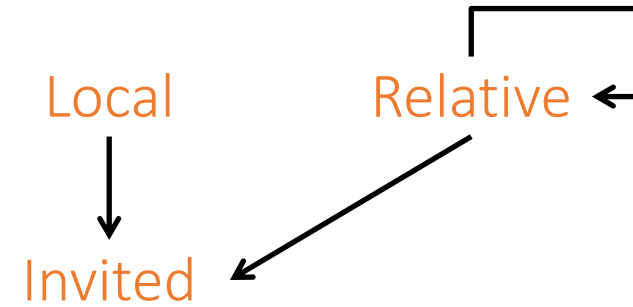
# Which of these programs is recursive?

1

```

Local(x) :- Person(x,y,'MA').
Relative(x,x) :- Person(x,y,z).
Relative(x,y) :- Relative(x,z),Parent(z,y).
Relative(x,y) :- Relative(x,z),Parent(y,z).
Relative(x,y) :- Relative(x,z),Spouse(z,y).
Invited(y) :- Relative('myself',y),Local(y).

```

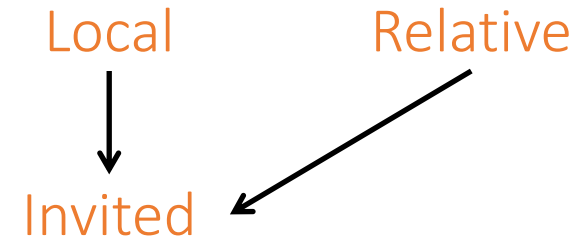


2

```

Local(x) :- Person(x,y,'MA').
Relative(x,x) :- Person(x,y,z).
Invited(y) :- Relative('myself',y),Local(y).

```

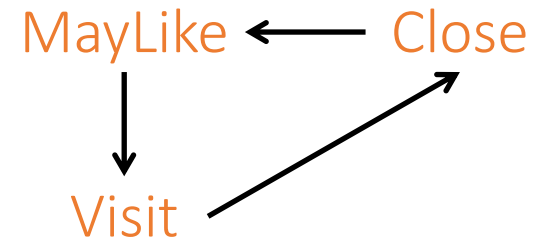


3

```

MayLike(x,y) :- Close(x,z),Likes(z,y).
Visit(x,y) :- MayLike(x,y).
Close(x,z) :- Visit(x,y),Visit(z,y).

```



# Outline: T1-4: Datalog

- Datalog
  - Datalog rules
  - Recursion
  - Recursion in SQL [moved here from T1-U1: SQL]
  - Semantics
  - Datalog<sup>¬</sup>: Negation, stratification
  - Datalog<sup>±</sup>
  - Stable model semantics (Answer set programming)
  - Datalog vs. RA
  - Naive and Semi-naive evaluation (incl. Incremental View Maintenance)



# 1. A simple recursive query

non-recursive part

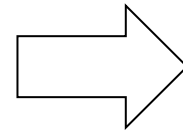
recursive part, contains reference to the query's output

same as "select 1"

```

WITH RECURSIVE T(n) as (
  values (1)
 UNION ALL
  select n+1
  from T
  where n<=3)
SELECT n FROM T

```



	n	
	integer	🔒
1		1
2		2
3		3
4		4

Step	Results	Intermediate / Working Table
1.		
2.	?	?
3.		
4.		
5.		

## Recursive Query Evaluation

1. Evaluate the non-recursive term. For UNION (but not UNION ALL), discard duplicate rows. Include all remaining rows in the result of the recursive query, and also place them in a temporary working table.
2. So long as the working table is not empty, repeat these steps:
  - a. Evaluate the recursive term, substituting the current contents of the working table for the recursive self-reference. For UNION (but not UNION ALL), discard duplicate rows and rows that duplicate any previous result row. Include all remaining rows in the result of the recursive query, and also place them in a temporary intermediate table.
  - b. Replace the contents of the working table with the contents of the intermediate table, then empty the intermediate table.



# 1. A simple recursive query

non-recursive part

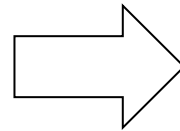
recursive part, contains reference to the query's output

same as "select 1"

```

WITH RECURSIVE T(n) as (
  values (1)
 UNION ALL
  select n+1
  from T
  where n<=3)
SELECT n FROM T

```



	n	
	integer	🔒
1		1
2		2
3		3
4		4

Step	Results	Intermediate / Working Table
1.	{1}	{1}
2.	{1,2}	{2}
3.	{1,2,3}	{3}
4.	{1,2,3,4}	{4}
5.	{1,2,3,4}	∅

## Recursive Query Evaluation

1. Evaluate the non-recursive term. For UNION (but not UNION ALL), discard duplicate rows. Include all remaining rows in the result of the recursive query, and also place them in a temporary working table.
2. So long as the working table is not empty, repeat these steps:
  - a. Evaluate the recursive term, substituting the current contents of the working table for the recursive self-reference. For UNION (but not UNION ALL), discard duplicate rows and rows that duplicate any previous result row. Include all remaining rows in the result of the recursive query, and also place them in a temporary intermediate table.
  - b. Replace the contents of the working table with the contents of the intermediate table, then empty the intermediate table.



## 2. Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13, ...



WITH RECURSIVE Fib as (

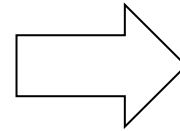
?

UNION ALL

?

SELECT \* FROM Fib

LIMIT 10;



Fib

	n integer	fib <sub>n</sub> integer	fib <sub>n+1</sub> integer
1	0	0	1
2	1	1	1
3	2	1	2
4	3	2	3
5	4	3	5
6	5	5	8
7	6	8	13
8	7	13	21
9	8	21	34
10	9	34	55

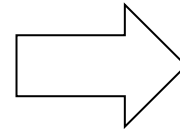
## 2. Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13, ...



```
WITH RECURSIVE Fib as (  
  select 0 as n,  
         0 as "fibn",  
         1 as "fibn+1"  
  UNION ALL
```

?

```
SELECT * FROM Fib  
LIMIT 10;
```



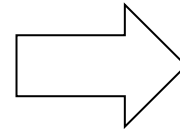
**Fib**

	n integer	fib <sub>n</sub> integer	fib <sub>n+1</sub> integer
1	0	0	1
2	1	1	1
3	2	1	2
4	3	2	3
5	4	3	5
6	5	5	8
7	6	8	13
8	7	13	21
9	8	21	34
10	9	34	55

## 2. Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13, ...



```
WITH RECURSIVE Fib as (  
  select 0 as n,  
         0 as "fibn",  
         1 as "fibn+1"  
UNION ALL  
  select n+1,  
         ?  
        from Fib)  
SELECT * FROM Fib  
LIMIT 10;
```



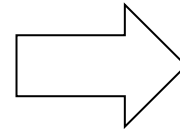
**Fib**

	n integer	fib <sub>n</sub> integer	fib <sub>n+1</sub> integer
1	0	0	1
2	1	1	1
3	2	1	2
4	3	2	3
5	4	3	5
6	5	5	8
7	6	8	13
8	7	13	21
9	8	21	34
10	9	34	55

## 2. Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13, ...



```
WITH RECURSIVE Fib as (  
  select 0 as n,  
         0 as "fibn",  
         1 as "fibn+1"  
UNION ALL  
  select n+1,  
         "fibn+1", ?  
from Fib)  
SELECT * FROM Fib  
LIMIT 10;
```



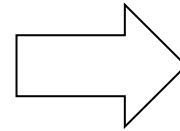
**Fib**

	n integer	fib <sub>n</sub> integer	fib <sub>n+1</sub> integer
1	0	0	1
2	1	1	1
3	2	1	2
4	3	2	3
5	4	3	5
6	5	5	8
7	6	8	13
8	7	13	21
9	8	21	34
10	9	34	55

## 2. Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13, ...



```
WITH RECURSIVE Fib as (  
  select 0 as n,  
         0 as "fibn",  
         1 as "fibn+1"  
UNION ALL  
  select n+1,  
         "fibn+1",  
         "fibn" + "fibn+1"  
  from Fib)  
SELECT * FROM Fib  
LIMIT 10;
```



**Fib**

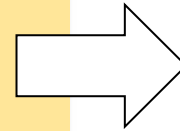
	n integer	fib <sub>n</sub> integer	fib <sub>n+1</sub> integer
1	0	0	1
2	1	1	1
3	2	1	2
4	3	2	3
5	4	3	5
6	5	5	8
7	6	8	13
8	7	13	21
9	8	21	34
10	9	34	55



## 2. Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13, ...



```
WITH RECURSIVE Fib(n,"fib_n","fib_{n+1}") as(  
  select 0, 0, 1  
  
  UNION ALL  
  select n+1,  
         "fib_{n+1}",  
         "fib_n" + "fib_{n+1}"  
  from Fib)  
SELECT * FROM Fib  
LIMIT 10;
```

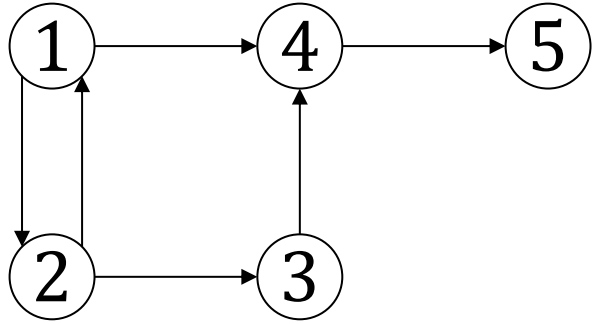


**Fib**

	n integer	fib <sub>n</sub> integer	fib <sub>n+1</sub> integer
1	0	0	1
2	1	1	1
3	2	1	2
4	3	2	3
5	4	3	5
6	5	5	8
7	6	8	13
8	7	13	21
9	8	21	34
10	9	34	55

# 3. Recursion on graphs

*A for directed edges ("arcs")*  $A(S,T)$



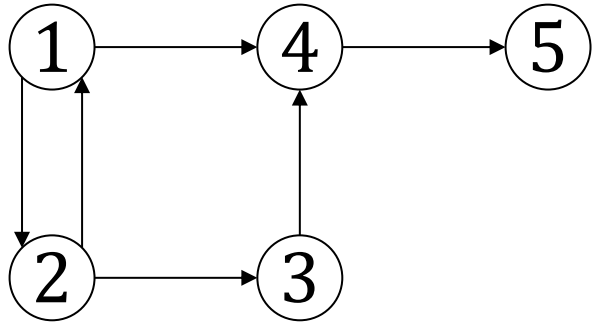
“Find all paths (transitive closure)”

	S	T
A	1	2
	1	4
	2	1
	2	3
	3	4
	4	5

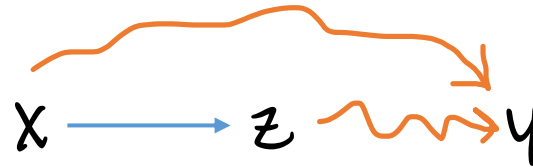


# 3. Recursion on graphs

*A for directed edges ("arcs")*  $A(S,T)$



“Find all paths (transitive closure)”



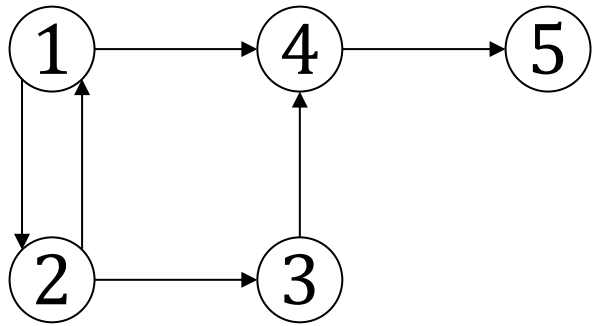
	S	T
A	1	2
	1	4
	2	1
	2	3
	3	4
	4	5

1. Create a path for every arc

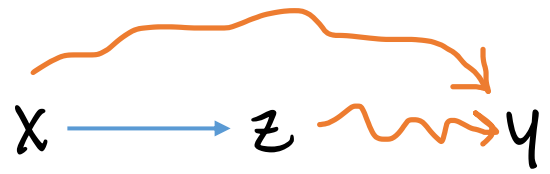
2. An arc + a path can make another path

# 3. Recursion on graphs

A for directed edges ("arcs")  $A(S,T)$



“Find all paths (transitive closure)”



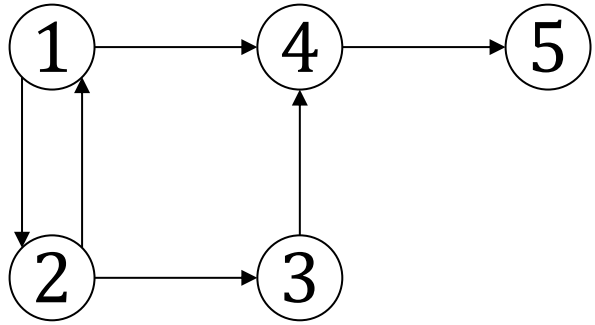
	S	T
A	1	2
	1	4
	2	1
	2	3
	3	4
	4	5

For all nodes x and y:  
If there is an **arc** from x to y,  
then there is a **path** from x to y.

$P(x,y) :- A(x,y).$   
 $P(x,y) :- A(x,z), P(z,y).$

For all nodes x, z, and y:  
If there is an **arc** from x to z, and there is a **path** from z to y  
then there is a **path** from x to y.

# 3. Recursion on graphs



*1<sup>st</sup> iteration*

	S	T
A	1	2
	1	4
	2	1
	2	3
	3	4
	4	5



```
P(x,y) :- A(x,y).  
P(x,y) :- A(x,z), P(z,y).
```

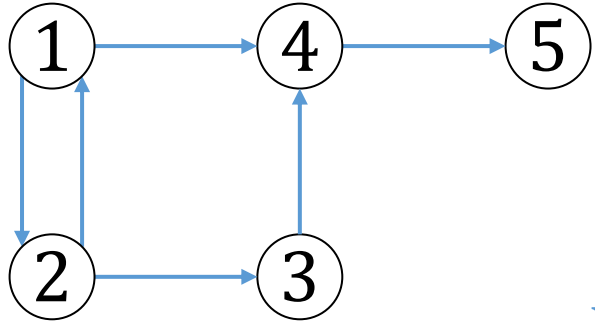
A(S,T)





# 3. Recursion on graphs

A(S,T)



$P(x,y) :- A(x,y).$   
 $P(x,y) :- A(x,z), P(z,y).$

1<sup>st</sup> iteration

2<sup>nd</sup> iteration

A	S	T
	1	2
	1	4
	2	1
	2	3
	3	4
	4	5

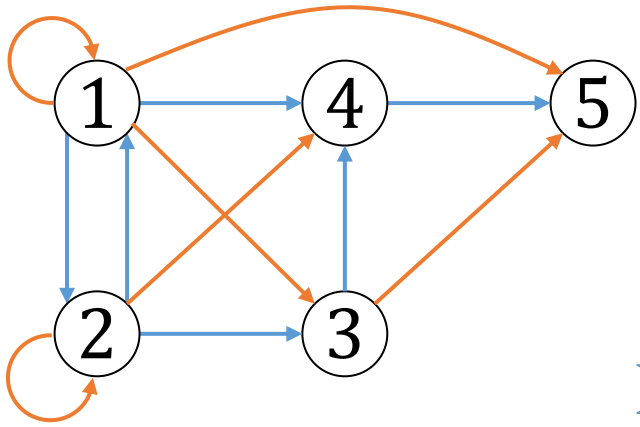
P	S	T
	1	2
	2	1
	2	3
	1	4
	3	4
	4	5

*P=A from 1<sup>st</sup> rule*



# 3. Recursion on graphs

A(S,T)



$P(x,y) :- A(x,y).$   
 $P(x,y) :- A(x,z), P(z,y).$

**A**

	S	T
1	1	2
1	1	4
2	2	1
2	2	3
3	3	4
4	4	5

*1<sup>st</sup> iteration*

**P**

1	2
2	1
2	3
1	4
3	4
4	5

*P=A from 1<sup>st</sup> rule*

*2<sup>nd</sup> iteration*

**P'**

1	2
2	1
2	3
1	4
3	4
4	5

**P**

1	1
2	2
1	3
2	4
1	5
3	5

*1<sup>st</sup> rule*

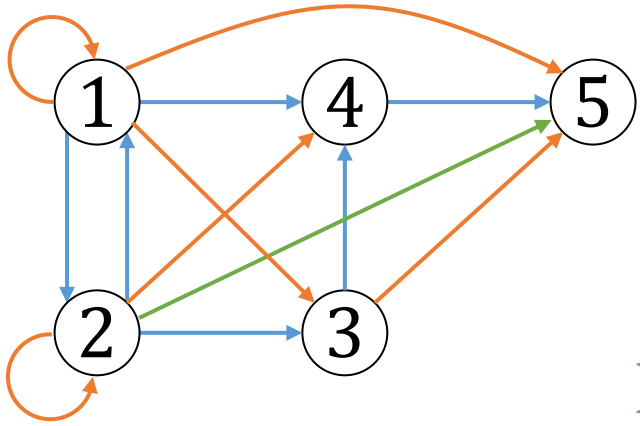
*2<sup>nd</sup> rule*

*3<sup>rd</sup> iteration*



# 3. Recursion on graphs

A(S,T)



$P(x,y) :- A(x,y).$   
 $P(x,y) :- A(x,z), P(z,y).$

**A**

	S	T
1	1	2
1	1	4
2	2	1
2	2	3
3	3	4
4	4	5

1<sup>st</sup> iteration

**P**

1	2
2	1
2	3
1	4
3	4
4	5

$P=A$  from  
1<sup>st</sup> rule

2<sup>nd</sup> iteration

**P'**

1	2
2	1
2	3
1	4
3	4
4	5

1<sup>st</sup> rule

**P**

1	1
2	2
1	3
2	4
1	5
3	5

2<sup>nd</sup> rule

3<sup>rd</sup> iteration

**P'**

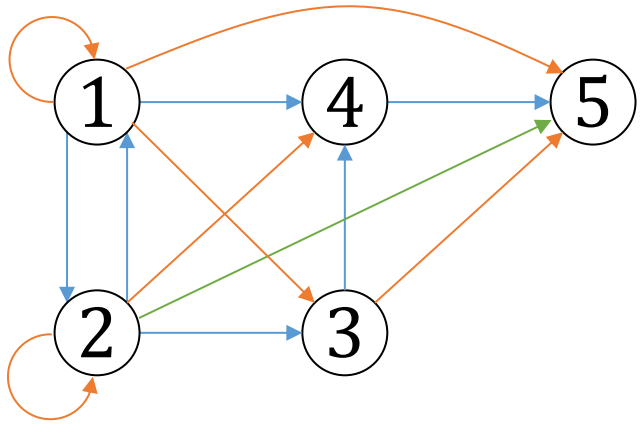
1	1
2	2
1	3
2	4
1	5
3	5

2<sup>nd</sup> rule

**P**

2	5
---	---

# 3. Recursion on graphs



```
P(x,y) :- A(x,y).  
P(x,y) :- A(x,z), P(z,y).
```

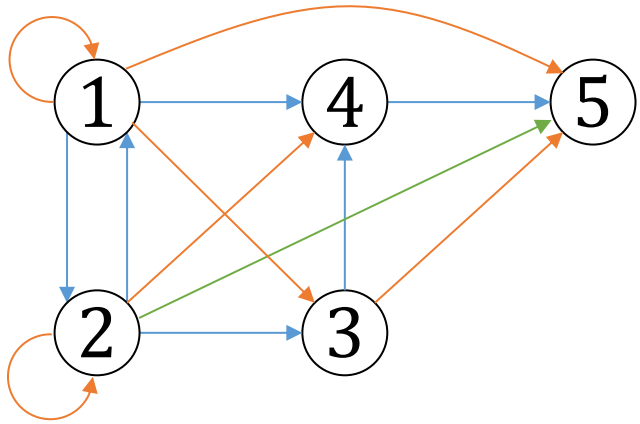
A(S,T)



	S	T
A	1	2
	1	4
	2	1
	2	3
	3	4
	4	5

In SQL ?

# 3. Recursion on graphs



```
P(x,y) :- A(x,y).  
P(x,y) :- A(x,z), P(z,y).
```

A(S,T)



A	S	T
	1	2
	1	4
	2	1
	2	3
	3	4
	4	5

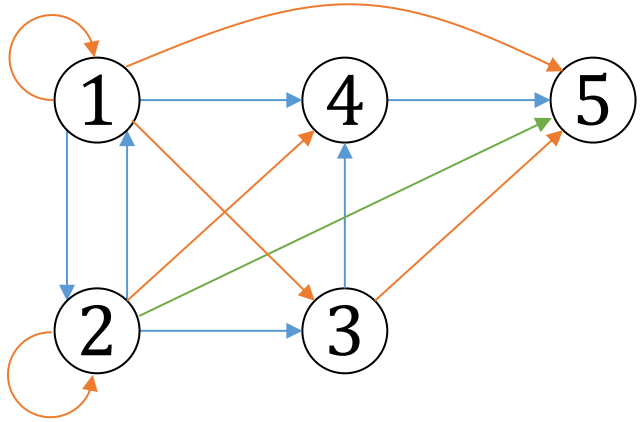
```
WITH RECURSIVE P AS (  
    ?  
    UNION  
    ?  
    SELECT *  
    FROM P
```

# 3. Recursion on graphs

A(S,T)



$P(x,y) :- A(x,y).$   
 $P(x,y) :- A(x,z), P(z,y).$

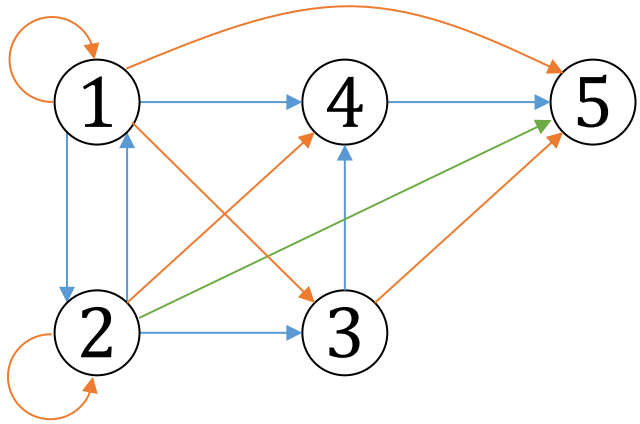


	S	T
A	1	2
	1	4
	2	1
	2	3
	3	4
	4	5

```
WITH RECURSIVE P AS (  
  SELECT S, T  
  FROM A  
  UNION  
  SELECT A.S, P.T  
  FROM A, P  
  WHERE A.T = P.S)  
SELECT *  
FROM P
```

# 3. Recursion on graphs

```
P(x,y) :- A(x,y).
P(x,y) :- A(x,z), P(z,y).
```



**A**

S	T
1	2
1	4
2	1
2	3
3	4
4	5

Strictly speaking, this process is iteration, not recursion:

```
WITH RECURSIVE P AS (
  SELECT S, T
  FROM A
  UNION
  SELECT A.S, P.T
  FROM A, P
  WHERE A.T = P.S)
SELECT *
FROM P
```

Recursion and Iteration both repeatedly execute a set of instructions.

- Recursion (self-similarity) is when a statement in a function calls itself repeatedly.
- Iteration (repetition) is when a loop repeatedly executes until the controlling condition becomes false.

# Challenge



- Write a query that finds the shortest path to each node from a starting node
- Create an interesting minimum database instance
- Show interesting variations
- <https://www.postgresql.org/docs/14/queries-with.html>





# Outline: T1-4: Datalog

- Datalog
  - Datalog rules
  - Recursion
  - Recursion in SQL [moved here from T1-U1: SQL]
  - **Semantics**
  - Datalog<sup>¬</sup>: Negation, stratification
  - Datalog<sup>±</sup>
  - Stable model semantics (Answer set programming)
  - Datalog vs. RA
  - Naive and Semi-naive evaluation (incl. Incremental View Maintenance)

# Semantics of Datalog Programs

- Let  $\mathbf{S}$  be a schema,  $\mathbf{D}$  a database over  $\mathbf{S}$ , and  $\mathbf{P}$  be a Datalog program over  $\mathbf{S}$  (i.e., all EDBs predicates belong to  $\mathbf{S}$ )
- The result of evaluating  $\mathbf{P}$  over  $\mathbf{D}$  is a database  $\mathbf{I}$  over the IDB schema of  $\mathbf{P}$
- We give 2 definitions:
  1. Fixpoint semantics *operative (think procedural)*
  2. model-theoretic *declarative*

# 1. Fixpoint semantics via the chase (operative definition)

Pseudo-code of a **chase** procedure:

Chase(**P**,**D**)

```
I := empty
repeat {
  if(DUI satisfies all the rules of P), then return I
  Find a rule head(x) :- body(x,y) and constants a,b
    s.t. that DUI contains body(a,b) but not head(a)
  I := I ∪ {head(a)}
}
```

*("DUI" is here just a set of tuples)*

Notice since rules are monotone, **I** is also monotonically increasing

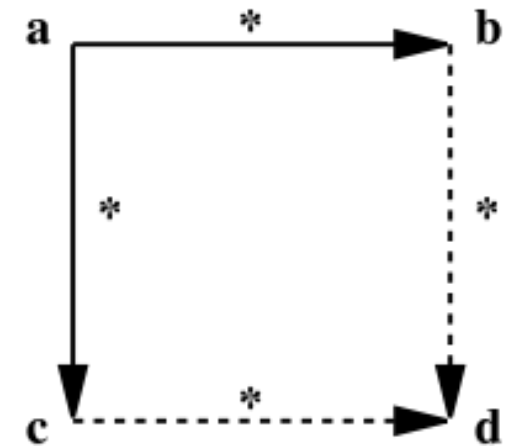
# Nondeterminism

- Note: the chase is underspecified (i.e., not fully defined)
  - There can be many ways of choosing the next violation to handle
  - And each choice can lead to new violations, and so on
- We can view the choice of a new violation as **nondeterministic**

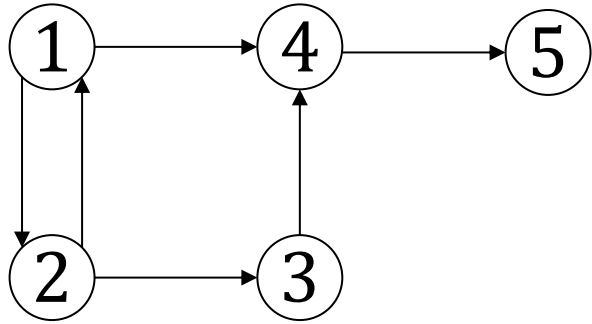
**Church-Rosser property** (defined for term reduction):

*If term **a** can be reduced to both **b** and **c**, then there must be a further term **d** (possibly equal to either **b** or **c**) to which both **b** and **c** can be reduced.*

In computer science, **confluence** is a property of **rewriting** systems, describing which terms in such a system can be rewritten in more than one way, to yield the same result.



# Example



```
Path(x,y) :- Arc(x,y).  
Path(x,y) :- Arc(x,z), Path(z,y).  
Reachable(y) :- Path('1',y).
```

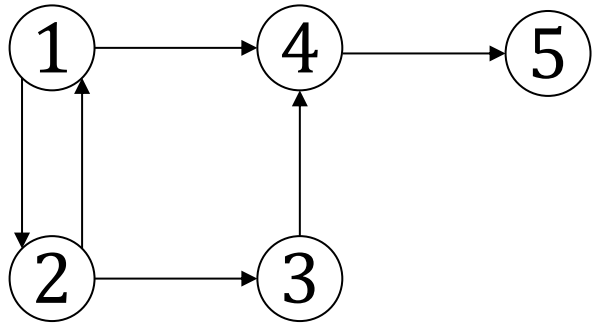
Arc

1	2
2	1
2	3
1	4
3	4
4	5

Path

Reachable

# Example



```
Path(x,y) :- Arc(x,y).  
Path(x,y) :- Arc(x,z), Path(z,y).  
Reachable(y) :- Path('1',y).
```



Arc

1	2
2	1
2	3
1	4
3	4
4	5

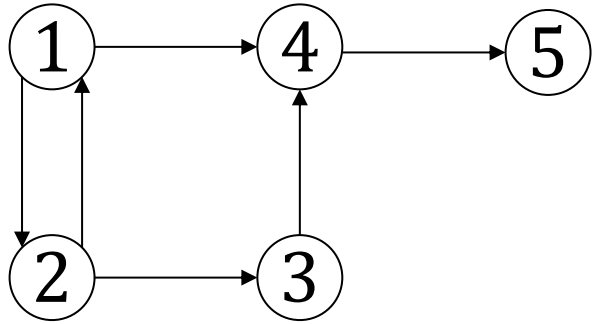


Path

1	2
---	---

Reachable

# Example



```
Path(x,y) :- Arc(x,y).  
Path(x,y) :- Arc(x,z), Path(z,y).  
Reachable(y) :- Path('1',y).
```



Arc

1	2
2	1
2	3
1	4
3	4
4	5

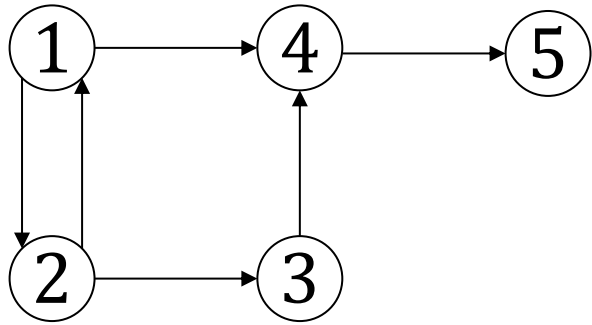


Path

1	2
2	1


Reachable

# Example




```
Path(x,y) :- Arc(x,y).  
Path(x,y) :- Arc(x,z), Path(z,y).  
Reachable(y) :- Path('1',y).
```

Arc



1	2
2	1
2	3
1	4
3	4
4	5

Path

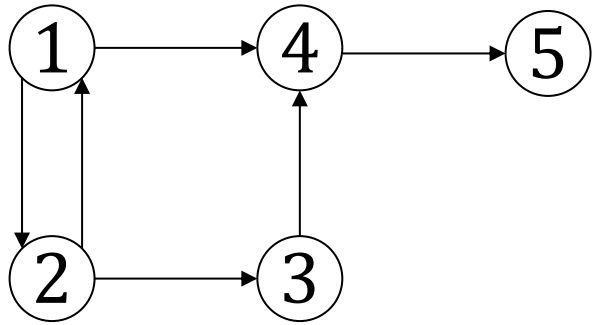


1	2
2	1
2	3

Reachable



# Example



```
Path(x,y) :- Arc(x,y).  
Path(x,y) :- Arc(x,z), Path(z,y).  
Reachable(y) :- Path('1',y).
```



Arc

1	2
2	1
2	3
1	4
3	4
4	5

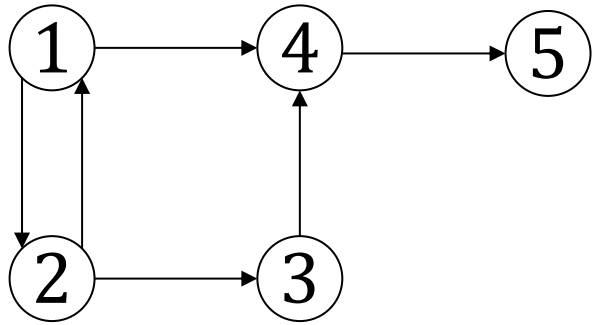


Path

1	2
2	1
2	3
1	1

Reachable

# Example



```
Path(x,y) :- Arc(x,y).  
Path(x,y) :- Arc(x,z), Path(z,y).  
Reachable(y) :- Path('1',y).
```

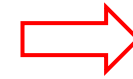
Arc

1	2
2	1
2	3
1	4
3	4
4	5



Path

1	2
2	1
2	3



Reachable

2
---

## 2. Minimal model semantics (model-theoretic definition)

- We say that **IDB I** is a **model** of Datalog program **P** (w.r.t. **EDB D**) if **DI** satisfies all the rules of **P**

$$\forall \text{var} [\text{Head}(\text{IDB}) \leftarrow \text{Body}(\text{EDB}, \text{IDB})]$$

- We say that **I** is a **minimal model** if **I** does not properly contain any other model
- Theorem: there exists one minimal model

# Illustration with our example

```
Path(x,y) :- Arc(x,y).  
Path(x,y) :- Arc(x,z), Path(z,y).
```

## 1. Fixpoint semantics



## 2. Minimal model semantics: smallest Path s.t.

# Illustration with our example

$\text{Path}(x,y) :- \text{Arc}(x,y).$   
 $\text{Path}(x,y) :- \text{Arc}(x,z), \text{Path}(z,y).$

## 1. Fixpoint semantics

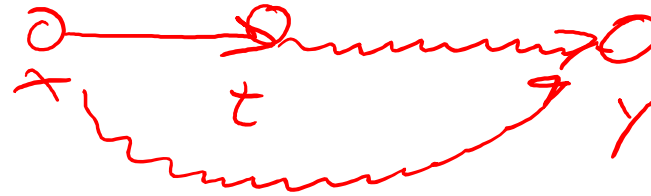
$\text{Path}^{(0)} := \emptyset, t:=0$

Repeat {

inc(t)

$\text{Path}^{(t)}(x,y) := \text{Arc}(x,y) \cup \Pi_{xy}(\text{Arc}(x,z) \bowtie \text{Path}^{(t-1)}(z,y))$

until  $\text{Path}^{(t)} = \text{Path}^{(t-1)}$ }



"immediate consequence operator"

## 2. Minimal model semantics: smallest relation $\text{Path}$ s.t.

# Illustration with our example

$\text{Path}(x,y) :- \text{Arc}(x,y).$   
 $\text{Path}(x,y) :- \text{Arc}(x,z), \text{Path}(z,y).$

## 1. Fixpoint semantics

$\text{Path}^{(0)} := \emptyset, t:=0$

Repeat {

  inc(t)

$\text{Path}^{(t)}(x,y) := \text{Arc}(x,y) \cup \Pi_{xy}(\text{Arc}(x,z) \bowtie \text{Path}^{(t-1)}(z,y))$

until  $\text{Path}^{(t)} = \text{Path}^{(t-1)}$ }

*"immediate consequence operator"*

## 2. Minimal model semantics: smallest relation $\text{Path}$ s.t.

$\forall x,y [\text{Arc}(x,y) \Rightarrow \text{Path}(x,y)] \wedge$

$\forall x,y,z [\text{Arc}(x,z) \wedge \text{Path}(z,y) \Rightarrow \text{Path}(x,y)]$

# Datalog Semantics & equivalence b/w the definitions

(nondeterministic)

1. The **fixpoint** semantics tells us **how** to compute a Datalog query
2. The **minimal model** semantics is more declarative: only says **what** we get

THEOREM: For all Datalog programs  $P$  and DBs  $D$  there is a **unique minimal model**, and **every chase** returns this model

Proof sketch:

1. If  $I_1$  and  $I_2$  are models, so are  $I_1 \cap I_2$
2. Every chase returns a model (finite)
3. Pick a chase and prove by induction: If  $I'$  is a model, then every intermediate  $I$  is contained in  $I'$  (monotonicity)

The minimal model is the *result*, denoted  $P(D)$

**Lemma 8.8** *Model intersection property.* Let  $P$  be a positive program, and  $M_1$  and  $M_2$  be two models for  $P$ . Then,  $M_1 \cap M_2$  is also a model for  $P$ .

**Definition 8.9** *Minimal model and least model.* A model  $M$  for a program  $P$  is said to be a minimal model for  $P$  if there exists no other model  $M'$  of  $P$  where  $M' \subset M$ . A model  $M$  for a program  $P$  is said to be its least model if  $M' \supseteq M$  for every model  $M'$  of  $P$ .

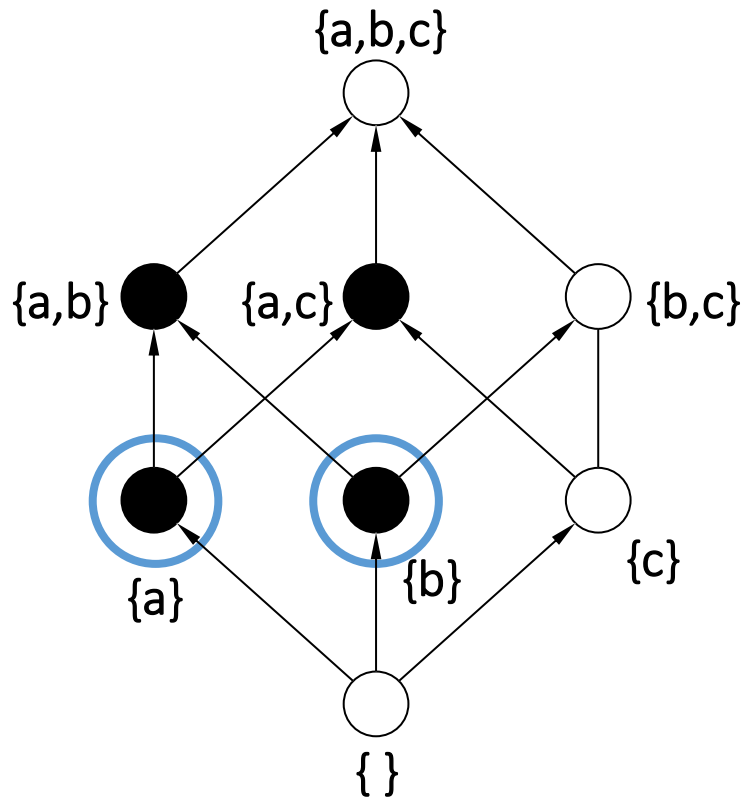
Then, as a result of the last lemma we have the following:

**Theorem 8.10** *Every positive program has a least model.*

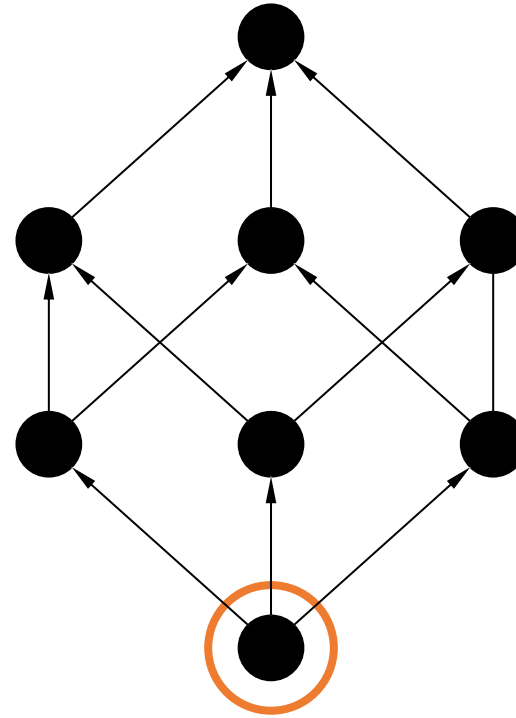
**Proof.** Since  $B_P$  is a model,  $P$  has models, and therefore minimal models. Thus, either  $P$  has several minimal models, or it has a unique minimal model, the least model of  $P$ . By contradiction, say that  $M_1$  and  $M_2$  are two distinct minimal models, then  $M_1 \cap M_2 \subset M_1$  is also a model. This contradicts the assumption that  $M_1$  is a minimal model. Therefore, there cannot be two distinct minimal models for  $P$ .  $\square$

**Definition 8.11** *Let  $P$  be a positive program. The least model of  $P$ , denoted  $M_P$ , defines the meaning of  $P$ .*





2 minimal models



1 least model

# Semantics Summary

## 1. Fixpoint-theoretic

- Most "operational": Based on the immediate consequence operator for a Datalog program.

## 2. Model-theoretic

- Most "declarative": Based on model-theoretic semantics of first order logic. View rules as logical constraints.

# Semantics Summary

## 1. Fixpoint-theoretic

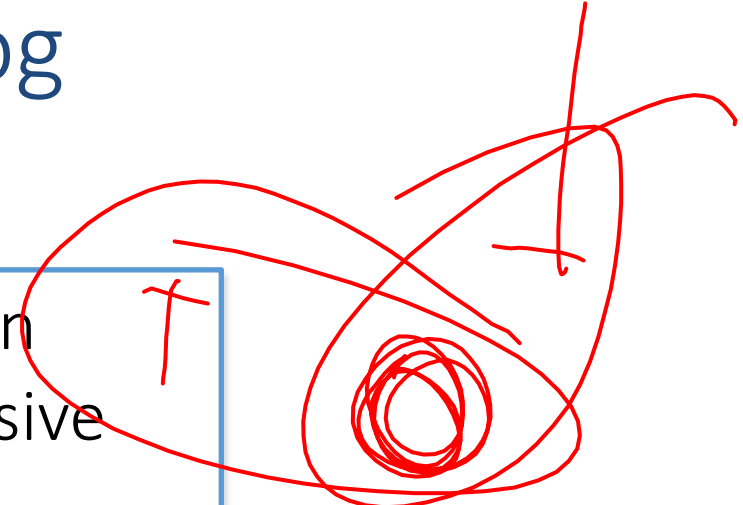
- Most "operational": Based on the immediate consequence operator for a Datalog program.
- Least fixpoint is reached after finitely many iterations of the immediate consequence operator.
- Basis for practical, bottom-up evaluation strategy.

## 2. Model-theoretic

- Most "declarative": Based on model-theoretic semantics of first order logic. View rules as logical constraints.
- Given input DB  $D$  and Datalog program  $P$ , find the smallest possible DB instance  $D'$  that extends  $D$  and satisfies all constraints in  $P$ .

# Expressiveness of Non-recursive Datalog

THEOREM: **Non-recursive** Datalog with built-in predicates ( $<, >, \leq, \geq, \neq$ ) has the same expressive power as the positive algebra  $\{\sigma, \pi, \times, \cup\}$



If we restrict selection to  $\sigma_=$  (i.e. selection with a single equality), this fragment is also called at times **UCQs** (Union of Conjunctive Queries) or **USPJ** (Union-Select-Project-Join) queries.

# Monotonicity

- Can Datalog express **difference**?
  - Answer: **No!**
- Proof: Datalog is **monotone**, difference is not
  - That is, if **D** and **D'** are such that every relation of **D** is contained in the corresponding relation of **D'** ( $\mathbf{D} \subseteq \mathbf{D}'$ ), then  $\mathbf{P}(\mathbf{D}) \subseteq \mathbf{P}(\mathbf{D}')$

$$\mathbf{D} \subseteq \mathbf{D}' \Rightarrow \mathbf{P}(\mathbf{D}) \subseteq \mathbf{P}(\mathbf{D}')$$

# Outline: T1-4: Datalog

- Datalog
  - Datalog rules
  - Recursion
  - Recursion in SQL [moved here from T1-U1: SQL]
  - Semantics
  - Datalog<sup>¬</sup>: Negation, stratification
  - Datalog<sup>±</sup>
  - Stable model semantics (Answer set programming)
  - Datalog vs. RA
  - Naive and Semi-naive evaluation (incl. Incremental View Maintenance)

# What should be the Semantics?



**Friend**(x,y) :- Likes(x,y), ¬Parent(y,x).

Likes (1,2). Parent (2,1).  
Likes (1,3).



# What should be the Semantics?



$\text{Friend}(x,y) :- \text{Likes}(x,y), \neg \text{Parent}(y,x).$

$\text{Box}(x) :- \text{Item}(x), \neg \text{Box}(x).$

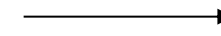
$\text{Likes}(1,2).$   $\text{Parent}(2,1).$

$\text{Likes}(1,3).$



$\text{Friend}(1,3)$

$\text{Item}('ball')$



?



# What should be the Semantics?



$\text{Friend}(x,y) \text{ :- Likes}(x,y), \neg \text{Parent}(y,x).$

$\text{Box}(x) \text{ :- Item}(x), \neg \text{Box}(x).$

$\text{LeftBox}(x) \text{ :- Item}(x), \neg \text{RightBox}(x).$   
 $\text{RightBox}(x) \text{ :- } \neg \text{LeftBox}(x).$

Likes (1,2). Parent (2,1).

Likes (1,3).  $\longrightarrow$  Friend(1,3)

Item('ball')  $\longrightarrow$  Box('ball') ???

Item('ball')  $\longrightarrow$  ?

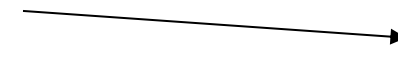
# What should be the Semantics?



$\text{Friend}(x,y) :- \text{Likes}(x,y), \neg \text{Parent}(y,x).$

Likes (1,2). Parent (2,1).

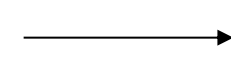
Likes (1,3).



Friend(1,3)

$\text{Box}(x) :- \text{Item}(x), \neg \text{Box}(x).$

Item('ball')

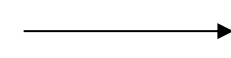


Box('ball') ???

$\text{LeftBox}(x) :- \text{Item}(x), \neg \text{RightBox}(x).$

$\text{RightBox}(x) :- \neg \text{LeftBox}(x).$

Item('ball')



LeftBox('ball') ???

*unsafe!*

$\text{LeftBox}(x) :- \text{Item}(x), \neg \text{RightBox}(x).$

$\text{RightBox}(x) :- \text{Item}(x), \neg \text{LeftBox}(x).$

Item('ball')

?

# What should be the Semantics?



$\text{Friend}(x,y) :- \text{Likes}(x,y), \neg \text{Parent}(y,x).$

Likes (1,2). Parent (2,1).

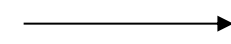
Likes (1,3).



Friend(1,3)

$\text{Box}(x) :- \text{Item}(x), \neg \text{Box}(x).$

Item('ball')

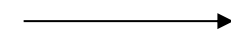


Box('ball') ???

$\text{LeftBox}(x) :- \text{Item}(x), \neg \text{RightBox}(x).$

$\text{RightBox}(x) :- \neg \text{LeftBox}(x).$

Item('ball')



LeftBox('ball') ???

*unsafe!*

$\text{LeftBox}(x) :- \text{Item}(x), \neg \text{RightBox}(x).$

$\text{RightBox}(x) :- \text{Item}(x), \neg \text{LeftBox}(x).$

Item('ball')



LeftBox('ball')

RightBox('ball')

*⇒ Adding negation to Datalog is not straightforward!*

# What should be the Semantics?



$\text{Friend}(x,y) \text{ :- Likes}(x,y), \neg \text{Parent}(y,x).$

Likes (1,2). Parent (2,1).

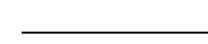
Likes (1,3).



Friend(1,3)

$\text{Box}(x) \text{ :- Item}(x), \neg \text{Box}(x).$

Item('ball')



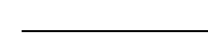
~~Box('ball')~~

no "stable" model

$\text{LeftBox}(x) \text{ :- Item}(x), \neg \text{RightBox}(x).$

$\text{RightBox}(x) \text{ :- } \neg \text{LeftBox}(x).$

Item('ball')



~~LeftBox('ball')~~

unsafe!

$\text{LeftBox}(x) \text{ :- Item}(x), \neg \text{RightBox}(x).$

$\text{RightBox}(x) \text{ :- Item}(x), \neg \text{LeftBox}(x).$

Item('ball')



LeftBox('ball')

RightBox('ball')

Later discussed "stable model" semantics  
(intended models = answer sets)