Database Theory

VU 181.140, SS 2018

2. Introduction to Datalog

Reinhard Pichler

Institute of Logic and Computation
DBAI Group
TU Wien

13 March, 2018

Outline

Motivation

- SQL, relational algebra, relational calculus (both tuple and domain relational calculus) are "relational complete", i.e., they have the full expressive power of relational algebra.
- But: many interesting queries cannot be formulated in these languages
- Example: no recursive queries (SQL now offers a recursive construct)

Example

- Relation `parent(PARENT,CHILD)`, gives information on the parent-child relationship of a certain group of people.
- Problem: look for all ancestors of a certain person.
- Solution: define relation `ANCESTOR(X,Y)`: `X` is ancestor of `Y` by generating one generation after the other (one join and one projection each) and finally merge all generations (union):

grandparent(GRANDPARENT, GRANDCHILD) :=
$\qquad \pi_{1,4}(\texttt{parent[CHILD = PARENT]parent})$
grandgrandparent(GRANDGRANDPARENT,GRANDGRANDCHILD) :=
$\qquad \pi_{1,4}(\texttt{parent[CHILD = GRANDPARENT]grandparent})$

...

ancestor(ANCESTOR,NAME) := parent $\cup$ grandparent $\cup$ grandgrandparent $\cup$ ...

Possible Solution

- Use of a programming language with an embedded relational complete query language:

  begin
    result $:= \{\}$;
    newtuples $:=$ *parent*;
    while *newtuples* $\not\subseteq$ *result* **do**
    begin
        result $:=$ *result* $\cup$ *newtuples*;
        newtuples $:= \pi_{1,4}(\mathit{newtuples}[2 = 1]\mathit{parent})$;
    end;
    ancestor $:=$ *result*
  end.

- procedural, needs knowledge of a programming language, leaves little room for query optimization.

Better Solution: Datalog

- Prolog-like logical query language,
- allows recursive queries in a declarative way
- Example:
  - compute all ancestors on the basis of the relation `parent`
    ```
    ancestor(X,Y) :- parent(X,Y).
    ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).
    ```
  - use the ancestor predicate to compute the ancestors of a certain person (`Hans`):
    ```
    hans_ancestor(X) :- ancestor(X,hans).
    ```
  - compute the ancestors of a certain person (`Hans`) directly:
    ```
    hans_ancestor(X) :- parent(X,hans).
    hans_ancestor(X) :- hans_ancestor(Y), parent(X,Y).
    ```

Datalog - Syntax

```
<datalog_program> ::= <datalog_rule> |
                      <datalog_program><datalog_rule>

<datalog_rule>    ::= <head> :- <body>

<head>            ::= <literal>
<body>            ::= <literal> | <body>, <literal>

<literal>         ::= <relation_id>(<list_of_args>)

<list_of_args>    ::= <term> | <list_of_args>, <term>

<term>            ::= <symb_const> | <symb_var>

<symb_const>      ::= <number> | <lcc> | <lcc><string>
<symb_var>        ::= <ucc> | <ucc><string>

(lcc = lower_case_character; ucc = upper_case_character)
```

Restrictions on the Datalog Syntax

<relation_id>:

- name of an existing relation of the database (`parent`) - can be used only in rule bodies
- name of a new relation defined by the datalog program (`ancestor`)
- has always the same number of arguments.

comparison predicates:

$=$, $<>$, $<$, $>$ are treated like known database relations.

variables:

- each variable that appears in the head of a rule has to be bound in the body
- variables that appear as arguments of comparison predicates must appear in the same body in literals without comparison predicates

A datalog query is also called datalog program

Logical Semantics of Datalog

We consider

R . . . datalog rule of the form $L_0 :- L_1, L_2, \ldots, L_n$,

L_i . . . literal of the form $p_i(t_1, \ldots, t_{n_i})$

$x_1, x_2, \ldots, x_\ell$ variables in R

P . . . datalog program with the rules $R_1, R_2, \ldots, R_m$

We construct

$$R^* = \forall x_1 \forall x_2 \ldots \forall x_\ell ((L_1 \wedge L_2 \wedge \cdots \wedge L_n) \Rightarrow L_0).$$

We assign to each datalog program P the (semantically) well-defined formula P^* as follows:

$$P^* = R_1^* \wedge R_2^* \wedge \cdots \wedge R_m^*$$

We consider now

$\mathtt{REL}$ ... a relation of the database.

$\langle t_1, \ldots, t_n \rangle$ ... a tuple of the relation $\mathtt{REL}$.

$rel(t_1, \ldots, t_n)$ ... a **fact**

DB ... database with relations $\mathtt{REL_1}, \mathtt{REL_2}, \ldots, \mathtt{REL_k}$

We assign to each database relation $\mathtt{REL}$ the formula

$$\mathtt{REL}^* = \text{conjunction of all facts}$$

- a relation is an unordered set of tuples
- the assignment $\mathtt{REL} \mapsto \mathtt{REL}^*$ is therefore not uniquely defined.
- take an arbitrary order (e.g. lexicographical order) since conjunction is associative and commutative.

We assign to each database DB the (semantically) well-defined formula DB^* as follows:

$$DB^* = REL_1^* \wedge REL_2^* \wedge \cdots \wedge REL_k^*.$$

We have:

DB^* is a conjunction of ground atoms (i.e., the facts) and
P^* is a conjunction of formulas with implication

Let G be a conjunction of facts and formulas with implication. Then the set **cons(G)** of facts that follow from G is uniquely defined.

In other words, we have **cons(G)** $= \{A \mid A$ is a fact with $G \models A\}$.

Definition

The semantics of a datalog program P is defined as the function $M[P]$, that assigns to each database DB the set of all facts that follow from the formula "$P^* \wedge DB^*$"

$$M[P] : DB \to cons(P^* \wedge DB^*)$$

Example

Consider the database DB with relations `woman(NAME)`, `man(NAME)`, `parent(PARENT, CHILD)` and the datalog program:

```
grandpa(X,Y) :- man(X), parent(X,Z), parent(Z,Y).
```

woman (NAME)
Grete
Linda
Gerti

man (NAME)
Hans
Karl
Michael

parent (PARENT	CHILD)
Hans	Linda
Grete	Linda
Karl	Michael
Linda	Michael
Karl	Gerti
Linda	Gerti

Let us compute DB^*, P^* and $cons(P^* \land DB^*)$:

$DB^* = REL_1^* \land \cdots \land REL_k^*$ with $\mathtt{REL_i}^*$ = conjunction of all facts

$$\begin{aligned} DB^* \quad = \quad & woman(grete) \land woman(linda) \land woman(gerti) \land \\ & man(hans) \land man(karl) \land man(michael) \land \\ & parent(hans, linda) \land parent(grete, linda) \land \\ & parent(karl, michael) \land parent(linda, michael) \land \\ & parent(karl, gerti) \land parent(linda, gerti). \end{aligned}$$

$P^* = R_1^* \land \cdots \land R_m^*$ with $R_i^* = \forall x_1 \forall x_2 \ldots \forall x_\ell ((L_1 \land \cdots \land L_n) \Rightarrow L_0)$.

$$P^* = \forall X \forall Y \forall Z : ((\mathtt{man}(\mathtt{X}) \land \ \mathtt{parent}(\mathtt{X},\mathtt{Z}) \land \mathtt{parent}(\mathtt{Z},\mathtt{Y})) \ \Rightarrow \\ \mathtt{grandpa}(\mathtt{X},\mathtt{Y})).$$

The new facts in $cons(P^* \wedge DB^*)$:

`grandpa(hans,michael)`, `grandpa(hans,gerti)`.

The datalog program P with

P = `grandpa(X,Y) :- man(X), parent(X,Z), parent(Z,Y)`

defines a new relation `grandpa` with the following tuples:

grandpa (X	Y)
Hans	Michael
Hans	Gerti

Operational Semantics of Datalog

- Datalog rules are seen as inference rules,
- a fact that appears in the head of a rule can be deduced, if the facts in the body of the rule can be deduced.

Example:

facts: `parent(linda,michael)`, `parent(linda,gerti)`
rule: `siblings(michael,gerti) :-`
`parent(linda,michael), parent(linda,gerti).`

the following fact can be deduced:

$$\mathtt{siblings}(\mathtt{michael}, \mathtt{gerti})$$

Rules with variables

- A rule R with variables represents all variable-free rules we get from R by substituting the variables with the constant symbols.
- The constant symbols are taken from the database DB and the program P.
- A variable-free rule resulting from such a substitution is called **ground instance** of R with respect to P and DB
- We write $Ground(R, P, DB)$ to denote the set of all ground instances over P and DB of R.

Example:

Compute all relations between siblings with the following rule:

$$\mathtt{siblings(Y,Z)} :- \mathtt{parent(X,Y)}, \mathtt{parent(X,Z)}, \mathtt{Y} <> \mathtt{Z}.$$

All 6^3 ground instances of this rule with respect to P and DB from above are (Note that there are 6 constant symbols: {grete, linda, gerti, hans, michael, karl}):

$$\begin{aligned}
\mathtt{siblings(grete,grete)} &:- \mathtt{parent(grete,grete)}, \mathtt{parent(grete,grete)},\\
&\quad \mathtt{grete} <> \mathtt{grete} \quad (X = Y = Z = \mathtt{grete})\\
\mathtt{siblings(grete,linda)} &:- \mathtt{parent(grete,grete)}, \mathtt{parent(grete,linda)},\\
&\quad \mathtt{grete} <> \mathtt{linda} \quad (X = Y = \mathtt{grete}, Z = \mathtt{linda})\\
\ldots &\quad \ldots\\
\mathtt{siblings(karl,karl)} &:- \mathtt{parent(karl,karl)}, \mathtt{parent(karl,karl)},\\
&\quad \mathtt{karl} <> \mathtt{karl} \quad (X = Y = Z = \mathtt{karl})
\end{aligned}$$

Idea: execution of a datalog program P on a database DB: iterative deduction of facts until saturation is reached (fixpoint)

Formalization: define a fixpoint operator

- define Operator $T_P(DB)$: augments DB with all facts, that can be deduced in one step by applying the rules from P to DB.

$$T_P(DB) = DB \cup \bigcup_{R \in P} \{L_0 \quad | \quad L_0 :- L_1, \ldots, L_n \in Ground(R; P, DB), \\ L_1, \ldots, L_n \in DB\}$$

- T_P is called the immediate consequence operator.
- $T_P^i(DB) = T_P(T_P^{i-1}(DB))$ iterated application of T_P.

$$
\begin{array}{rcl}
T_P^0(DB) & = & DB \\
T_P^1(DB) & = & T_P(T_P^0(DB)) = T_P(DB) \\
 & = & DB \cup \bigcup_{R \in P} \{L_0 \mid L_0 :- L_1, \ldots, L_n \in \mathit{Ground}(R; P, DB), \\
 & & L_1, \ldots, L_n \in DB\} \\
T_P^2(DB) & = & T_P(T_P^1(DB)) = T_P(T_P(DB)) \\
\ldots & & \ldots \\
T_P^i(DB) & = & T_P(T_P^{i-1}(DB)) = T_P(\ldots T_P(DB)) \\
\ldots & & \ldots
\end{array}
$$

Properties of $T_P(DB)$

- The set of facts is monotonically increasing i.e.:

$$T_P^i(DB) \subseteq T_P^{i+1}(DB)$$

- the sequence $\langle T_P^i(DB) \rangle$ converges finitely: there exists n with $T_P^m(DB) = T_P^n(DB)$ for all $m \geq n$.

- $T_P^\omega(DB)$ ... set of facts, to which $\langle T_P^i(DB) \rangle$ converges is the result of the application of P to DB.

- The operational semantics of a datalog program P assigns to each database DB the set of facts $T_P^\omega(DB)$:

$$O[P] : DB \rightarrow T_P^\omega(DB).$$

Theorem (Equivalence of semantics)

Assume a program P. Then it holds that $M[P] = O[P]$. In other words, for any database DB, we have: $cons(P^* \wedge DB^*) = T_P^\omega(DB)$

Proof of Theorem

Let P be a program and DB a database. We show

$$cons(P^* \wedge DB^*) = T_P^\omega(DB).$$

(1) We first show $T_P^\omega(DB) \subseteq cons(P^* \wedge DB^*)$. By induction on i, we show that $T_P^i(DB) \subseteq cons(P^* \wedge DB^*)$ for every $i \geq 0$. Note that this includes the case where $i = \omega$.

Base case. Assume $i = 0$. Take a fact $L \in T_P^0(DB)$. Then by definition of $T_P^0(DB)$, $L \in DB$. By definition, DB^* is a conjunction of literals and L occurs in it. Hence, by classical logic, $L \in cons(P^* \wedge DB^*)$.

The inductive step. Suppose $T_P^i(DB) \subseteq cons(P^* \wedge DB^*)$ for $i \geq 0$. We show that $T_P^{i+1}(DB) \subseteq cons(P^* \wedge DB^*)$. Recall that $T_P^{i+1}(DB) = T_P(T_P^i(DB))$. Thus by the definition of T_P,

$$T_P^{i+1}(DB) = T_P^i(DB) \cup \bigcup_{R \in P} \{L_0 \quad | \quad L_0 :- L_1, \ldots, L_n \in Ground(R, P, DB),$$
$$L_1, \ldots, L_n \in T_P^i(DB)\}$$

By the induction hypothesis, $T_P^i(DB) \subseteq cons(P^* \wedge DB^*)$. Thus it remains to show that $L_0 \in cons(P^* \wedge DB^*)$ for any rule $R \in P$ such that there is $L_0 :- L_1, \ldots, L_n \in Ground(R, P, DB)$ with $L_1, \ldots, L_n \in T_P^i(DB)$.

Assume such a rule $R = L'_0 :- L'_1, \ldots, L'_n$ in P, and suppose π is the substitution of variables with constants such that applying π to R results in $L_0 :- L_1, \ldots, L_n$, i.e. $\pi(L'_j) = L_j$ for $j \in \{0, \ldots, n\}$.

By construction, in $P^* \wedge DB^*$ we have the conjunct

$$R^* = \forall x_1 \forall x_2 \ldots \forall x_\ell ((L'_1 \wedge L'_2 \wedge \cdots \wedge L'_n) \Rightarrow L'_0).$$

Thus, by employing the semantics of classical logic, for *any* variable substitution π' such that $\{\pi'(L'_1), \ldots, \pi'(L'_n)\} \subseteq cons(P^* \wedge DB^*)$ we also have $\pi'(L'_0) \in cons(P^* \wedge DB^*)$. Since π is a substitution such that $\{\pi(L'_1), \ldots, \pi(L'_n)\} = \{L_1, \ldots, L_n\} \subseteq cons(P^* \wedge DB^*)$ by the induction hypothesis, we get $\pi(L'_0) = L_0 \in cons(P^* \wedge DB^*)$.

(2) We show $cons(P^* \wedge DB^*) \subseteq T_P^\omega(DB)$. To this end, we prove that $L \notin T_P^\omega(DB)$ implies $L \notin cons(P^* \wedge DB^*)$, for any fact L. We thus simply show that $T_P^\omega(DB)$ is a model of $P^* \wedge DB^*$.

This suffices because of the following simple property: if M is a model of a formula F, then any fact $L \notin M$ is not a logical consequence of F (as witnessed by M itself).

$T_P^\omega(DB)$ is a model of DB^* because $DB = T_P^0(DB) \subseteq T_P^\omega(DB)$ by the definition of $T_P^\omega(DB)$.

It remains to show that $T_P^\omega(DB)$ is also a model of P^*. Consider an arbitrary rule $R \in P$. We have to show that $T_P^\omega(DB)$ is a model of R^* with $R^* = \forall x_1 \forall x_2 \ldots \forall x_\ell ((L_1 \wedge L_2 \wedge \cdots \wedge L_n) \Rightarrow L_0)$.

Consider an arbitrary (ground) variable assigment π on the variables $x_1, \ldots, x_\ell$. The only non-trivial case is that all facts $\pi(L_1)$, ..., $\pi(L_n)$ are true in $T_P^\omega(DB)$, i.e., $\{\pi(L_1), \ldots, \pi(L_n)\} \subseteq T_P^\omega(DB)$.

We have to show that then also $\pi(L_0)$ is true in $T_P^\omega(DB)$, i.e., $\pi(L_0) \in T_P^\omega(DB)$.

We know $\pi(L_0) :- \pi(L_1), \ldots, \pi(L_n) \in \mathit{Ground}(R, P, DB)$. Thus by the definition of T_P, $\pi(L_0) \in T_P(T_P^\omega(DB))$. Since $T_P(T_P^\omega(DB)) = T_P^\omega(DB)$ by the definition of $T_P^\omega(DB)$, we obtain $\pi(L_0) \in T_P^\omega(DB)$.

Algorithm: INFER

INPUT: datalog program P, database DB
OUTPUT: $T_P^\omega(DB) \quad (= cons(P^* \wedge DB^*))$

STEP 1. $GP := \bigcup_{R \in P} Ground(R; P, DB)$,
(* GP ... set of all ground instances *)

STEP 2. $OLD := \{\}$; $NEW := DB$;

STEP 3. **while** $NEW \neq OLD$ **do begin**
$OLD := NEW$; $NEW := ComputeTP(OLD)$;
end;

STEP 4. output OLD.

Subroutine ComputeTP

INPUT: Set of facts OLD
OUTPUT: $T_P(OLD)$

STEP 1. $F := OLD$;

STEP 2. **for each** rule $L_0 :- L_1, \ldots, L_n$ in GP **do**
if $L_1, \ldots, L_n \in OLD$
then $F := F \cup \{ L_0 \}$;

STEP 3. return F;

Example

Apply the following program P to calculate all ancestors of the above given database DB.

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).
```

Step 1. (INFER) build GP

$GP = \{$
```
ancestor(grete,grete) :- parent(grete,grete),
ancestor(grete,linda) :- parent(grete,linda),
  ...,
ancestor(grete,grete) :- parent(grete,grete),
                         ancestor(grete,grete),
ancestor(grete,grete) :- parent(grete,linda),
                         ancestor(linda,grete),
  ...
```
$\}$.

(There are $6^2 + 6^3 = 252$ ground instances.)

Step 2. $\mathit{OLD} := \{\}$, $\mathit{NEW} := \mathit{DB}$;

Step 3. $\mathit{OLD} \neq \mathit{NEW}$

Cycle 1: $\mathit{OLD} := \mathit{DB}, \mathit{NEW} := \mathit{TP}(\mathit{OLD}) = \mathit{TP}(\mathit{DB})$
$\mathit{TP}(\mathit{OLD}) = \mathit{OLD} \cup \{\mathtt{ancestor}(\mathrm{A}, \mathrm{B}) \mid \mathtt{parent}(\mathrm{A}, \mathrm{B}) \in \mathit{DB}\}$;

Cycle 2: $\mathit{OLD} := \mathit{TP}(\mathit{DB}), \mathit{NEW} := \mathit{TP}(\mathit{OLD}) = \mathit{TP}(\mathit{TP}(\mathit{DB}))$
$\mathit{TP}(\mathit{OLD}) =$
$\mathit{OLD} \cup \{\mathtt{ancestor}(\mathtt{hans}, \mathtt{michael}), \mathtt{ancestor}(\mathtt{hans}, \mathtt{gerti}),$
$\mathtt{ancestor}(\mathtt{grete}, \mathtt{michael}),\ \mathtt{ancestor}(\mathtt{grete}, \mathtt{gerti})\}$.

Cycle 3: $\mathit{TP}(\mathit{OLD}) = \mathit{OLD}$, there are no new facts

Step 4. Output of OLD.

The result corresponds to the extension of DB with the new table `ancestor` parent

(PARENT	CHILD)
Hans	Linda
Grete	Linda
Karl	Michael
Linda	Michael
Karl	Gerti
Linda	Gerti

ancestor

(ANCESTOR	NAME)
Hans	Linda
Grete	Linda
Karl	Michael
Linda	Michael
Karl	Gerti
Linda	Gerti
Hans	Michael
Hans	Gerti
Grete	Michael
Grete	Gerti

Datalog with negation

- Without negation, datalog is not relational complete because set difference ($R - S$) cannot be expressed.
- We introduce the negation (**not**) in bodies of rules.
- Restriction on the application of the negation:

  A relation R must not be defined on the basis of its negation.

- Check for this constraint: with graph-theoretic methods.

Graph representation

Let P be a datalog program with negated literals in the body of rules

Definition: dependency graph

$DEP(P)$ is defined as the directed graph, with:

- nodes ... predicates (predicate symbols) p in P,
- edges ... $p \to q$, if there exists a rule in P where p is the head atom and q appears in the body (meaning: "p depends on q").

Mark an edge $p \to q$ of $DEP(P)$ with a star "*", if q in the body is negated.

Definition

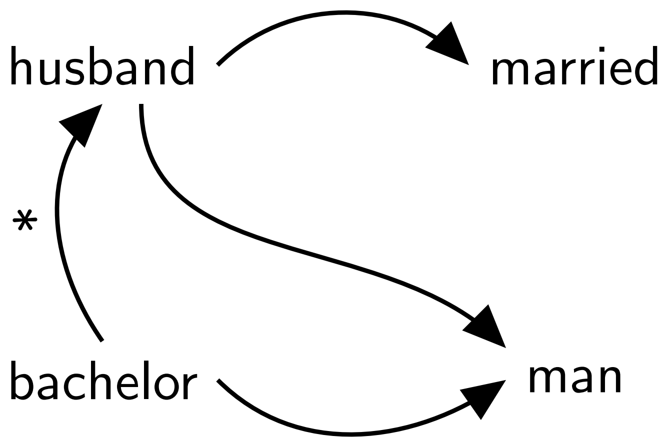
A datalog program P with negation is called valid if the graph $DEP(P)$ has no directed cycle that contains an edge marked with "*".

Such programs are called **stratified**, since they can be divided into strata with respect to the negation.

Example

The following program P with the rules:

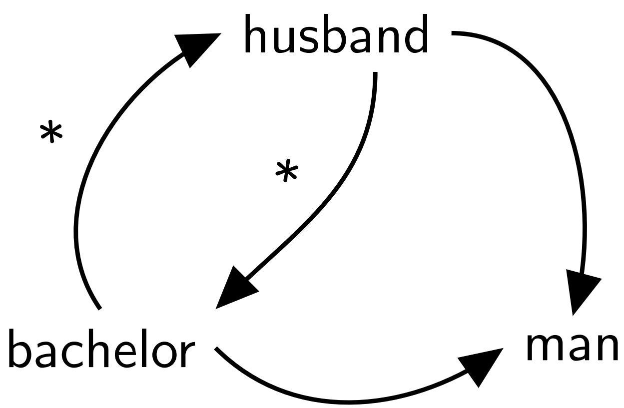
```
husband(X) :- man(X), married(X).
bachelor(X) :- man(X), not husband(X).
```



is stratified.

The program P with the rules:

```
husband(X) :- man(X), not bachelor(X).
bachelor(X) :- man(X), not husband(X).
```



is not stratified.

Stratification

Definition

A stratum is composed by the maximal set of predicates for which the following holds:

1. if a predicate p appears in the head of a rule, that contains a negated predicate q in the body, then p is in a higher stratum than q.
2. if a predicate p appears in the head of a rule, that contains an unnegated (positive) predicate q in the body, then p is in a stratum at least as high as q.

Algorithm

INPUT: A set of datalog rules.

OUTPUT: the decision whether the program is stratified and the classification of the predicates into strata.

METHOD:

1. initialize the strata for all predicates with 1.
2. **do** for all rules R with predicate p in the head:
   - if (i) the body of R contains a negated predicate q,
     (ii) the stratum of p is i, and
     (iii) the stratum of q is j with $i \leq j$, then set $i := j + 1$.
   - if (i) the body of R contains an unnegated predicate q,
     (ii) the stratum of p is i, and
     (iii) the stratum of q is j with $i < j$, then set $i := j$.

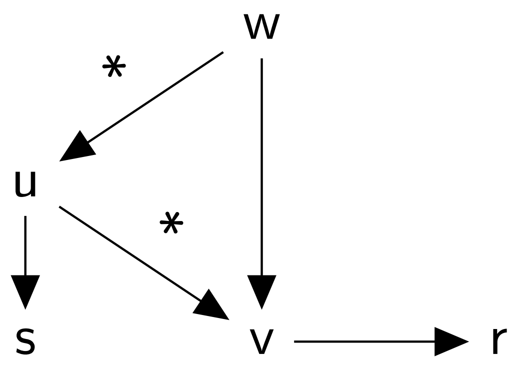
   until:

   - status is stable $\Rightarrow$ program is stratified.
   - stratum $n >$ # predicates $\Rightarrow$ not stratified.

Example

Consider R, S relations of the database DB, P:

```
v(X,Y) :- r(X,X), r(Y,Y).
u(X,Y) :- s(X,Y), s(Y,Z), not v(X,Y).
w(X,Y) :- not u(X,Y), v(Y,X).
```



w	level 3
u	level 2
r, s, v	level 1

Semantics of datalog with negation

Note: when calculating the strata of a datalog program with negation the following holds:

Step 1: computation of all relations of the first stratum.

Step i: computation of all relations that belong to stratum i.

$\Rightarrow$ the relations computed in step $i-1$ are known in step i.

Semantics of datalog with negation is therefore uniquely defined.

Computation of P from the last example above:

Step 1: compute V from R

Step 2: compute U from S and V

Step 3: compute W from U and V

Properties of datalog with negation

- Datalog with negation is relational complete:
  - The difference $D = R - S$ of two (e.g. binary) relations R and S:

    `d(X,Y) :- r(X,Y),` **not** `s(X,Y).`

- syntactical restrictions of datalog with negation:

  all variables that appear in the body within a negated literal must also appear in a positive (unnegated) literal

Example

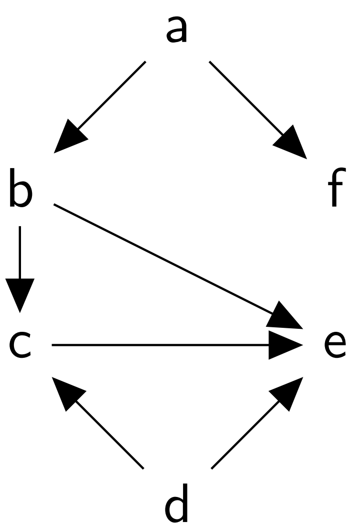
Let *DB* be a database that contains information on graphs, with relations `v(X)`, saying `X` is a node and `e(X,Y)` saying there is an edge from `X` to `Y`. Write a datalog program that computes all pairs of nodes `(X,Y)`, where `X` is a source, `Y` is a sink and `X` is connected to `Y`.

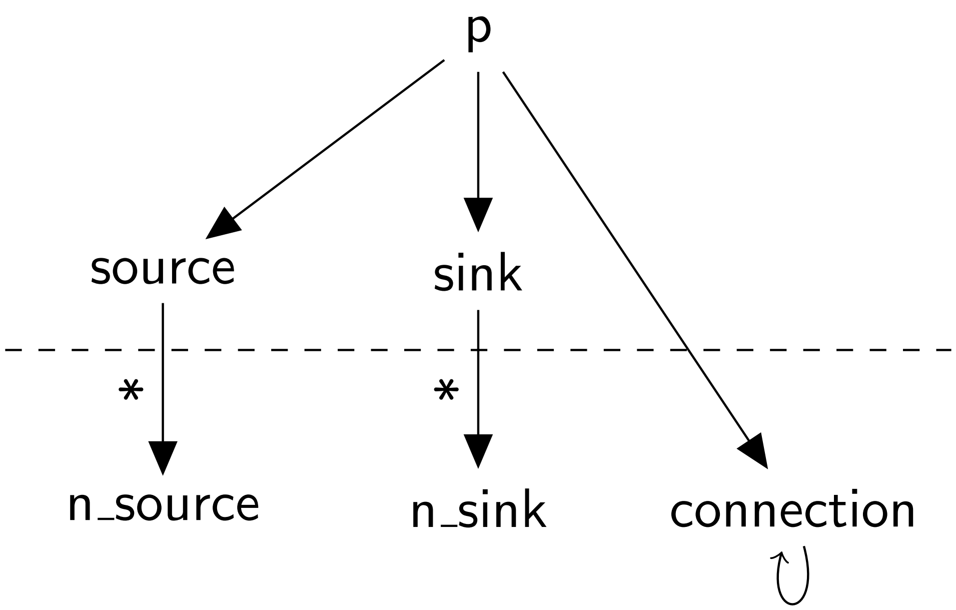
```
p(X,Y) :- source(X), sink(Y), connection(X,Y).

connection(X,X) :- v(X).
connection(X,Y) :- e(X,Z), connection(Z,Y).

n_source(X) :- e(Y,X).
source(X) :- v(X), not n_source(X).

n_sink(X) :- e(X,Y).
sink(X) :- v(X), not n_sink(X).
```





n_source:	b, c, e, f
n_sink:	a, b, c, d
connection:	(a,a), . . . , (f,f), (a,b), (a,c), (a,e), (a,f), (b,c), (b,e), (c,e), (d,c), (d,e)
source:	a, d
sink:	e, f
p:	(a,e), (a,f), (d,e)

Learning objectives

- Motivation for Datalog (recursive queries)
- Syntax of Datalog
- Semantics of Datalog:
  - logical semantics,
  - operational semantics.
- Datalog with negation:
  - the need for negation,
  - the notions of dependency graph and stratification,
  - semantics of Datalog with negation.